VADRE: A Visual Approach to Performance Analysis of Distributed, Real-time Systems

Trevor Harmon and Raymond Klefstad
Department of Electrical Engineering and Computer Science
The Henry Samueli School of Engineering
University of California, Irvine
608 Engineering Tower
Irvine, CA 92697-2625
{tharmon, klefstad}@uci.edu

Published in:
Proceedings of the 2005 International Conference on Modeling, Simulation and Visualization Methods.
Arabnia H. R., Editor. June 2005, Las Vegas, USA. ISBN 1-932415-70-X. 272 pages.

***Abstract*— Distributed, real-time, and embedded (DRE) systems are becoming increasingly complex, and as a result, performance analysis of such systems is becoming increasingly difficult. Current profiling tools are ill-equipped to analyze DRE system performance, primarily due to the distributed nature of these systems. We have begun to address this problem by forging the first in a suite of tools that we call VADRE (vā'dər): *Visual Analysis of Distributed, Real-time, and Embedded systems*. Like a CAT scan for distributed systems, these tools will provide a simplified and highly visual means of inspecting and understanding a system's performance.**

To demonstrate the feasibility and potential benefits of VADRE, we have developed the first tool in the suite. Called *Jango*, it is specialized for the CORBA domain. It can automatically collect timing data from a CORBA-based distributed system and display a timeline of remote method calls. With input from the user, it can run a basic deadline checking algorithm, revealing precisely when and where a deadline is missed. This technique simplifies and quickens the process of testing a distributed system for adherence to real-time constraints. As a case study in validating the capabilities of Jango, we have applied it to a robotic DRE control system and discuss the results here.

I. Background

For stand-alone software, a variety of useful tools for performance analysis have become available in recent years. Valgrind [1], FunctionCheck [2], and JProbe [3], for example, collectively known as profilers, usually come equipped with powerful and mature user interfaces that simplify the process of performance analysis. Armed with these tools, a developer can more easily instrument code under observation, collect timing measurements, and graph the results.

Profiling code in this manner helps uncover hidden inefficiencies that, when compounded, greatly reduce performance. Finding and eliminating these buried bottlenecks naturally makes the software run faster, but it can also lower resource utilization to the point where additional features and services can be added. A cryptographic system, for example, may only be able to support 512-bit encryption because a longer key length would exceed the abilities of the processor. With sufficient profiling, a developer could reduce CPU utilization to a point where the encryption strength could be increased. Thus, performance analysis is vital not just for improving the observed performance of a system; it can also be the catalyst for new capabilities of the software that would not otherwise be possible.

These seemingly sophisticated tools have an Achilles' heel, however. As soon as an application is distributed across a network, today's profilers become ineffective. They are typically designed for monolithic, stand-alone applications, and they fail to address the unique requirements of networked software. While some profilers can attach to processes remotely through a network, this does not solve the problem, as profiling still takes place within a single node. Other tools, such as Ethereal [4], can provide performance analysis of a network by examining individual packets, but this low level of detail is generally not useful for developers of distributed, real-time applications, who need to answer higher-level questions such as:

- Which process in my system is causing the most network congestion?
- Where and when does my system miss its real-time deadlines?
- Why does CPU utilization on this node suddenly rise every three seconds?
- Or simply: What in the world is my distributed system doing?

Today's performance analysis tools are ill-equipped to answer these questions. They simply cannot handle multiple pieces of code on multiple devices, all running in parallel. With this type of concurrency, the system state is unpredictable; performance bottlenecks may occur one day and disappear the next. In the future, the analysis problem will only grow worse: Distributed, real-time systems are becoming increasingly complex, and without better profiling tools, developers will find increasing difficulty in solving performance problems.

II. Motivation

In our research, we have observed first-hand the limitations of current performance analysis tools. We develop middleware for distributed, real-time, and embedded (DRE) systems, and the most sophisticated profiler we have at our disposal is the simple "get current time" command. We can, for example, determine the round-trip time of a remote method call by inserting two such commands, one before and one after the call, and then computing the difference.

In order to analyze the performance of a distributed application that uses our middleware, we must scatter dozens of these command pairs throughout a distributed system, hoping to collect enough timing data that tells us where the bottlenecks lie. It is a tedious and error-prone task: Insert too many statements, and the timing trace becomes nearly unreadable; insert too few, and a vital piece of timing data may be lost. And no matter how effective our logs are, we are still faced with a monumental maintenance problem. Whenever our code changes, our logs must change as well, wasting time and possibly breaking the delicate performance traces we had constructed.

We knew that this method of performance analysis was both inadequate and inefficient, and we have been investigating alternative techniques. Our research in this area has led us to what we believe is the key to effective performance analysis for DRE applications: the power of visualization. With a highly graphical depiction of a system, rather than the text-based approach provided by traditional tools, developers could more easily pinpoint bottlenecks, detect missed deadlines, and ensure that performance requirements are met. Thus, our approach would provide something like a "CAT scan" for the DRE domain, where the patient is the system and the doctor is the developer.

With this motivation in mind, we have taken the initial steps in developing and refining our visual approach to performance analysis. Our goal is to develop a suite of tools, which we refer to collectively as VADRE (v ā'dər), or *Visual Analysis of Distributed, Real-time, and Embedded systems*. They will be analogous to traditional debugging and profiling tools but enhanced for the unique requirements of distributed, real-time systems. With VADRE, the objective is to provide a user-friendly graphical view of highly complex distributed systems, augmented with low-level performance metrics, without overwhelming the developer with stack traces and timing logs.

The visual approach we advocate here should not be confused with the graphical user interfaces already available for many profiling tools. KProf [5], for instance, is a graphical front-end that can translate the flat text dumps generated by gprof [6] or FunctionCheck into two-dimensional function call graphs. Massif [7], another graphical profiler, can produce charts showing heap memory consumption over time. While these graphical depictions can be very useful, they typically provide too much detail. With no application-specific knowledge, they must assume that all data is important, resulting in complex graphs and hierarchies that can bury the user in an avalanche of information. And unlike VADRE, they ignore the problem of distributed applications and generate data only for a single local process.

III. The VADRE Concept

Our visual approach to performance analysis is fundamentally different from existing tools. We want to abolish the traditional, rigid concept of a profiler as nothing more than a histogram of function call frequency. The computing power available today, even in commodity desktop workstations, is capable of much more sophisticated algorithms for analyzing and visualizing performance data. Powerful 3D rendering hardware, for example, is popular for games and scientific visualization, but it is underutilized as a software engineering tool. By applying these highly optimized graphics processors to the task of visualizing performance data, analysis of DRE system behavior could be greatly simplified.

A. Structural Visalization

Instead of text-based logs and the occasional two-dimensional chart, we want to present performance data as a three-dimensional virtual world that the user can explore and view from any angle. Each node in the distributed system would appear as a sphere in this virtual world, and lines running between the spheres would represent connections between nodes (e.g., serial lines or Ethernet cables), as depicted in Figure 1. This representation of the distributed system is potentially more natural and intuitive than traditional performance analysis techniques, for it views the distributed system as a whole, rather than a disparate collection of individual performance metrics.

With lines and spheres representing the structure of the distributed system, we augment this virtual environment with performance data, drawing as much inspiration as possible from the natural world. For example, congested

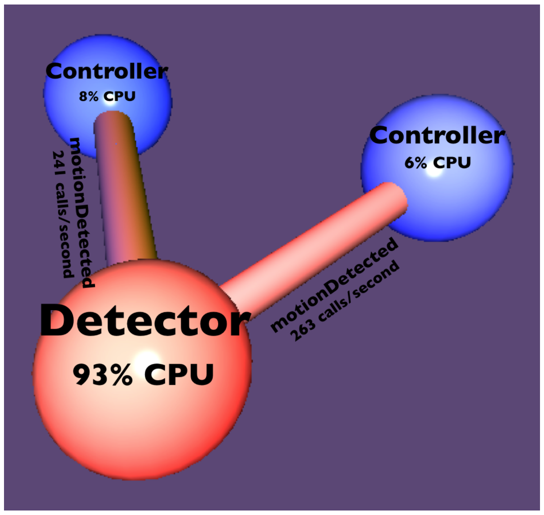

Fig. 1. A visual depiction of a distributed system, such as this concept art of a three-node control system, can make performance analysis a much easier task. With a single glance, one can see that the Detector node is performing most of the work and is in need of optimization. As the number of nodes increases, the distributed system would become more complex, and the size of the performance data would grow, but these intuitive visual cues would remain, making the analysis a more tractable task and reducing reliance on text-based logs.

("hot") resources, such as a network link full of packets, are represented by red colors; underutilized ("cold") resources, such as an idle processor, are blue. Network bandwidth is depicted by line size: low-baud serial lines are thin, while Ethernet connections are fat. Higher CPU utilization on a node increases the size of the sphere; lower utilization decreases it.

With this "bird's-eye-view" of a distributed system, developers could avoid information overload and see at a glance where the system is underperforming. The approach could also reveal interactions between nodes that conventional profilers would miss, such as increased CPU utilization in one area of the system causing network congestion in another.

While a global level of detail is often exactly what a DRE developer needs when profiling a system, situations may arise in which a finer grain of performance data is necessary. For example, a developer may wish to see more data about the performance of a particular node. This local-only level of analysis is not prevented by VADRE; in fact, we believe it should be integrated directly into the VADRE tool suite. A simple mouse click on a sphere, for instance, could open a window of performance data, including function call and memory usage graphs, generated by a traditional profiling tool.

Thus, the VADRE approach is a hybrid of high-level and low-level visualization, eliminating the tedious and error-prone task of sifting through text-based performance data. It combines the best of both worlds: a highly informative yet easily digestible look at the system as a whole, plus fine-grained performance metrics available when needed.

B. Temporal Visualization

As defined in Section III-A, tools based on the VADRE approach address the problem of distributed performance analysis, but they do not provide any profiling of real-time behavior. In this section, we refine the VADRE approach for visualization and analysis of the temporal characteristics of DRE systems.

Temporal analysis is necessary because developers in the DRE domain often must know whether messages passed from one node to another arrive before a specific deadline. Typically, these messages are passed periodically, at regular intervals, and if a missed deadline ever occurs, the entire system may fail. Thus, when the system is in a development and testing phase, it is critical that the developer know exactly where missed deadlines occur and what sequence of events led to the failure.

The conventional approach to this problem is unfortunately quite primitive. As a case in point, we collaborated recently with an aerospace company on a DRE system that required high predictability and extremely low variation in message arrival times. Analyzing this system's temporal performance required the company to field test it and send us long lists of numbers showing the round-trip times of remote method calls. As we optimized the system, the company would re-test it and send us new numbers, but finding evidence of the expected performance increase was a tedious process. Even after calculating the timing deltas between two successive field tests, identifying speed improvements required a time-consuming walk through the logs.

With this experience to guide us, we were inspired to develop a smarter approach. We desired a visual representation of these timing logs, showing us at a glance how well a system is performing. We envisioned a timeline much like the one in Figure 2, where time progresses down along the vertical axis, and messages passed between nodes appear as diagonal lines running between the two axes. This visual depiction would not only facilitate our understanding of the temporal behavior of a system, but it would also allow easy performance comparisons between two competing implementations, simply by overlaying one timeline with another.

In the process of developing the VADRE approach to performance analysis, we realized that the timeline concept could be integrated directly into the tool suite. For example, a VADRE profiler that collects performance data for generating the visualization of Figure 1 would already have enough information to generate the timeline of Figure 2. Therefore, a user navigating the performance visualization could simply select two nodes (spheres)

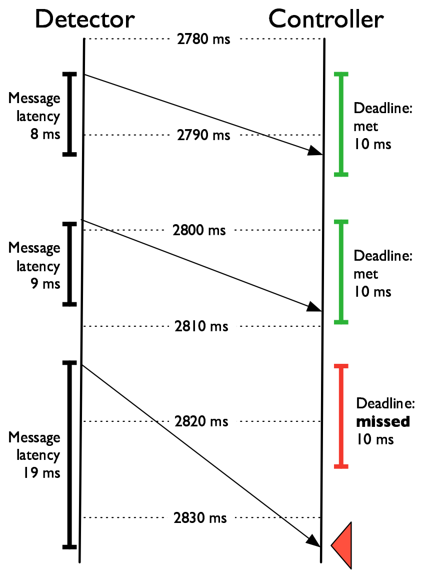

Fig. 2. A timeline such as this one, which features message-passing delays and critical deadlines, shows at a glance how well the system is performing. In this example, the timeline provides the developer with a visual cue—the red arrow—indicating that the system has missed a deadline. Simple but effective, this visual depiction of timeliness is a marked improvement over current techniques of manual, log-based analysis.

of interest and ask the tool to generate a timeline for the nodes automatically. The developer could then scroll through a complete visual history of the interaction between two nodes, providing greater insight into the cause of a failure, for instance. This automatic timeline generation would also enable developers to modify their code and quickly see how the change affects run-time performance of the system and whether it breaks any deadlines.

IV. Jango: VADRE for CORBA

Building the VADRE tool suite described in Section III is a formidable task. DRE systems vary widely in language, operating system, choice of middleware, and overall complexity. Therefore, it is a long-term process to develop tools that will automatically inspect a DRE system, identify its structure, instrument its code to gather performance data, and finally visualize this data.

In the near term, however, we have constructed a first-generation, fully functional prototype of a profiler based upon the VADRE approach. This was made possible by focusing our prototype on a specific domain: CORBA-based [8] applications that conform to the Real-time Specification for Java (RTSJ) [9]. By limiting ourselves to this context, we were able to produce the first VADRE-based performance analysis tool, which we call *Jango*.

Jango works by parsing the structure of a CORBA application and automatically identifying the method calls between remote nodes. It then instruments these methods so that the time of each call is recorded. Next, the developer runs the application, and Jango logs a history of every remote method call. Finally, upon termination of the application, Jango loads the recorded history and displays it according to the timeline concept described in Section III-B.

A. CORBA Facilities for VADRE

In addition to its use of our VADRE approach for visualization, Jango is innovative in the way it harnesses the power of CORBA for performance analysis. It demonstrates two key techniques that show CORBA as a remarkably appropriate platform for visual analysis of DRE systems:

1) CORBA requires the developer to declare inter-object, cross-network method calls in strict Interface Definition Language (IDL) [10] syntax. This restriction greatly simplifies Jango's task of parsing application code and finding remote calls. It also eliminates guesswork: By writing IDL, the developer has already done the work of identifying the important methods that are candidates for performance analysis. Thus, Jango instantly gains application-specific knowledge and can automatically filter out events that are not likely to be of interest to the developer.
2) CORBA provides *portable interceptors* [11], a framework for instrumenting method calls with additional run-time behavior. Traditionally, portable interceptors have been used for adding functionality [12] or for debugging. Jango, on the other hand, exploits these high-level interceptors for collecting performance metrics. This technique is vastly simpler and much less error-prone than attempting to identify method calls at the packet level, which may be required when outside of the CORBA environment.

B. Implementation Challenges

Because IDL provides no information about timing and other real-time constraints, developers must supply these details to Jango manually. This inconvenience is contrary to our VADRE philosophy, which strives for automatic gathering of performance data. Therefore, developing a version of IDL that supports real-time metadata is a key component of our future work in this area, as discussed in Section V.

In addition, Jango has a dangerous dependence on clock synchronization. Performing time comparisons between two nodes, as shown in Figure 2, requires some notion of global time. As a result, high-precision synchronization of the local clock on each node may be necessary for accurate performance analysis and timing

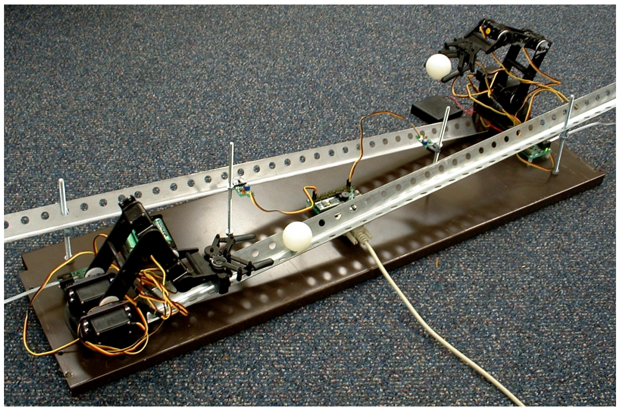

Fig. 3. This distributed real-time control system, consisting of robot arms, light sensors, and a microcontroller, served as a prototype for testing the performance analysis abilities of Jango.

comparisons. Theoretically, true synchronization of this type is impossible due to clock drift, and workarounds such as Lamport's logical clocks [13] must be employed.

In practice, however, we have found that clock synchronization using the Network Time Protocol [14] is sufficient for Jango. For the vast majority of applications, cross-network message-passing deadlines are specified on the order of milliseconds, a large enough margin that NTP can guarantee. For this reason, clock drift can be ignored for VADRE tools such as Jango.

C. *Results and Analysis*

To test Jango's potential as a novel and practical performance analysis tool, we developed a small case study. We constructed a real-time control system of three distributed, concurrently-executing components: two robotic arms and a light sensor. The light sensor interacts with the robotic control system to perform tasks in real-time, such as grabbing a ball as it passes by (see Figure 3). The entire system is controlled by a distributed, CORBA-based application that we developed using ZEN middleware [15] and the RTSJ.

This real-world prototype enabled us to test and refine our implementation of Jango. For example, we were able to record the message flow between the robots and the sensor, then display a timeline of the results. The timeline revealed the average latency in message passing, as well as any instances where the latency exceeded our specified deadlines.

The prototype also helped us discover capabilities in Jango that we had not expected. We learned, for instance, that Jango could help determine the impact of individual hardware components in a DRE system. For example, we could swap the network hub in our system for a different model and then see how the change affects the message-passing delays of Jango's timeline, and therefore which network hub offers the greater performance benefit. Although we could come to the same conclusion without the help of Jango, its VADRE approach makes the task faster, less tedious, and, because of the CORBA facilities described in Section IV-A, it eliminates the need for any manual logging or poring through console traces.

V. Future Work

Naturally, a single test case of our own making is not sufficient to validate the usefulness of both Jango and our VADRE approach to performance analysis. It demands a more rigorous test, such as an experiment among end-users. For instance, two groups of distributed application developers could be provided with faulty, underperforming code; one group analyzes it with traditional tools while the other uses Jango. The speed and ease at which the two groups locate and repair the performance problems in the system could then be compared. Because our VADRE project is still in its infancy, however, we relegate this experiment to future work.

Another component of our future work is to provide support in Jango for real-time extensions to IDL. Such extensions would enable Jango to annotate timelines directly, without user input of deadline constraints, making the entire process of instrumenting, analyzing, and visualizing the performance of a real-time CORBA application completely automatic. Currently, we are considering adopting and building upon the Real-time Multimedia Interface Definition Language (RIDL) [16] for use with Jango.

As our VADRE tool suite expands and matures, we plan to extend it with recording capabilities. These recordings, or "DRE movies," could be shared among colleagues, providing a medium for communication and team collaboration never before possible. For example, the movies could be clipped, showing only some critical failure, then played back as a visual aid for new team members. Such movies could also become an important educational tool, in much the same way as animations of sorting algorithms are used in programming classes today.

VI. Conclusion

Current profilers are adequate for stand-alone software, but they cannot meet the needs of distributed, real-time applications. While some tools, such as the system profiler for QNX Neutrino [17], have attempted to fill this gap, they tend to flood the developer with low-level details, and they lack application-specific knowledge.

We argue that a visually rich profiling tool is necessary, capable of providing a more intuitive, visceral perspective of an application's performance metrics. The VADRE approach is one step toward this goal and should prove useful not just as a profiler but as a reverse-engineering mechanism, helping shed light on the sometimes mysterious inner-workings of distributed, real-time systems.

References

[1] J. Seward, "Valgrind," http://valgrind.kde.org/, 2004.

[2] Y. Perret, "FunctionCheck," http://www710.univ-lyon1.fr/~yperret/fnccheck/profiler.html, 2002.

[3] Q. Software, "JProbe," http://www.quest.com/jprobe/index.asp, 2004.

[4] G. Combs, "Ethereal," http://www.ethereal.com/, 2004.

[5] F. Pillet, "KProf," http://kprof.sourceforge.net/, 2004.

[6] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: a call graph execution profiler," in *SIGPLAN Symposium on Compiler Construction*, 1982, pp. 120–126. [Online]. Available: citeseer.ist.psu.edu/graham82gprof.html

[7] J. Seward, "Massif: a heap profiler," http://developer.kde.org/~sewardj/docs-2.2.0/ms_main.html, 2004.

[8] O. M. Group, "The Common Object Request Broker: Architecture and Specification," 2000.

[9] R.-T. for Java Expert Group, "The Real-Time Specification for Java," 2004.

[10] O. M. Group, "Interface Definition Language," http://www.omg.org/gettingstarted/omg_idl.htm, 2004.

[11] R. Baldoni, C. Marchetti, and L. Verde, "CORBA request portable interceptors: analysis and applications," in *Concurrency and Computation: Practice and Experience*, 2003, pp. 551–579. [Online]. Available: citeseer.ist.psu.edu/baldoni03corba.html

[12] M. Wegdam and A. Halteren, "Experience with CORBA interceptors," in *Workshop on Reflective Middleware*, 2000. [Online]. Available: citeseer.ist.psu.edu/wegdam00experiences.html

[13] L. Lamport, "Time, clocks and the ordering of events in a distributed system," in *Communications of the ACM*, 1978.

[14] D. Mills, "Network time protocol (version 3) specification, implementation and analysis," 1992.

[15] D. C. S. Raymond Klefstad, Arvind S. Krishna, "Design and performance of a modular portable object adapter for distributed, real-time, embedded CORBA applications," in *Distributed Objects and Applications*, 2002.

[16] S. Pope, A. Engberg, and F. Holm, "The real-time (multimedia) interface description language: RIDL," in *IEEE Multimedia Technology and Applications Conference*, 2001.

[17] P. N. Leroux, "System profiling optimizes distributed applications," *EE Times*, 2003.