Automatic Performance Visualization 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, California 92697-2625
{tharmon, klefstad}@uci.edu

Abstract

For distributed real-time systems, adequate profiling tools are exceedingly rare. The sheer variety and low-level nature of these systems impede the adoption of standard, general-purpose tools for performance analysis and visualization. Although much research has been devoted to profiling parallel clusters and supercomputers, the literature virtually ignores the real-time domain. Correspondingly, a handful of commercial tools is available for profiling real-time software, but they invariably make a single-node assumption and are unable to cope with distributed environments.

We examine the state of performance analysis and discuss why profilers are conspicuously absent in the field of distributed real-time systems. We then explore how developers of these systems could benefit from graphical profiling tools with automatic instrumentation and data collection. Toward that end, we demonstrate the prototype of a performance visualization tool called "Bacara," the second addition to our suite of tools for Visual Analysis of Distributed Real-time systems*, or VADR (vā'dər).*

1 Background

Over the last twenty years, software profilers have become commonplace on the programmer's workbench. These tools help focus optimization efforts on selecting the right algorithms and tuning time-critical code, often resulting in large overall performance gains. Profiling software can also help reveal performance problems that may lie far from where programmer's intuition expects them to be, saving development time that would be wasted chasing down false bottlenecks.

Finding and eliminating these buried bottlenecks naturally makes any software run faster, but for real-time systems, the benefits can be dramatic. Specifically, optimization through profiling can lower resource requirements to a point where additional features and services can be added. Consider, for example, a real-time cryptography system that supports only moderate encryption because a longer key length would exceed the abilities of the embedded 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 only for improving the observed performance of a system; it can also be the catalyst for new capabilities of the software that would otherwise be impossible.

For these reasons, profilers are now standard in many desktop software development kits. Such tools have improved substantially since `gprof` [7], the first general-purpose profiler, was introduced in 1982. Since then, performance analysis software has greatly matured and has brought together an impressive assortment of features: detailed graphs of call stack depth, method call frequency, memory allocations, context switches, cache hits and misses, and a variety of other performance metrics. These tools can also augment high-level source code with the relative execution time of each line, showing precisely which areas need optimization. Some of the more sophisticated profilers offer tuning advice (i.e., the ability not just to detect bottlenecks but to suggest how to remove them) and even perform an exhaustive "sub-instruction-level" analysis of the application code, detailing the utilization of each individual functional unit and dispatch slot within the processor.

Despite the complexity and variety of modern profilers, they all collect data in one of two basic ways: *instrumentation* and *sampling*. With instrumentation, profilers inject code into key points of the system (such as at the top of every method call); this code then records the event at run-

Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing
0-7695-2561-X/06 $20.00 © 2006 IEEE

time for subsequent analysis by the profiler. With sampling, the system remains unmodified and is instead halted periodically by the profiler, allowing inspection of such metrics as the amount of memory allocated.

These seemingly powerful techniques have an Achilles' heel. Although they have proven successful for analyzing desktop software and parallel computers, instrumentation and sampling can be enormously problematic for real-time systems. The fundamental flaw is that profilers cannot perform their work in zero time, and thus any injected code or delays due to sampling will alter the temporal behavior of the system under test. In other words, the very act of observation disturbs the system's real-time characteristics in a way that may cause missed deadlines and scheduling conflicts that would not occur under normal execution. This situation is not unlike the effect of observation on quantum particles described by Heisenberg's Uncertainty Principle, leading some researchers to dub the condition a "Heisenbug." [8] Whatever the name, these probe effects are one of the reasons why real-time developers lack adequate profiling tools.

The dilemma does not end there, however. As soon as a real-time system is distributed across a network, the ineffectiveness of today's profilers becomes even more pronounced. Typically designed for monolithic, stand-alone applications, 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,[1] 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 often 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 one node suddenly rise every three seconds?
- Or simply: What in the world is my system doing?

Even the latest, most advanced performance analysis tools are ill-equipped to answer these questions. They simply cannot handle multiple pieces of code on multiple devices, all running independently. 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.

[1] http://www.ethereal.com/

2 Motivation

Through our experience designing middleware for real-time Java, such as RTZen [13], we have observed first-hand the limitations of current performance analysis tools. The most sophisticated profiler at our disposal is the simple `System.currentTimeMillis()` 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, the logging statements may have to 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 of distributed real-time applications: the power of visualization. With a highly graphical depiction of a system—a visual model of the actual nodes and their relationships—developers could more easily examine performance data to pinpoint bottlenecks, detect missed deadlines, and ensure that performance requirements are met. Thus, our approach would provide something like a "CAT scan" for distributed real-time systems, a scenario in which the system is the patient and the developer is the doctor.

With this motivation in mind, we have taken the initial steps in developing and refining our visual approach to performance analysis. We are implementing a suite of profiling tools, which we refer to collectively as VADR (vā'dər), or *Visual Analysis of Distributed Real-time systems*. These tools are analogous to traditional debugging and profiling tools, but they are specialized for the unique requirements of distributed real-time systems. With VADR, 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,[2] for instance, is a graphical front-end that can translate the flat text dumps gener-

[2] http://kprof.sourceforge.net/

ated by `gprof` or FunctionCheck[3] into two-dimensional call graphs. Massif,[4] another graphical profiler, can produce charts showing heap memory consumption over time. While these graphical depictions can be 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 VADR, they ignore the problem of distributed applications and generate data only for a single local process.

3 The VADR Concept

Our goal for performance analysis is fundamentally different from these existing tools. We want to abolish the traditional, rigid concept of a profiler as nothing more than a histogram of function call frequency or a table of some other performance metric. 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 exploiting these highly optimized graphics processors for the task of visualizing performance data, analysis of system behavior as a whole could be greatly improved, as shown by Dwyer's work [5] with three-dimensional UML diagrams (see Figure 1).

Although Dwyer was only interested in structural analysis, we want to apply similar ideas to performance visualization in our VADR research. For example, the inter-node connections of this UML diagram could be augmented with color coding to indicate the utilization—red for high, blue for low—of the links. As the number of nodes increases, the distributed system would become more complex, and the amount of performance data would grow, but these intuitive visual cues would remain easily seen, making performance analysis a more tractable task.

To achieve this goal, we intend to attack the problem of distributed real-time system profiling from two angles: *structural visualization* and *temporal visualization*.

3.1 Structural Visualization

Instead of basic tables and charts, we want to present performance data in a more natural, real-world style. Like Dwyer's 3D UML diagrams, we envision 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

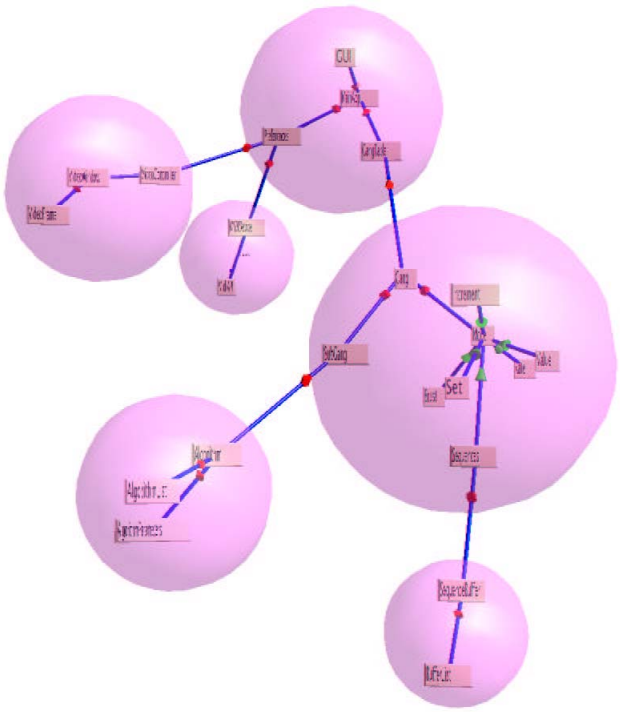

Figure 1. A visual depiction of a distributed system, such as this three-dimensional UML diagram developed by Dwyer [5], can make system analysis a much easier task. Dwyer showed in usability studies that 3D UML diagrams convey information more intuitively and efficiently than their 2D counterparts.

lines running between the spheres would represent connections between nodes (e.g., serial lines or Ethernet cables). 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 the loosely coupled collection of individual performance metrics provided by profilers of today.

With lines and spheres representing the structure of the distributed system, we would augment this virtual environment with performance data, drawing as much inspiration as possible from the natural world. For example, congested ("hot") resources, such as a network link full of packets, would be represented by red colors; underutilized ("cold") resources, such as an idle processor, would be blue. Network bandwidth would be depicted by line size: low-baud serial lines are thin, while Ethernet connections are fat. Higher CPU utilization on a node would increase the size of the sphere; lower utilization would decrease it.

With this "bird's-eye-view" of a distributed system, de-

[3] http://www710.univ-lyon1.fr/ỹperret/fnccheck/profiler.html
[4] http://valgrind.org/docs/manual/ms-manual.html

velopers 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 developer needs when profiling a system, situations may arise in which a finer grain of performance data is necessary. For example, the developer of the distributed system may wish to see more data about the performance of a particular node. This local-only level of analysis would not be prevented by a VADR approach; in fact, we believe it should be integrated directly into a VADR tool suite. A simple mouse click on a sphere, for instance, could open a window showing performance data, including function call and memory usage graphs, generated by a traditional profiling tool as described in Section 1.

Thus, the VADR approach we are working toward is a hybrid of high-level and low-level visualization, eliminating the tedious and error-prone task of sifting through text-based performance data. Although there remain open questions of scalability—for instance, how best to organize and navigate these potentially large-scale visualizations—we believe that this approach will combine 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.

3.2 Temporal Visualization

As defined thus far, tools based on the VADR approach would address the problem of distributed performance analysis, but they would not provide any profiling of real-time behavior. In this section, we refine the VADR approach for visualization and analysis of the temporal characteristics of distributed *and* real-time systems.

Temporal analysis is necessary because developers in the real-time 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. Therefore, 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. (Continuing with the medical analogy of Section 2, one might say that real-time developers need to perform "autopsies" of their systems.)

The conventional approach to this problem is unfortunately quite primitive. As a case in point, our research group recently collaborated with an aerospace company on a distributed real-time 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 in which 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. We took these ideas off the drawing board and produced a working prototype, which we called "Jango," as shown in Figure 2. Jango became the first tool in our VADR suite [9]. Simple but effective, this visual depiction of timeliness is an improvement over the more typical practice of manual, log-based analysis for real-time systems.

4 Bacara: VADR for Real-Time Java

The Jango tool provided temporal analysis of distributed real-time systems, but it did not offer the structural view of performance data described in Section 3.1. For the full VADR approach to performance analysis, we needed to construct an additional tool to complement the features already available in Jango.

Building such a tool is a formidable task. Distributed real-time systems vary widely in language, operating system, choice of middleware, and overall complexity. Therefore, it is a long-term process to develop a general tool that will automatically inspect an arbitrary system, identify its structure, gather performance data, and finally visualize this data.

In the near term, however, the field of distributed real-time systems is starting to homogenize. For example, much of the industry is moving away from the wide assortment of proprietary operating systems and is instead tending toward Linux with real-time extensions. [6] In the academic community, a similar movement is underway to migrate real-time developers away from C in favor of Java and its real-time specification, RTSJ [4]. This trend is fueled by the hope of bringing the productivity and portability advantages of Java into the real-time domain.

The increasing consolidation of tools for real-time de-

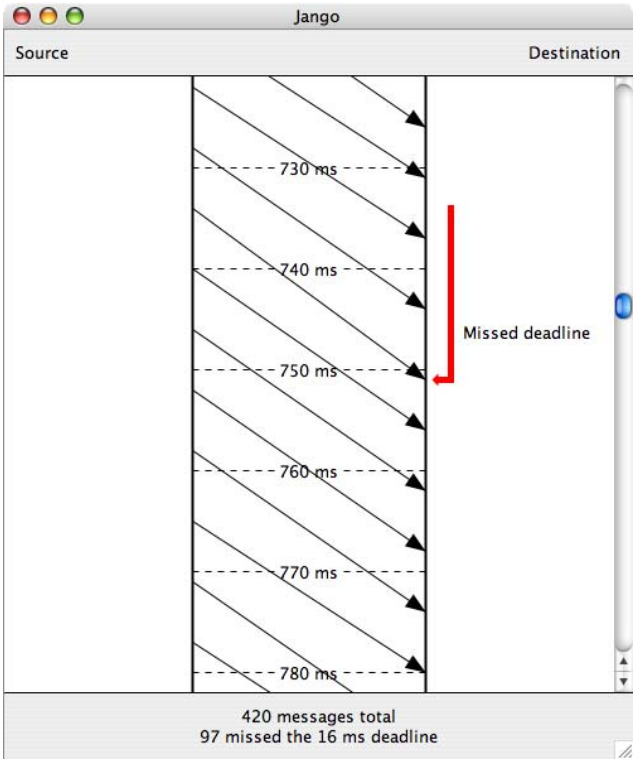

Figure 2. A timeline of real-time system events, including message-passing delays and critical deadlines, reveals at a glance how well a system is performing. As demonstrated by this screenshot of Jango [9], the timeline provides the developer with a visual cue—the red arrow—indicating that the system has missed a deadline.

velopment is a boon for VADR. We believe that by conforming to standards, such as RTSJ and RT-CORBA [1], we can make our performance visualization tools available to a wider audience and more easily collaborate with researchers in our field, speeding development and innovation. We also believe that the RTSJ in particular is on target for bringing real-time system development to a higher level of abstraction, and in much the same way, we want to make performance analysis of real-time systems an easier, more accessible task that does not rely as much on low-level inspection of code.

With these goals in mind, we have constructed a first-generation prototype of a *structural* performance profiler based upon the VADR approach. This accomplishment was made possible by targeting our prototype for a specific domain: CORBA-based [2] applications that conform to RTSJ. By limiting ourselves to this context, we were able to avoid many implementation details and focus instead on the functionality of the software. The result became the second addition to our suite of VADR tools, which we call *Bacara*.

4.1 Implementation

The current implementation of Bacara relies on the instrumentation technique described in Section 1. Although this technique is not ideal for the real-time domain, it is sufficient for our initial prototype, assuming that the instrumentation code is deployed along with the final system and is not removed, as this would affect temporal behavior. Traditional profilers take a similar approach, but these tools typically instrument every available function in the system. When the data is visualized, the developer can easily drown in a sea of information.

Bacara is smarter. As a CORBA-based tool, it exploits the fact that CORBA applications, by definition, have a well-defined structure: Any inter-object or cross-network method calls must be declared in strict Interface Definition Language (IDL) [3] syntax. This requirement gives Bacara a valuable hint about which functions in the system are of principal interest. By writing IDL, the developer has already done the work of identifying the important methods that are candidates for performance analysis and visualization. Thus, Bacara instantly gains application-specific knowledge and can automatically filter events that are not likely to be of interest to the developer.

Armed with this knowledge, Bacara parses the structure of a CORBA application and identifies the method calls between objects and nodes. It then instruments these methods using the Byte Code Engineering Library (BCEL)[5] so that the time of each call can be recorded. Finally, the developer runs the application, and Bacara logs a history of the method calls.

In a non-real-time environment, this history is all that would be needed to process and visualize the performance data for analysis by the developer. It is insufficient for VADR, however, for we are interested in the real-time characteristics of the system under test. Bacara needs more information in order to visualize the most important metric when debugging or optimizing any real-time system: When and where does it miss its critical deadlines?

4.2 Implementation Challenges

Unfortunately, IDL provides no metadata about timing and other real-time constraints. The developer must supply these details to Bacara manually. This inconvenience is contrary to our VADR philosophy, which strives for automatic gathering of performance data. Therefore, developing a version of IDL that supports real-time metadata is a key

[5] http://jakarta.apache.org/bcel/

component of our future work in this area, as discussed in Section 5.

In addition, Jango has a dangerous dependence on clock synchronization. Performing time comparisons between any two nodes of a distributed system 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 comparisons. Theoretically, true synchronization of this type is impossible due to clock drift, and workarounds such as Lamport's logical clocks [10] must be employed.

In practice, we assume that clock synchronization using the Network Time Protocol [11] is sufficient for Bacara. For the vast majority of applications, cross-network message-passing deadlines are typically specified on the order of milliseconds, a large enough margin that NTP can guarantee. For this reason, we ignore clock drift in Bacara's current implementation.

4.3 Case Study

To test Bacara's capacity as a practical performance analysis tool, we developed a small case study. We constructed a distributed real-time system consisting of two Sun Fire servers running Solaris 10 and the 1.0 release of the Mackinac[6] RTSJ virtual machine. Each was connected to the other by an isolated Ethernet local area network. On top of this environment, we added RTZen, configuring one Sun Fire as the client and the other as the server. We then designed a real-time system that simulates the operation of an unmanned aerial vehicle (UAV).

The `UAVServer` application is composed of three CORBA components:

- *Location*—Manages the (virtual) GPS device and responds to queries for the aircraft's current position
- *TargetAcquisition*—Waits for incoming commands to fire on a new target, then carries out the command
- *NoFlyZone*—Prevents the aircraft from flying within certain zones as directed by the ground station

The `UAVClient` application simulates the ground station and mission control entities, both of which act as clients of the UAV. For example, `GroundStation` periodically queries for the aircraft's location, and `MissionControl` asynchronously directs the aircraft to a new target.

Next, we applied Bacara to visualize the performance of this system. The procedure was not entirely automatic because, as explained in Section 4.2, we had to perform the extra step of specifying the round-trip deadlines for each method call. However, this step was fairly simple; we created an XML file containing a list of every function in our IDL, each of which was mapped to a deadline value in milliseconds. Bacara could then read this file and obtain enough information to visualize the performance of the entire system.

The results can be seen in Figure 3. This diagram is the actual output of Bacara showing the performance of the UAV simulation. (To be more precise, Bacara generates output in an XML format readable by OmniGraffle,[7] a vector-drawing application that we use to display the output.) The two purple boxes represent CORBA applications—the Sun Fire client and server in this case—and the rounded rectangles represent CORBA objects within those applications. Each arrow represents method calls from a client to a server.

These interconnects between client and server objects show how Bacara applies the VADR philosophy. In particular, the two visual properties of the arrows describe performance metrics:

- *Size*—The thicker the arrow, the higher the frequency of the method calls between client and server.
- *Color*—Red indicates that at some point during performance monitoring, the round-trip time for this method call missed its deadline. Green indicates that no deadlines were missed; yellow indicates that at least one method call came within 10% of missing its deadline (e.g., 45 milliseconds or higher on a 50-millisecond deadline).

With these visual clues, one can easily see that despite the high utilization of the `Location` object, the `TargetAcquisition` object has missed its deadline and is need of optimization and debugging.

4.4 Analysis

A deduction like this is certainly possible without the help of Bacara. A developer could collect performance measurements manually and arrive at the same conclusion. The difference when using Bacara, as demonstrated by this case study, is that a simple diagram can communicate vital performance characteristics of the system very quickly. Without such a tool, the developer may waste valuable time examining logs and other performance data before arriving at the answer.

A more important benefit highlighted by this example is *automation*. The existing practice of performance analysis for distributed real-time systems is entirely manual. It follows the basic lifecycle shown in Figure 4:

1) The system under test is instrumented with logging code; 2) the performance data is acquired by running the system; 3) the resulting data is preprocessed (e.g., converted into a specific file format more suitable for visualization) and possibly filtered by removing unneeded or unwanted information; 4) and finally, the data is organized and displayed on the computer screen for interactive analysis.

[6] http://research.sun.com/projects/mackinac/

[7] http://www.omnigroup.com/applications/omnigraffle/

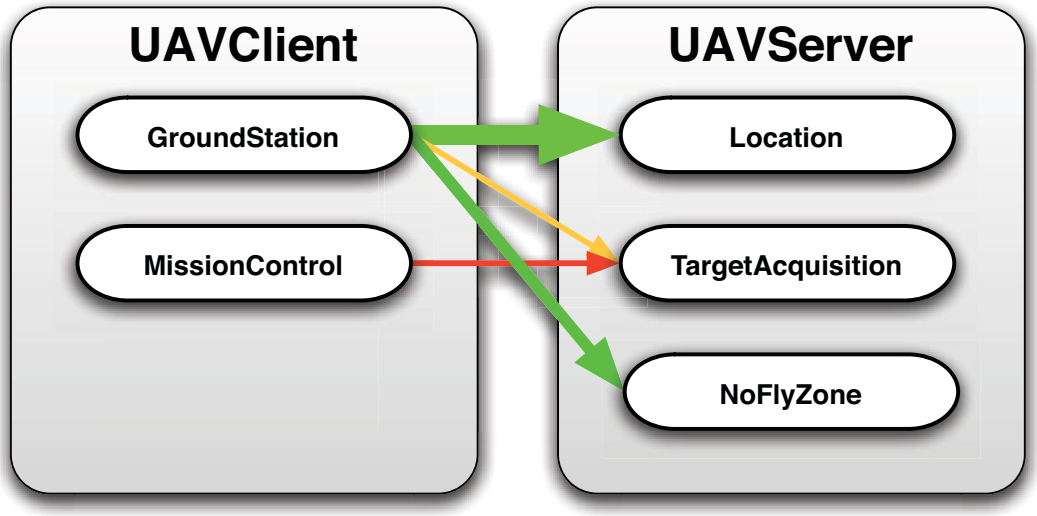

Figure 3. This is actual output of the Bacara tool, as displayed by a helper application called OmniGraffle. It shows the performance of a distributed UAV simulation running under an RTSJ virtual machine. The arrows, which represent method calls between distributed objects, reveal the influence of the VADR approach to performance analysis. For example, red arrows indicate missed deadlines, and thick arrows indicate a large number of method calls. One can quickly deduce from these visual clues that the `TargetAcquisition` object requires debugging and optimization.

These steps are labor-intensive, and whenever the system changes, many of the steps must be repeated. With Bacara, the entire process is automated end-to-end, making the performance analysis task faster, less tedious, and without the need for any *ad hoc* logging or poring through console traces.

5 Future Work

A single test case is not sufficient to validate the usefulness of both Bacara and our VADR approach to performance analysis. We would prefer 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 Bacara. The speed and ease at which the two groups locate and repair the performance problems in the system could then be compared. Because our VADR project is still in its infancy, however, we relegate this experiment to future work.

Our plans for VADR also include:

- *Tool integration*—Currently, the Jango and Bacara tools in the VADR suite are self-contained, but there are advantages in integrating the two. For example, a developer using Bacara to visualize the performance of the system as a whole could select two objects of interest, and Jango would display a message-passing timeline for the objects automatically. The developer could then scroll through a complete visual history of the interaction between two objects, providing greater insight into the cause of a failure.

- *Real-time IDL*—Another component of our future work is to provide support in Bacara for real-time extensions to IDL. Such extensions would enable Bacara to discover deadline constraints without extra user input, making the entire process of instrumenting, analyzing, and visualizing the performance of a real-time system completely automatic. Currently, we are considering adopting and building upon the Real-time (multimedia) Interface Definition Language (RIDL) [12] for use with Bacara. We are also evaluating the annotation feature of Java 5.0 to determine whether it would also be appropriate for specifying worst-case method execution deadlines.

- *Non-intrusive monitoring*—The current version of Bacara reduces the complexity of profiling a distributed real-time system, but as a result of instrumentation, it does not do so without altering the temporal behavior of the system. A relatively large body of work has aimed at solving this "Heisenbug" problem, usually through a hardware-based technique known as Real-time Non-Intrusive (RTNI) monitoring. This

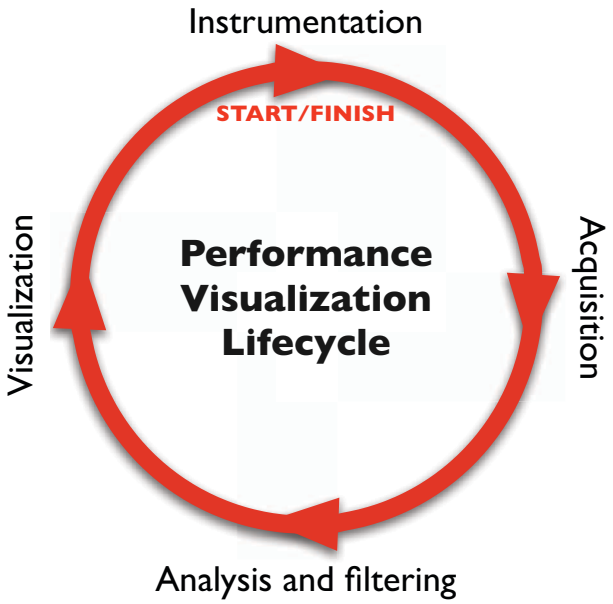

Figure 4. Visualizing the performance of a system is a cyclical process. Developers of distributed real-time systems must perform all of these steps manually, but Bacara automates this visualization lifecycle end-to-end, reducing tedium and allowing the profiling process to become faster, more frequent, and thus more effective.

technique is normally intended for debugging and testing a real-time system, not performance analysis. We are exploring ways of applying RTNI to profiling.

6 Conclusion

Despite the prodigious amounts of research that have been devoted to distributed real-time systems, a remarkably small portion of that effort has gone into learning how best to acquire and visualize the performance of these systems. There are a number of reasons for this discrepancy, including the problem of the "Heisenbug," inherent portability issues, and the current lack of standard frameworks on which to build performance analysis tools.

The increasing consolidation of the foundations for distributed real-time systems—including RTSJ, RT-CORBA, and real-time Linux—is beginning to remedy these problems. We have found, for example, that RTSJ is a remarkably appropriate platform for evaluating new performance visualization techniques. It hides many of the gory details in real-time system development, allowing us to focus on our key problem: building visually rich profiling tools, like Jango and Bacara, based on the VADR concept.

Whatever form these tools take, we argue that they are clearly necessary. The complexities of real-time development demand a more intuitive, visceral perspective of an application's performance metrics. The VADR approach is one step toward this goal and should prove useful not just as a profiler but as a debugger as well, helping shed light on the sometimes mysterious inner-workings of distributed real-time systems.

References

[1] *RealTime-CORBA Specification*. Object Management Group, 2.0 edition, November 2003.

[2] *Common Object Request Broker Architecture: Core Specification*. Object Management Group, 3.0.3 edition, March 2004.

[3] *Common Object Request Broker Architecture: Core Specification*, chapter 3, pages 3–1–3–74. Object Management Group, 3.0.3 edition, March 2004.

[4] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Addison Wesley Longman, January 2000.

[5] T. Dwyer. Three dimensional UML using force directed layout. In P. Eades and T. Pattison, editors, *Australian Symposium on Information Visualisation*, volume 9, pages 77–85, Darlinghurst, Australia, 2001. Australian Computer Society, Inc.

[6] A. Gonsalves. Linux gains support in embedded systems. *InformationWeek*, December 2003.

[7] S. L. Graham, P. B. Kessler, and M. K. McKusick. gprof: a call graph execution profiler. In *SIGPLAN Symposium on Compiler Construction*, pages 120–126, 1982.

[8] J. Gray. Why do computers stop and what can be done about it? Technical Report 85.7, Tandem Computers, 1985.

[9] T. Harmon and R. Klefstad. VADRE: A visual approach to performance analysis of distributed, real-time systems. In H. R. Arabnia, editor, *Proceedings of the 2005 International Conference on Modeling, Simulation and Visualization Methods*, pages 121–126, June 2005.

[10] L. Lamport. Time, clocks and the ordering of events in a distributed system. In *Communications of the ACM*, volume 21, pages 558–565. ACM Press, July 1978.

[11] D. L. Mills. *Network Time Protocol (Version 3) Specification, Implementation and Analysis*. Network Working Group, March 1992.

[12] S. T. Pope, A. Engberg, and F. Holm. The Real-time (multimedia) Interface Description Language: RIDL. In *Multimedia Technology and Applications Conference*, November 2001.

[13] K. Raman, Y. Zhang, M. Panahi, J. A. Colmenares, and R. Klefstad. Patterns and tools for achieving predictability and performance with real-time Java. In *Real-Time Computing Systems and Applications*, August 2005.