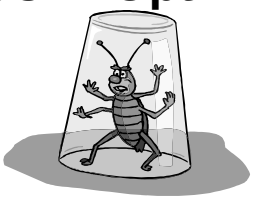
Stuck in the Middle: Challenges and Trends in Optimizing Middleware



Daniel M. Yellin
IBM T. J. Watson Research Center
Hawthorne, NY 10532
dmy@us.ibm.com

1. INTRODUCTION

This paper summarizes the main themes of my talk of the same title at the First ACM Workshop on Optimization of Middleware and Distributed Systems.

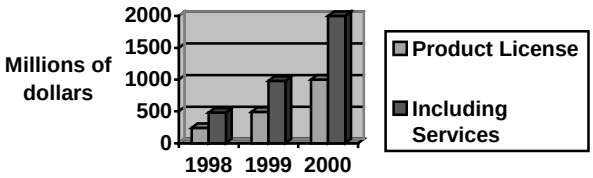
To begin with, lets discuss what we mean by the term "middleware". Exactly what is middleware in the middle of? There are two similar but slightly different approaches to this question. One approach views middleware as being in the middle of a sea of distributed components, such as applications, directories, databases, etc. Middleware serves as the glue that allows all of these components to interact with one another. Using this approach, middleware is seen as enabling the *horizontal* composition of components. The second approach views middleware as the stuff in between an application and lower level services offered by the distributed network. In other words, middleware serves as the abstraction layer that allows an application programmer to easily access distributed services without having to worry about the details of connectivity, protocol conversion, data transformation, load balancing etc. Using this approach, middleware is seen as enabling the *vertical* composition of an application with lower level services. Both approaches are correct and they describe two important functions of middleware: enabling component interaction and simplifying the programming model for application writers.

Concretely, when we speak of middleware we refer to things like Web Application Servers, Object Request Brokers (ORBs), and Message Oriented Middleware (MOM). All of these technologies present programming models to the application programmer and facilitate the collaboration between distributed components. It is hard to define the functionality of middleware precisely as every new middleware product release incorporates additional features.

OM 2001, Snowbird, Utah, USA

1.1 Business drivers: the growth in the middleware market

By all estimates, the market for middleware has seen explosive growth over the last few years. Figure below shows the growth in just one segment of the marketplace, the growth of revenue in integration broker suites. (These numbers are taken from the Gartner Strategic Analysis Report, Application Integration Middleware, September 2000.)



There are several reasons why middleware has experienced such robust growth. First, it greatly increases productivity by encapsulating common tasks in a set of frameworks and components. Second, many recent business trends require a great deal of application integration to succeed, and middleware is the technology of choice when it comes to integration. According to a recent survey (Morgan Stanley CIO Survey, May 2001), the "top strategic software platform projects" companies were undertaking in the coming year is application integration. The business trends driving this phenomenon include enterprise resource planning (ERP), supply chain management (SCM), and customer relationship management (CRM). Consider CRM in the insurance industry as an example. Many insurance companies, in their desire to be *customer-centric*, want to consolidate much of the information about their customers into a single repository and to provide access to integrated customer information, even when that information resides in several different systems. This will enable them, presumably, to better serve their customers. As an example, when a customer calls in with a problem, the service representative should be able to access all of the insurance policies the customer has with the company (life, auto, home…). Similarly, a change of the home address in

a homeowner's policy should trigger a change of address in the auto policy held by the same policyholder – even if these policies are administered on totally different systems.

The Internet has further accelerated the need for middleware. This is because as companies allow consumers and business partners to interact with them on-line, they need to electronically enable business processes, and this requires information and processes to flow across multiple systems. Consider the step-by-step processing associated with on-line shopping. The buyer must first browse a catalog and configure his order. Then she must place her order, thereby causing the order to be confirmed and logged. Next the transaction must be transferred to a fulfillment system, which controls the operational aspects of servicing the order. Of course, payment and financial systems must also be included in the process. All of these steps are usually performed in different specialized systems, requiring a great deal of coordination and integration. The movement towards business-to-business process integration (B2B) will further accelerate the growth in the middleware marketplace.

2. Paradigm Shift: from programs to compusystems

The last section provided motivation for why middleware is becoming so important. We now examine its technical implications. In my view, the "rise in middleware" is having a truly profound impact. It is accelerating a paradigm shift from "micro" programming to "macro" programming. By micro programming, I mean the construction of a single application using a homogenous programming environment (a single programming language, a single mechanism for persisting and retrieving data from secondary storage,...). By macro programming I mean the construction of new applications primarily by integrating existing logic dispersed across various systems, often involving heterogeneous programming environments (multiple programming languages, multiple transactional systems, multiple paradigms for persisting and retrieving data from secondary storage,...).

Evidence of this shift can be found in various sources. One piece of evidence lies in the commercial marketplace for application development tools; more and more products in this marketplace attempt to address the issue of building applications via composition and integration. New IDEs (integrated development environments) provide visual metaphors for composition and contain palettes of components that can be used for common integration tasks. Many products are geared towards providing the developer a uniform programming model by which he can integrate the components without having to understand the underlying technologies upon which these components are built. But perhaps the best evidence for this shift can be found by contrasting project plans used today for large development projects against those used a few years ago. I believe that such an inspection will show that the percentage of time and resources spent on programming application functionality is dramatically decreasing in contrast to the percentage of time and resources spent on integration tasks.

This shift has profound implications not only for the paradigms and tools we present to programmers, but also to the way we think about computing. In this respect it is interesting to make an analogy to nature. Webster defines an *ecosystem* as follows:

A community of animals and plants and the environment with which it is interrelated.

I would like to offer up a parallel definition of a *compusystem*:

A community of software and hardware components and the middleware with which it is interrelated.

By defining a compusystem in this way, I am trying to conjure up the fact that we need to start thinking about computing systems differently. We have to stop thinking about individual applications and individual systems. We need to start viewing the systems we create like ecosystems -- a set of interrelated computing "species". As middleware links more and more systems together, we need better understanding of the consequences of these new-found relationships. When we think of computing in this way, we can begin to ask ourselves if there are common aspects to all compusystems. Here are some of them:

- *They are complex*. The complexity comes from the huge number of components that make up a compusystem. Like an ecosystem, the different component interactions in a compusystem are often hard to understand. A single component in a compusystem can cause another "remote" component to exhibit sporadic pathological behavior, even if there is no direct relationship between the two components. This makes compusystems especially hard to understand. Although we have always built complex systems, the systems we are building today are different. We have built systems out of millions of lines of Cobol in the past, but for the most part these systems were constructed out of a limited number of technologies. In contrast, systems built today are made out of hundreds of heterogeneous components. These systems may contain tens or hundreds or even thousands of different machines, often running different operating systems. They include legacy systems running high volume Cobol and CICS (transactional system) programs. They include legacy as well as relational databases. They often have a multi-tier structure, with HTTP

servers on the front, followed by Web Application Servers, followed by an integration server connecting to back-end applications. No single individual can grasp the end-to-end solution, or anticipate all of the far-reaching effects when hooking these systems together. Additionally, although we built complex systems in the past, these systems were not as ubiquitous as they are today. In a recent survey (Information Week, "Conquering Complexity", April 2, 2001) 95% of the population being surveyed felt that information technology is more complex to manage today than in the past. Mastering this complexity, in my mind, is one of the major challenges of computer science over the next decade.

- *They are forever changing*. Compusystems are dynamic. By this I mean two things. First, like an ecosystem, a compusystem experiences "bio-rhythms". Cyclic events occur during the day, month, and year (caused by such "natural" events as payroll processing) causing the compusystem to experience fixed patterns of interactions. Second, just as ecosystems evolve with climate changes or the introduction of a new species, compusystems evolve as new applications or hardware is added. This implies that understanding a compusystem is not a one-time event, but a continuous process. It will require us to rethink the models we use to analyze systems.
- *They run forever*. There is no such thing as stopping a compusystem to test, debug, and fix it. They are usually required to run 24 by 7, 365 days a year. This has important implications for the way we instrument systems. For example, many program analysis tools have a lot of overhead and affect the application they are analyzing. This makes them poorly suited for live analysis of compusystems. Instead, tools are required that can be deployed "in the field" and have very little effect on the running applications.
- *They have a lot of history* (*calcification*). By running for a long time, compusystems acquire a lot of constraints. This often means that elegant solutions (like adding a field to a record) are impractical (because of the tremendous number of existing data records and APIs that were built without the expectation of the extra field). Maintaining and evolving compusystems will be a major challenge and will seriously impede the adoption of new technologies if not addressed.

3. Optimizing Middleware

In the rest of this paper I would like to focus on the role of middleware in compusystems, as I defined above. In particular, given the transformation of computing towards compusystems, and the fundamental role middleware plays in that transformation, what are the important research directions in middleware? I would like to point out three fundamental research areas:

1. Tools and methodologies for *understanding* compusystems,
2. Development of *new programming models* for building compusystems,
3. Building adaptive middleware that has self-healing and self-optimizing properties.

3.1 Understanding compusystems

We already made the case that understanding compusystems is very difficult. The middleware that unites the many parts of compusystem introduces effects that can be *non-local*, that can *interfere* with one another, and that may only be visible *sporadically* (e.g., when loads are very heavy). It is not enough to thoroughly understand the components of the compusystem nor even pointwise interactions between the components; one needs to understand the characteristics of the compusystem as a whole.

Consider a web transaction processing system. It will typically accommodate many different types of front-end devices (web browsers, "fat" clients, pervasive devices such as cellular phones). It will be multi-tiered, with a front-end http server (processing http requests) passing application requests to an application server (supporting session and state management, presentation and navigation) feeding an integration server (supporting intelligent routing, protocol conversion, and data transformation) that connects to many back-end systems and databases. It will utilize multiple security, authentication, and directory services. It will have to support a variety of communication protocols. Following even a single transaction as it weaves its way through such a system is not an easy task. As thousands or even tens of thousands of transactions flow through these systems each minute, understanding the system as a whole can be overwhelming.

Hence we need tools to help us reason about the dynamic aspects of these systems. One example is a toolkit developed in IBM Research. This toolkit allows you to "spray" *environmental* monitors on many of the components of the compusystem. It allows you to collect, in real time, different *vital signs* of the system; e.g., traditional operating system-level metrics such as CPU utilization, as well as component specific metrics, such as the number of database locks held, the number of threads being used by the http server, or the number of active

MQSeries connections (for a description of MQSeries, the most popular messaging system in commercial use, see *Distributed Computing with MQSeries*, by see Len Gilman and Richard Schreiber, Wiley, September 1996). By capturing this information as the system runs and displaying it (with constant updates) on a screen (we call it the *dashboard*), one can get an immediate understanding of how the system as a whole is operating. Part of the challenge here is figuring out what to display at the highest level – not to overwhelm the architect looking at the dashboard, but to give him a sense of the compusystem as a whole by displaying at first only the most useful information and allowing him to go "deeper" as he unravels the behavior of the compusystem. The toolkit allows one to add additional environmental monitors and also includes a tool that enables fast instrumentation of application code without the need to access source code. Hence this approach is both flexible and customizable.

We have found that displaying information on the dashboard allows the viewer to find correlations between components of the compusystem that otherwise would be hard to detect. For instance, it may be apparent by looking at the dashboard that idle threads in a downstream component are due to the fact that an upstream component doesn't have a matching capacity to generate work. We have also found the toolkit to be an effective ingredient in a performance testing methodology. This methodology uses a *stress tester* to generate increasing workloads on the compusystem while an individual watches the results on the dashboard, thereby uncovering when and where bottlenecks appear. Future research is focusing on how far we can go in automating this process – automatically detecting recognizable patterns of aberrant behavior and even automatically taking corrective actions.

The toolkit illustrates an attempt to understand the dynamic (run-time) characteristics of a compusystem. It is also important to understand its static (code) characteristics. This can be illustrated by another project in IBM Research, called "Asset Locator" (see "eCollabra: An Enterprise Collaboration and Reuse Environment", by Orit Edelstein, Avi Yaeli, and Gabi Zodik, 4th International Workshop, NGITS'99, Zikhron-Yaakov, Israel, July 1999" for an early description of this work). The goal of this project is to facilitate an understanding of the software resources within some large scope (like an enterprise or a compusystem), facilitate an understanding of the relationships among these resources, and facilitate the reuse of software assets in new contexts.

Asset Locator has three fundamental phases: information gathering, repository analysis, and search. Information gathering is like web crawling except it involves code repositories not web pages. In this phase, performed at regularly scheduled intervals, all the code repositories of the compusystem are searched, and each code fragment that is found is "labeled" by its *code type* (e.g., Java, Cobol, HTML, XML, JSP, C++, etc.). The analysis phase categorizes each code fragment discovered in the preceding phase. The categorization rules are type specific; e.g., there are special categorization rules for Java code, for HTML, etc. Based on this categorization, an index into all of the code fragments is built. In the search phase, an end-user can specify the semantic attributes and well as textual key words of the code resource he is interested in. The index is consulted and all matching fragments are retrieved. The user can also browse dependencies between different code fragments.

Asset Locator is a good tool not only for browsing a large distributed code repository and for finding reusable code, it is also useful for *impact analysis* – helping discover how changes to one part of a compusystem may effect another. In this regard, we need to further extend Asset Locator to better identify dependencies between heterogeneous code types. Ultimately this may require not only static analysis of the compusystem, but dynamic analysis as well. Asset Locator shows that it is useful to augment traditional deep analysis of relatively small fragments of code to more shallow analysis of very large code repositories.

3.2 New programming models

The 1990s saw the emergence of several distributed programming models (such as DCE and CORBA). The goals of these models were to facilitate the construction of compusystems. In retrospect, these models were lacking in several ways: they were fairly heavy weight to implement, they required a substantial amount of agreement between the communicating components to work correctly, and they did not accommodate legacy systems very well. Based upon this learning experience, and with the advent of the Internet, a new distributed programming model is emerging. Its key features include the following:

- It is based upon *open standards*, such as XML, Web Services, WSDL, WSFL, UDDI, and others;
- It facilitates the integration of *loosely coupled* systems, with clear separation of interface, content, and business logic and with *minimal connectivity* requirements between the components;
- It supports the *late binding* of components through run-time discovery and dynamic binding mechanisms.

There is much development yet to be done in building the base infrastructure of this new computing model and it is too early to judge its success. But if successful it will certainly make it easier to build, optimize, and evolve compusystems.

To optimize a compusystem, it is also important to minimize the resources associated with the flow of a request

through the system – to minimize the number of components it must touch and the number of transformations it must go through. Any programming model for compusystems must address this issue. One obvious way to limit the flow through a compusystem, at least for web-based informational queries, is to rethink the way caching is done. There are several dimensions to this problem: figuring out where to cache, figuring out what to cache, and coming up with good cache coherence protocols. Once again, let me use a project in IBM Research to explore these issues (see "High-Performance Web Site Design Techniques", Arun Iyengar Jim Challenger, Daniel Dias, and Paul Dantzig, In IEEE Internet Computing, vol. 4 #2, March/April 2000). This project focuses on how to build highly scaleable web sites. Using this infrastructure, for example, we were able to build sites handling 875 million hits per day, with a peak rate of 1.2 million hits per minute, with 100% availability!

Building such robust and scaleable infrastructure requires rethinking the way web pages are put together. This project views each web page as being made up of a hierarchy of embedded fragments. At the most elementary level, each fragment is either a piece of text, a jpeg image, a video clip, etc. These elementary fragments can be combined into a new fragment, which may be recursively combined with other fragments into a yet another fragment. One can also maintain an index mapping each elementary fragment into all the pages containing it. In this way, whenever an elementary fragment becomes obsolete, one can easily find the pages that have become obsolete with it. Additionally, given a time period for each elementary fragment in which it is guaranteed to be valid, one can determine the minimal time period that each page is guaranteed to be valid.

Using this approach, one can begin to cache throughout the layers of the network—at cache servers placed at the origin of the content, at "edge of network" servers, at ISPs, and so forth. Each page cached is given a conservative expiration date, facilitating cache coherence. Whenever an elementary fragment changes, by using the index, new pages can be generated for all affected web pages and propagated to the caching servers. This approach proved to be very effective. For all deployments of this technology, on the average only a small percentage of web hits were processed by the content engines themselves; cache servers somewhere on the network absorbed the vast majority of the requests. This approach shows that a programming model that proactively generates and distributes content upon availability can have significant performance improvements over approaches that lazily generate content upon demand. This methodology may not be applicable to all compusystems. It is simply intended to show the sorts of new approaches we need to take when building a compusystem in order to optimize performance.

3.3 Adaptive systems

We already mentioned that compusystems show different computational patterns at different times. It is therefore often the case that trying to optimize the system for one sort of behavior causes degradations for other behaviors. Furthermore, compusystems evolve and change, so optimizations may loose their effectiveness over time.

An important research direction beginning to address this issue is the development of *adaptive* systems. An adaptive system is one that monitors its behavior at runtime and changes its behavior in order to optimize performance. One example can be found in the work on dynamic data distribution (see "An Algorithm for Dynamic Data Distribution", by Ouri Wolfson and Sushil Jajodia, in Proceedings of the 2nd Workshop on the Management of Replicated Data (WMRD-II), Monterey, CA, Nov. 1992.). In this work, there are many clients trying to access a data object. To optimize performance, one can create replicas of this object. These replicas can critically affect performance, since reading a local replica is less costly than reading a remote data object. On the other hand, updates to the data object become more costly as all the replicas need to be updated. Hence the best strategy depends upon the read/write characteristics of each data object, and these characteristics may change over time. The work of dynamic data distribution shows how to adaptively change the replication scheme in response to the change in read/write patterns to data objects.

Another example of an adaptive system is a project at IBM Research that is applying control theory to server software. In one experiment, the project looked at tradeoffs between the number of clients serviced by a particular server, and the response time for service requests. The goal was to maximize the number clients being processed without slowing up response time below a given threshold value (the maximum delay acceptable was determined empirically by system administrators). The experiment showed that traditional control theory could be used to dynamically converge to an optimal solution.

Both of these examples show very different ways of using adaptive techniques to optimize middleware. As compusystems link more and more systems together, it becomes increasingly harder to statically predict computational patterns and loads on the various components of the system. Adaptive techniques will become more and more important in this environment for achieving high performance.

4. Summary

Middleware is growing very rapidly because it is the glue that binds systems together, and there is great business need in integrating systems today. I have argued that this growth

has foreshadowed the emergence of compusystems-- large systems containing enormous numbers of computational components. The challenges in building, maintaining, and optimizing these systems are great. I have demonstrated three areas of research that promise help in this regard: better models and tools for understanding compusystems, new programming models to help us more effectively construct, maintain, and optimize compusystems, and adaptive middleware for optimizing compusystems based upon the dynamic state of the system.

Acknowledgement

This paper has strongly benefited from discussions I have had with many people. It would be impossible to name them all, but I here acknowledge discussions with those people whose gave me insight into their projects which I referenced in this paper: Paul Dantzig (caching in large web systems), Joe Hellerstein (applying control theory to software servers), Doug Kimelman & Chet Murthy & Darrell Reimer (toolkit for monitoring compusystems), and Gabi Zodik (Asset Locator).