Middleware Challenges Ahead

Kurt Geihs
Goethe University

Facing dynamic modifications in distributed system technology, middleware developers strive to support applications that meet the technical challenges of ubiquitous computing.

In the first attempts to define comprehensive software platforms for distributed applications 25 years ago, researchers created basic middleware elements such as remote procedure call, file service, and directory service based on dramatic advances in hardware technology and fast networking and workstation systems. Today, the scope of middleware is broader, and distributed system technology occupies a prominent place in industrial and academic research and development.

The term *middleware* refers to the software layer between the operating system—including the basic communication protocols—and the distributed applications that interact via the network. This software infrastructure facilitates the interaction among distributed software modules. I avoid defining middleware further because its definitions share a common problem: Depending on the application environment, opinions differ as to which components comprise middleware. General middleware systems support the interaction of arbitrary application programs; specific functions such as remote database access, groupware support, and workflow systems require special middleware solutions.

A middleware layer seeks primarily to hide the underlying networked environment's complexity by insulating applications from explicit protocol handling, disjoint memories, data replication, network faults, and parallelism. Further, middleware masks the heterogeneity of computer architectures, operating systems, programming languages, and networking technologies to facilitate application programming and management. Middleware design includes quality of service (QoS) management and information security. Different middleware systems address these issues in different ways. The "Two Decades of Middleware Development" sidebar gives an overview of middleware history.

New application requirements challenge the established middleware design principles. As the first phase of middleware evolution draws to a close, we are poised to enter a major middleware design and development phase that requires new insights into distributed system technology.

THE CHANGING ENVIRONMENT

The communication and computing world has changed dramatically since the era in which early middleware developers worked in environments dominated by locally connected Unix workstations.

Enterprise application integration

Today, large enterprises with potentially autonomous suborganizations face a pressing requirement—integrating a multitude of applications and data sources

0018-9162/01/$10.00 © 2001 IEEE

within an enterprise and across enterprises. Formerly independent applications must interact to access and share functions and data stored in heterogeneous databases. In a large bank, hundreds of application subsystems must be integrated. These applications access data sources ranging from relational databases to external information providers. Further, collaborations with other enterprises, mergers, acquisitions, and the Internet—a totally unstructured data source—complicate the integration task. Such an environment's dominating attributes—large-scale configuration, diverse interaction models, autonomous interacting partners, and heterogeneous data views—may render previously appropriate middleware principles unsuitable.

Consider a travel reservation system with airline, hotel, and rental car services. With remote procedure call middleware, a reservation request for a trip leads to a chain of consecutive RPCs (synchronous, tightly coupled procedure invocations), for example, client → airline → hotel → rental car, and back to the client along the same sequence in reversed order. In RPC middleware, invocation calls a procedure; in object-oriented programming, invocation calls an object's method or a subroutine. With message-passing middleware, the interaction pattern is more flexible. The services notify the client directly about the reservation's progress. Generally, as the number of independent service providers increases, a chain of consecutive RPCs becomes too rigid. This requires a loosely coupled interaction model that adequately reflects the autonomy of the involved parties and provides the necessary spatial and temporal decoupling. The autonomy and decoupling aspects' importance increases with the size of the distributed system—an essential issue in an open global-service market.

Internet applications

The Internet's enormous success and growth have created a distributed application environment that differs from typical enterprise application scenarios. Web-based applications must cope with a variety of performance issues.

- The number of users may fluctuate and be unpredictable. If users can't get short response times, they tend to abort requests.
- The concept of a stateful user session is harder to maintain. Keeping state information at the server for user sessions and user profiles may be infeasible given the potentially large numbers of concurrent accesses.
- The interacting parties belong to independent, autonomous organizations that do not necessarily trust each other. Further, interaction takes place over an insecure medium.
- The communication infrastructure does not provide QoS guarantees.
- Because of the inherently open environment, exchanging information can require self-describing data and agreement on common ontologies.
- New Web-based applications must interoperate seamlessly with existing legacy applications.

For existing middleware systems, this nonexhaustive list of properties poses new demands. Developers

Two Decades of Middleware Development

Early examples of distributed system platforms include Athena at MIT,[1] ITC/ Andrew at CMU,[2] ANSAware,[3] and DACNOS at IBM and the University of Karlsruhe.[4] The DACNOS programming model, developed in the second half of the 1980s, was built on the elegant combination of an asynchronous communication model and a simple shared object model.[4] Thereafter, the industry consortia's standardization activities resulted in specifications for standard middleware architectures, such as DCE[5] and Corba. The International Organization for Standardization and its related standardization bodies provided a Reference Model for Open Distributed Processing that failed to make a significant impact in the middleware evolution.

In today's information technology environments, OMG Corba and Microsoft's Distributed Component Object Model/ COM+[6] provide widely used, general-purpose middleware architectures for distributed object computing. Lately, middleware approaches such as remote method invocation, Jini, JavaSpaces, and Enterprise JavaBeans that depend on Java's homogeneous programming language environment have received widespread attention. Other notable middleware products include message-based systems such as IBM's MQS.

References

1. E. Balkovich, S. Lerman, and R. Parmelee, "Computing in Higher Education: The Athena Experience," *CACM*, vol. 28, no. 11, 1985, pp. 1214-1224.
2. J.H. Morris et al., "A Distributed Personal Computing Environment," *CACM*, vol. 29, no. 3, 1986, pp. 184-201.
3. A. Herbert, "Key Architectural Concepts," ANSA Project, report no. AO-82-02, 1987.
4. K. Geihs and U. Hollberg, "A Retroperspective on DACNOS," *CACM*, vol. 33, no. 4, 1990, pp. 439-448.
5. W. Rosenberry, D. Kenney, and G. Fisher, *Understanding DCE*, O'Reilly & Associates, Sebastopol, Calif., 1992.
6. D.S. Platt, *Understanding COM+*, Microsoft Press, Redmond, Wash., July 1999.

QoS management aims to control attributes such as response time, availability, data accuracy, consistency, and security level.

must reevaluate architectures such as Corba and the Distributed Component Object Model from the Internet's perspective. Internet middleware developers must address issues such as autonomy, decentralized authority, intermittent connectivity, continual evolution, and scalability.

Quality of service

Increasing concerns about service quality have led to several proposals that advocate integrating QoS management into networking and distribution infrastructures. QoS management at the middleware and application levels aims to control attributes such as response time, availability, data accuracy, consistency, and security level. From the Internet perspective, QoS concerns seem to arise automatically when commercial applications meet a best-effort communication environment. When clients must pay for a service, they are certainly concerned about QoS, and they will expect to pay less for lower quality.

For Corba middleware, the Object Management Group recently proposed new messaging extensions that support QoS guarantees such as message delivery and error handling. Research projects have produced specific Corba-based platforms for handling individual QoS categories such as real time[1] or replication and fault tolerance.[2] Others have addressed generic frameworks for generating and operating customized systems for various QoS categories.[3,4] Although integrating QoS management into middleware architectures is essential, a procedure for doing so has yet to be agreed upon.

Nomadic mobility

Nomadic users in a highly mobile society enthusiastically take their "computing environment" everywhere—for business or private use.[5] New wireless communication technologies provide connectivity for laptop computers and personal digital assistants, phone organizers offer the processing power, and application-level protocols such as the wireless application protocol—a tiny first step—allow convenient applications to run on these devices. Undoubtedly, these developments point toward the widely expressed goal of accessing and processing information almost "anywhere and any time." Independent of our current location, we can already use voice communication via mobile phones almost anywhere, and we have worldwide access to personal e-mail, bank accounts, and home-country news over the Web.

Mobility introduces a key technical challenge because the available resources vary widely and unpredictably. Communication bandwidth and error rates change dynamically in wireless communication networks, a mobile system's battery power decreases, portable devices can be temporarily switched off or unreachable because of network partitions, and the monetary cost of communication can vary significantly. Envisaging a middleware system that makes these dynamics transparent is difficult; we anticipate that middleware must support the applications to explicitly accommodate these changes. Another mobile-computing issue, location awareness, demands that mobile-computer applications know their operating environment for context-dependent activities, such as giving directions or employing more or less stringent security mechanisms.

Ubiquitous computing

The ubiquitous or pervasive computing vision assumes that future computing environments will comprise diverse computing devices ranging from large computers to microscopic, invisible processing units contained in objects we use in activities of daily life.[6] We may, for example, wear "personal area networks" powered through motion energy. These computing devices will communicate with one another over a wireless network. Mobility and dynamic reconfiguration will be inherent features in this environment. Devices will automatically detect other devices, forming ad hoc agglomerations spontaneously. The middleware must withstand these dynamic challenges.

Addressing and naming the multitude of computing devices cannot be done with current technology. Even if we assume that the new IPv6 provides a sufficiently large IP address space, we must question whether every supermarket product's smart label, for example, should obtain its own IP address because losing such an address with the product's sale would be wasteful.

Although a low-end computing device possesses limited resources, it has stringent security requirements. Imagine remotely monitoring and controlling a heart monitor. Interrupted communication and security context changes at runtime make the security problem a critical issue.

PROGRAMMING MODELS

Since the dawn of the computer age, computer scientists have sought to determine the appropriate programming abstractions, particularly for distributed processing and middleware.

Client-server

The client-server model has been the predominant abstraction for building distributed software systems. The client, which binds to a server, initiates the interaction, sends a request, and awaits the answer. In principle, this is a sequential *pull* model with a single

logical control thread. The server stores long-term state information related to particular client-server associations. The term *client-server* is generally synonymous with distributed processing. However, increasingly important new application scenarios fit poorly with the client-server interaction model, denoting the end of this model's dominance.

Information dissemination uses a *push* model in which the information source sends information to a group of receivers who have registered their interest in a particular subject. Such publish/subscribe systems do not request information explicitly. Most workflow systems are not strictly client-server—rather, a task moves from station to station, and each station performs activities on a joint task. By receiving and forwarding tasks, each station participates simultaneously as client and server. The Web provides another example of storing long-term state information in the client's file system—in this case as cookies. Scalability problems prohibit keeping the state entirely at the server. Dealing with state in this way requires a particular programming style, which is different from conventional client-server programming.

Lately, peer-to-peer interaction models in the Internet have attracted a lot of attention. Their serverless file sharing effectively makes every computer client and server at the same time.

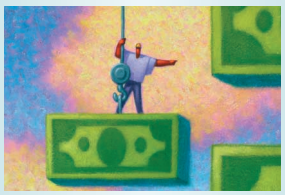
Thus, using the client-server model is not only inappropriate in many interaction scenarios, it also can be misleading to software developers and management. New application scenarios require another model and more adequate terminology.

Asynchronous interaction

Independent from any particular communication style, distributed programming models such as RPC and the later remote object invocation (ROI) are natural companions for client-server applications. These programming models introduce a synchronous, blocking interaction style in which a server object remains passive until it receives a request, and the system blocks the client's execution until the server response arrives. Distributed programming models hide distribution because the transaction looks like a local procedure call, and they elegantly handle the implicit synchronization. RPC and ROI remain middleware's most popular communication models.

Obvious drawbacks occur if the client uses the network environment's inherent parallelism, for example, to send a search request in parallel to several directory services. RPC-style communications offer two choices: Either use multithreading and spawn a separate thread per request or use a modified nonblocking RPC facility. The RPC system's inherently sequential interaction style has received some criticism.[7] Only lately has the need for scalability, flexibility, and decoupling in large-enterprise and Internet applications caused a strong general trend toward asynchronous, message-based communication in middleware systems.

Corba's latest release offers more asynchronous invocation mechanisms than previously available.[8] In addition to the existing deferred invocation and one-way operations, Corba messaging defines an inherently asynchronous interaction style and includes selectable QoS guarantees such as message priorities, request time limits, and queuing strategies. Further, two programming models—polling and callback—ensure that the client program can deal appropriately with asynchronous responses.



Large-scale, widely distributed systems require decoupled, asynchronous interaction.

For Internet applications, the simple object access protocol defines a mechanism for transporting invocations between peers using HTTP or other protocols and XML as the interface description and encoding language. SOAP does not prescribe any particular programming model. SOAP implements patterns such as request-response pairs as one-way transmissions from a sender to a receiver. Developers designed SOAP to correspond with the Internet's need for a lightweight, open, and flexible mechanism for linking arbitrary applications and services.

Event-based middleware architectures address the requirement for decoupled, asynchronous interaction in large-scale, widely distributed systems. Using events as the primary means of interaction allows asynchronous, peer-to-peer notifications between objects and provides flexible pattern-based event filtering and forwarding options.[9]

Message passing accommodates peer-to-peer interaction because it has weaker coupling and better scalability. However, in terms of programming abstractions, this low-level paradigm makes programming potentially more error-prone and more difficult to test and debug for elaborate communication patterns. Thus, we can view message passing as a backward step in middleware evolution that illustrates the design trade-off between degree of abstraction and practical requirements.

Shared memory

A continuous and lively debate focuses on invocation-oriented versus shared-memory cooperation models for distributed processing. The interaction among the distributed shared-memory model's remote processes occurs primarily by accessing shared information items such as shared objects or shared information spaces. The middleware provides the illusion of a shared memory.

Large-scale heterogeneous systems with many autonomous entities and parallel activities require new programming models.

For example, developers have widely cited the Linda tuple space approach[10] as an elegant, flexible base for distributed applications. Although not widely used commercially, Linda offers an option when distribution is an issue. Recently, JavaSpaces revived the Linda idea. A JavaSpace is a shared space containing Java objects that supports an associative, Linda-style object matching.

Other projects like PerDiS[11] have shown that the shared-memory paradigm can be attractive for data-intensive cooperative applications. However, in its pure form, it lacks responsiveness. Because objects are passive, asynchronous events require additional event-notification mechanisms.

The distributed shared-memory paradigm raises interesting middleware research questions in mobile-computing environments in which temporary disconnections occur. For example, to control access to replicate shared data, conventional pessimistic locking and concurrency control mechanisms restrict mobile systems too tightly. Depending on the application scenario, data usage pattern, and networking situation, the middleware should adopt optimistic concurrency control schemes to provide higher-degree data availability and application flexibility. An optimistic approach accepts updates on replicated shared objects provisionally and stores them in an update log. When the disconnected system reconnects, reconciliation takes place based on the update logs.[12]

Mobile code and mobile agents

Mobile code and mobile agents enhance the flexibility and adaptability of distributed applications at runtime. They also provide performance advantages in situations in which shipping small amounts of code to the nodes where the data originates is advisable instead of wasting transmission bandwidth for transferring large data quantities with low information content. Sending code rather than an invocation introduces a novel programming paradigm that is more general than the conventional request-response or distributed shared-memory models. The interaction of autonomous agents cannot be classified generally as client-server or publish/subscribe. Agents are peer entities that interact at their own will. Mobile agents not only need a new programming model but also require new typing concepts[13,14] and security provisions.[15]

Mobile agent-based middleware suffers from the obvious shortcoming of requiring a homogeneous programming language environment, which creates a rather strong assumption in an inherently heterogeneous network environment. This trade-off between generality in the programming model and homogeneity in the programming language requires further exploration.

The inherent diversity of interaction styles in large-scale heterogeneous systems with many autonomous entities and parallel activities requires new programming models. Whether we will live with a multitude of different models or develop a unified middleware programming model that supports decoupled, flexible, and scalable interaction remains to be seen.

ARCHITECTURE

In light of new technological advances, established middleware architecture elements merit reconsideration.

Distribution transparency

Creating transparency to hide the complexity and isolate applications from the underlying hardware and software details forms a cornerstone of all system software, especially for middleware systems. Consequently, the definition and discussion of distribution transparencies played a major role in the International Organization for Standardization's Reference Model for Open Distributed Processing. Distribution transparency is beneficial and necessary for programming distributed applications. However, distribution transparency cannot be the foremost goal in nomadic computing and context-aware applications.[16] For example, mobile users want to know about the security guarantees their current environment provides.

Context-aware applications need selective transparency features. The open research questions involve how to expose network imperfections at the right level of granularity and abstraction and how applications on top of the middleware deal with a selectable degree of transparency.

Increasing awareness of QoS requires making certain effects of distribution explicit. For example, customers who are charged for a certain level of communication service want to know about bandwidth variations or bad transmission quality because they expect to pay less for lower quality. Complete distribution transparency is inappropriate when computing applications must adapt to fluctuations in resource supply such as variations in communication bandwidth and fading battery power. However, we do not currently have middleware facilities to control the degree of transparency.

Layering

Since the implementation of the Open Systems Interconnection ISO 7498 standard, layering has served as the structuring principle for communication protocols. OSI's seven-layered architecture supports separation of concerns, modularity, extensibility, flexibility,

and so forth. Although later proposals did not always agree with some OSI layers—the need for a separate session layer is not obvious in Internet applications, for example—new application requirements make it necessary to renounce the strict layering principle in middleware systems and support a direct interaction between nonadjacent layers. This development corresponds with the need for selective distribution transparencies. Several issues are of concern:

- Context-aware applications may need the IP address or the geographical location to make location-dependent decisions.
- The application may need authentication information, such as an encryption key, in the secure sockets layer protocol's transport layer for access-control decisions, whereas the middleware itself is not interested in this information.
- An application may require a customized transport protocol to perform a multimedia transfer that directly influences a nonadjacent layer's configuration.

Middleware could offer appropriate programming interface elements to the applications and pass information to and from the lower layers. Why should the middleware bother with handling information of no concern to itself just because of the layering principle? The conventional layering principle can also be violated in the other direction, bottom-up: The middleware may need runtime information via a call back to the applications to request handling decisions that relate, for example, to the caller's security credentials.

Monolithic architectures

To date, middleware products have been monolithic software systems that do not support customization and adaptation. Such products cannot satisfy the needs of future computing environments. Mobile ubiquitous computing devices have few resources compared with their stationary counterparts. Thus, current middleware platforms such as Corba or COM+ are too bloated to be loaded into a resource-scarce mobile device. A Corba platform must be stripped down to fit its basic client functionality into a PalmPilot PDA with 2-MByte memory. Developers have criticized the inefficiency of existing object middleware in application scenarios involving conventional desktop computers. Occasionally, customized, low-overhead-interaction support mechanisms have replaced this middleware.[17]

Likewise, QoS management demands customizable middleware architectures. Diverse application requirements make it impossible to construct a ready-made platform for all QoS needs. MAQS[3] and QuO[4] are examples of how QoS frameworks can help build customized platforms using enhanced interface specifications and additional QoS management services. Although developers recognize that frameworks are the main architectural technique for supporting flexibility and reusability, few in the middleware arena use them. Moreover, frameworks typically exploit software design patterns. We need more research to find the specific patterns that middleware frameworks require and to explore the adaptability and flexibility limits of reflexive middleware architectures.[18]

Finally, we must understand how to introduce composability and customization into middleware systems in response to QoS and ubiquitous computing demands. Rather than relinquish our established architectural design principles, we must accept the challenge of amplifying these principles to accommodate new requirements.

Future computing environments will require software systems that support customization and adaptation.

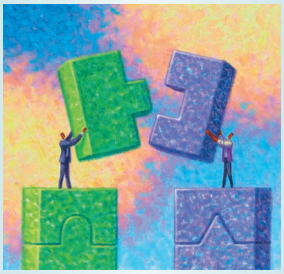
DYNAMIC CONFIGURATION

Dynamic changes in system configuration and operating context at runtime will be inherent characteristics of future computing environments. Middleware design and development must readily meet these challenges.

Disconnected operation

Mobile computing invalidates some implicit assumptions in current middleware platforms. For example, the Corba interoperability protocol assumes a permanent transport connection between the client and the object implementation. In contrast, PDAs automatically switch themselves off to save battery power. This requires buffering replies to client requests on the server side if the client is unreachable. How long should the server wait and keep the interaction's state? Can we view these situations as faults and handle them by applying conventional fault-tolerance mechanisms? Reconnection also poses a severe security problem, requiring mutual reauthentication within a client-server association's lifetime.

On the client side, receiving and reacting correctly to incoming replies requires facilities that uniquely identify and restore the state of earlier server associations. Web applications encode such state information in the URL or store it in cookies at the client's end, but Corba middleware, for example, currently cannot handle this operation in a systematic, architected way. Traditionally, we assumed that application entities would be permanently available during some application association. In modern networking



The middleware must monitor the resource supply and demand, compute adaptation decisions, and notify applications if they require adaptation.

environments, this assumption is no longer valid, although message-based, store-and-forward communication mechanisms offer one solution. The upcoming minimum Corba specification, part of Corba 3.0, will include enhanced messaging facilities, although whether this provides a viable practical solution remains to be seen.

Adaptive applications

Mobile-computing applications must cope with a dynamically varying resource supply. The underlying middleware cannot completely mask these fluctuations. A similar situation arises for QoS-aware applications in a best-effort networking environment such as the Internet. In the absence of resource guarantees, applications must adapt to the prevailing operating conditions. For example, if communication bandwidth becomes scarce, a tourist information application on a mobile computer with a wireless connection can display text and low-resolution pictures instead of video clips. Thus, the middleware must monitor the resource supply and demand, compute adaptation decisions, and notify applications if they require adaptation.

Although researchers have studied the viability of application adaptation in mobile systems, strategies for making adaptation decisions also require exploration. Currently, work is under way on a model that captures the essence of these decisions and allows comparison of different strategy performances.

Ad hoc organization

Ubiquitous computing requires enormously diverse computing devices, many of which are mobile and use wireless communications. Based on lower-layer discovery protocols, these devices automatically detect others and spontaneously form ad hoc agglomerations. Existing middleware platforms do not scale to the device diversity, population size, and runtime dynamics that ubiquitous computing requires. Static configuration assumptions about what infrastructure services are accessible in a computing environment are no longer sufficient. Self-organization, extensive information caching, and delegation of activities are important requirements.

The Java-based Jini system appears to anticipate these requirements. Jini defines a middleware infrastructure for spontaneous networking in which Java objects can discover, join, and interact with communities of objects. Whether the Jini approach will scale to the requirements of ubiquitous computing, however, remains unclear.

Intermediaries

The diversity and heterogeneity of distributed systems increase the need for intermediaries. For example, a low-end device may be incapable of hosting the complete middleware software in a ubiquitous computing environment. Therefore, a low-end device requires support from an intermediary on a more powerful computer. Then the intermediary translates and forwards external communication requests to the low-end device and manages agglomerations of low-end devices. Connecting heterogeneous middleware domains—for example, bridging COM+ to a message-queuing system—or separating authoritative domains—for example, to protect a corporate network with a firewall—also requires an intermediary. Developers also use intermediaries to transform ordinary information streams to enhance the information's quality.[19]

Developers have provided pragmatic solutions for various intermediary functions, but comprehensive general principles are missing. We must address not only intermediaries' functionality, but also the integration of issues like security, transactions, conversion overhead, and reliability. For example, using an intermediary to act as a representative for other devices requires sound security concepts for delegating authority. Although a few individual solutions are available such as the largely unexplored Corba security specification for delegation, how mature these concepts are remains unclear.

Middleware research and development has reached the end of its first major phase, and new requirements are arising that are so fundamentally different that they will lead to new-generation middleware systems. This transition poses a number of questions:

- What is the most appropriate programming model for the diverse application scenarios?
- Does a single distributed programming model fit all applications?
- Can we build customizable, configurable, and flexible middleware frameworks for inherently heterogeneous environments?
- What middleware features and infrastructure services will the dynamics and ad hoc nature of mobile-ubiquitous computing require?

These issues are the top challenges for future middleware research, generating open research problems that require building applications atop new middleware prototypes. Therefore, we have no reason to resign ourselves to believing Pike's provocative statement that "systems software research is irrelevant."[20] Many exciting and challenging questions await resolution. ✷

Acknowledgments

I thank Anne-Marie Kermarrec, Ant Rowstron, and Marc Shapiro for their technical contributions and constructive comments.

References

1. D.C. Schmidt, D.L. Levine, and S. Mungee, "The Design of the TAO Real-Time Object Request Broker," *Computer Comm. J.*, vol. 21, no. 4, 1998, pp. 294-324.
2. S. Maffeis, "The Object Group Design Pattern," *Proc. Conf. Object-Oriented Technologies and Systems* (COOTS 96), Usenix, Berkeley, Calif., 1996, pp. 294-303.
3. C. Becker and K. Geihs, "Generic QoS Support for Corba," *Proc. Int'l Symp. Computers and Communication* (ISCC 2000), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 60-65.
4. R. Vanegas et al., "QuO's Runtime Support for Quality of Service in Distributed Objects," *Proc. IFIP Int'l Conf. Distributed System Platforms and Open Distributed Processing*, Springer-Verlag, New York, 1998, pp. 207-223.
5. L. Kleinrock, "Nomadic Computing," *Proc. IFIP/ICCC Int'l Conf. Information Network and Data Communication*, Chapman & Hall, London, 1996, pp. 223-233.
6. M. Weiser, "Some Computer Science Issues in Ubiquitous Computing," *CACM*, July 1993, pp. 75-84.
7. A.S. Tanenbaum and R. Van Renesse, "A Critique of the Remote Procedure Call Paradigm," *Proc. European Teleinformatics Conf.* (EUTECO 88), North-Holland, Amsterdam, 1988, pp. 775-783.
8. J. Siegel, "An Overview of Corba 3," *Proc. 2nd IFIP Int'l Working Conf. Distributed Applications and Interoperable Systems* (DAIS 99), Kluwer, Boston, 1999, pp. 119-132.
9. J. Bacon et al., "Generic Support for Distributed Applications," *Computer*, Mar. 2000, pp. 68-76.
10. N. Carriero and D. Gelernter, "Linda in Context," *CACM*, Apr. 1989, pp. 444-458.
11. P. Ferreira et al., "PerDiS: Design, Implementation, and Use of a Persistent Distributed Store," *Recent Advances in Distributed Systems*, Springer-Verlag, New York, 2000, pp. 427-452.
12. M. Shapiro, A. Rowstron, and A-M. Kermarrec, "Application-Independent Reconciliation for Nomadic Applications," *Proc. 9th ACM SIGOPS European Workshop*, ACM Press, New York, 2000, pp. 1-6.
13. N. Minar et al., "Hive: Distributed Agents for Networking Things," *Proc. 1st Int'l Symp. Agent Systems and Applications and 3rd Int'l Symp. Mobile Agents*, IEEE CS Press, Los Alamitos, Calif., 1999, pp. 118-129.
14. M. Zapf and K. Geihs, "What Type Is It? A Type System for Mobile Agents," *Proc. 15th European Meeting on Cybernetics and Systems Research* (EMCSR 2000), Austrian Soc. for Cybernetic Studies, Vienna, 2000, pp. 585-590.
15. M. Zapf, H. Müller, and K. Geihs, "Security Requirements for Mobile Agents in Electronic Markets," *Proc. Int'l Working Conf. Trends in Distributed Systems for Electronic Commerce* (TrEC 98), Lecture Notes in Computer Science, Springer-Verlag, New York, 1998, pp. 205-214.
16. P.J. Brown, J.D. Bovey, and X. Chen, "Context-Aware Applications: From the Laboratory to the Marketplace," *IEEE Personal Comm.*, vol. 4, no. 5, 1997, pp. 58-64.
17. T. Weis and K. Geihs, "Components on the Desktop," *Proc. Technology of Object-Oriented Languages and Systems* (TOOLS Europe 2000), IEEE CS Press, Los Alamitos, Calif., 2000, pp. 250-261.
18. F. Costa, G. Blair, and G. Coulson, "Experiments with Reflective Middleware," *Proc. ECOOP 98 Workshop Reflective Object-Oriented Programming and Systems*, Springer-Verlag, New York, 1998, pp. 390-393.
19. R. Barrett and P. Maglio, "Intermediaries: An Approach to Manipulating Information Streams," *IBM Systems J.*, vol. 38, no. 4, 1999, pp. 629-641.
20. R. Pike, "Systems Software Research Is Irrelevant," http://cm.bell-labs.com/who/rob/utah2000.ps.

Kurt Geihs *is a professor of computer science at Goethe University, Frankfurt, Germany. His research interests include distributed systems, operating systems, networks, and software technology. Current projects focus on quality-of-service management in Corba, component-based software, and mobile agents. Geihs received a PhD in computer science from the Technical University in Aachen, Germany. Contact him at geihs@informatik.uni-frankfurt.de.*