

Software Engineering and Middleware: A Roadmap

Wolfgang Emmerich

Dept. of Computer Science

University College London

London WC1E 6BT, UK

w.emmerich@cs.ucl.ac.uk

ABSTRACT

The construction of a large class of distributed systems can be simplified by leveraging *middleware*, which is layered between network operating systems and application components. Middleware resolves heterogeneity, and facilitates communication and coordination of distributed components. Existing middleware products enable software engineers to build systems that are distributed across a local-area network. State-of-the-art middleware research aims to push this boundary towards Internet-scale distribution, adaptive and reconfigurable middleware and middleware for dependable and wireless systems. The challenge for software engineering research is to devise notations, techniques, methods and tools for distributed system construction that systematically build and exploit the capabilities that middleware deliver.

1 INTRODUCTION

Various commercial trends have led to an increasing demand for distributed systems. Firstly, the number of mergers between companies is continuing to increase. The different divisions of a newly merged company have to deliver unified services to their customers and this usually demands an integration of their IT systems. The time available for delivery of such an integration is often so short that building a new system is not an option and therefore existing system components have to be integrated into a distributed system that appears as an integrating computing facility. Secondly, the time available for providing new services are decreasing. Often this can only be achieved if components are procured off-the-shelf and then integrated into a system rather than built from scratch. Components to be integrated may have incompatible requirements for their hardware and operating system platforms; they have to be deployed on different hosts, forcing the resulting system to be distributed. Finally, the Internet provides new opportunities to offer products and services to a vast number of potential customers. In this setting, it is difficult to estimate the scalability requirements.

An e-commerce site that was designed to cope with a given number of transactions per day may suddenly find itself exposed to demand that is by orders of magnitude larger. The required scalability cannot usually be achieved by centralized or client-server architectures but demands a distributed system.

Distributed systems can integrate legacy components, thus preserving investment, they can decrease the time to market, they can be scalable and tolerant against failures. The caveat, however, is that the construction of a truly distributed systems is considerably more difficult than building a centralized or client/server system. This is because there are multiple points of failure in a distributed system, system components need to communicate with each other through a network, which complicates communication and opens the door for security attacks. Middleware has been devised in order to conceal these difficulties from application engineers as much as possible; As they solve a real problem and simplify distributed system construction, middleware products are rapidly being adopted in industry [6].

In order to build distributed systems that meet the requirements, software engineers have to know what middleware is available, which one is best suited to the problems at hand, and how middleware can be used in the architecture, design and implementation of distributed systems.

The principal contribution of this paper is an assessment of both, the state-of-the-practice that current middleware products offer and the state-of-the-art in middleware research. Software engineers increasingly use middleware to build distributed systems. Any research into distributed software engineering that ignores this trend will only have limited impact. We, therefore, analyze the influence that the increasing use of middleware should have on the software engineering research agenda. We argue that requirements engineering techniques are needed that focus on non-functional requirements, as these influence the selection and use of middleware. We identify that software architecture research should produce methods that guide engineers towards selecting the right middleware and employing it so that it meets a set of non-functional requirements. We then highlight that the use of middleware is not transparent for system design and that design methods are needed that address this issue.

This paper is further structured as follows. In Section 2, we discuss some of the difficulties involved in building distributed systems and delineate requirements for middleware. In Section 3, we use these requirements to attempt an assessment of the support that current middleware products provide for distributed system construction. We then present an overview of ongoing middleware research in Section 4 in order to provide a preview of what future middleware products might be capable of. In Section 5, we delineate a research agenda for distributed software engineering that builds on the capabilities of current and future middleware and conclude the paper in Section 6.

2 MIDDLEWARE REQUIREMENTS

In this section, we review the difficulties that arise during distributed system construction. We argue in this section that it is too expensive and time consuming if application designers have to resolve these problems by directly using network operating system primitives. Instead they require a middleware that provides higher-level primitives. This approach to distributed system construction with middleware is sketched in Figure 1.

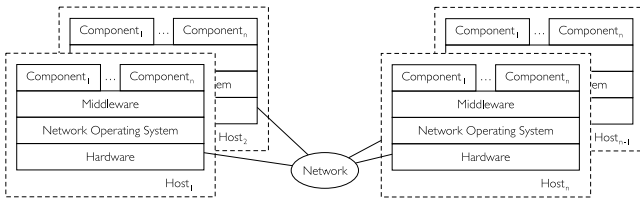


Figure 1: Middleware in Distributed System Construction

Thus middleware is layered between network operating systems and application components [13]. Middleware facilitates the communication and coordination of components that are distributed across several networked hosts. The aim of middleware is to provide application engineers with high-level primitives that simplify distributed system construction. The idea of using middleware to build distributed systems is comparable to that of using database management systems when building information systems. It enables application engineers to abstract from the implementation of low-level details, such as concurrency control, transaction management and network communication, and allows them to focus on application requirements.

Network Communication

As shown in Figure 1, the different components of a distributed system may reside on different hosts. In order for the distributed system to appear as an integrated computing facility, the components have to communicate with each other. This communication can only be achieved by using network protocols, which are often classified by the ISO/OSI reference model [25]. Distributed systems are usually built on top of the transport layer, of which TCP or UDP are good examples. The layers underneath are provided by the network operating system.

Different transport protocols have in common that they can transmit messages between different hosts. If the communication between distributed systems is programmed at this level of abstraction, application engineers need to implement session and presentation layer. This is too costly, too error-prone and too time-consuming. Instead, application engineers should be able to request parameterized services from possibly more than one remote components and may wish to execute them as atomic and isolated transactions, leaving the implementation of session and presentation layer to the middleware.

The parameters that a component requesting a service needs to pass to a component that provides a service are often complex data structures. The presentation layer implementation of the middleware should provide the ability to transform these complex data structures into a format that can be transmitted using a transport protocol, i.e. a sequence of bytes. This transformation is referred to as *marshalling* and the reverse is called *unmarshalling*.

Coordination

By virtue of the fact that components reside on different hosts, distributed systems have multiple points of control. Components on the same host execute concurrently, which leads to a need for synchronization when components communicate with each other. This synchronization needs to be implemented in the session layer implementation provided by the middleware.

Synchronization can be achieved in different ways. A component can be blocked while it waits for another component to complete execution of a requested service. This form of communication is often called *synchronous*. After issuing a request, a component can also continue to perform its operations and synchronize with the service providing component at a later point. This synchronization can then be initiated by either the client component (using, for example polling), in which case the interaction is often called *deferred synchronous*. Synchronization that is initiated by the server is referred to as *asynchronous* communication. Thus, application engineers need some basic mechanisms that support various forms synchronization between communicating components.

Sometimes more than two components are involved in a service request. These forms of communications are also referred to as *group requests*. This is often the case when more than one component is interested in events that occur in some other component. An example is a distributed stock ticker application where an event, such as a share price update, needs to be communicated to multiple distributed display components, to inform traders about the update. Although the basic mechanisms for this push-style communication are available in multi-cast networking protocols additional support is needed to achieve reliable delivery and marshalling of complex request parameters.

A slightly different coordination problem arises due to the sheer number of components that a distributed system may have. The components, i.e. modules or libraries, of a centralized application reside in virtual memory while the application is executing. This is inappropriate for distributed components for the following reasons:

- Hosts sometimes have to be shut down and components hosted on these machines have to be stopped and restarted when the host resumes operation;
- The resources required by all components on a host may be greater than the resources the host can provide; and
- Depending on the nature of the application, components may be idle for long periods, thus wasting resources if they were kept in virtual memory all the time.

For these reasons, distributed systems need to use a concept called *activation* that allows for component executing processes to be started (*activated*) and terminated (*deactivated*) independently from the applications that they execute. The middleware should manage persistent storage of components' state prior to deactivation and restore components' state during activation. Middleware should also enable application programmers to determine the *activation policies* that define when components are activated and de-activated.

Given that components execute concurrently on distributed hosts, a server component may be requested from different client components at the same time. The middleware should support different mechanisms called *threading policies* to control how the server component reacts to such concurrent requests. The server component may be single-threaded, queue requests and process them in the order of their arrival. Alternatively, the component may also spawn new threads and execute each request in its own thread. Finally the component may use a hybrid threading policy that uses a pool with a fixed number of threads to execute requests, but starts queueing once there are no free threads in the pool.

Reliability

Network protocols have varying degrees of reliability. Protocols that are used in practice do not necessarily guarantee that every packet that a sender transmits is actually received by the receiver and that the order in which they are sent is preserved. Thus, distributed system implementations have to put error detection and correction mechanisms in place to cope with these unreliabilities.

Unfortunately, reliable delivery of service requests and service results does not come for free. Reliability has to be paid for with decreases in performance. To allow engineers to trade-off reliability and performance in a flexible manner, different degrees of service request reliability are needed in practice.

For communication about service requests between two components, the reliabilities that have been suggested in the distributed system literature are *best effort*, *at-most-once*, *at-*

least-once and *exactly-once* [13]. Best effort service requests do not give any assurance about the execution of the request. At-most-once requests are guaranteed to execute only once. It may happen that they are not executed, but then the requester is notified about the failure. At-least-once service requests are guaranteed to be executed, possibly more than once. The highest degree of reliability is provided by exactly-once requests, which are guaranteed to be executed once and only once.

Additional reliabilities can be defined for group requests. In particular, the literature mentions *k-reliability*, *time-outs*, and *totally-ordered* requests. K-reliability denotes that at least K components receive the communication. Time-outs allow the specification of periods after which no delivery of the request should be attempted to any of the addressed components. Finally *totally-ordered* group communication denotes that a request never overtakes a request of a previous group communication.

The above reliability discussion applies to individual requests. We can extend it and consider more than one request. Transactions [18] are important primitives that are used in reliable distributed systems. Transactions have ACID properties, which means they enable multiple request to be executed in an *atomic*, *consistency-preserving*, *isolated* and *durable* manner. This means that the sequence of requests is either performed completely, or not at all. It enforces that every completed transaction is consistent. It demands that a transaction is isolated from concurrent transaction and, finally that once the transaction is completed its effect cannot be undone. Every middleware that is used in critical applications needs to support distributed transactions.

Reliability may also be increased by *replicating* components [4], i.e. components are available in multiple copies on different hosts. If one component is unavailable, for example because its host needs to be rebooted, a replica on a different host can take over and provide the requested service. Sometimes components have an internal state and then the middleware should support replication in such a way that these states are kept in sync.

Scalability

Scalability denotes the ability to accommodate a growing future load. In centralized or client/server systems, scalability is limited by the load that the server host can bear. This can be overcome by distributing the load across several hosts. The challenge of building a scalable distributed system is to support changes in the allocation of components to hosts without changing the architecture of the system or the design and code of any component. This can only be achieved by respecting the different dimensions of *transparency* identified in the ISO Open Distributed Processing (ODP) reference model [24] in the architecture and design of the system. *Access transparency*, for example demands that the way a component accesses the services of another component is in-

dependent of whether it is local or remote. Another example is *location transparency*, which demands that components do not know the physical location of the components they interact with. A detailed discussion of the different transparency dimension is beyond the scope of this paper and the reader is referred to [13].

If components can access services without knowing the physical location and without changing the way they request it, *load balancing* mechanisms can migrate components between machines in order to reduce the load on one host and increase it on another host. It should again be transparent to users whether or not such a migration occurred. This is referred to as *migration transparency*.

Replication can also be used for load balancing. Components whose services are in high demand may have to exist in multiple copies. *Replication transparency* means that it is transparent for the requesting components, whether they obtain a service from the master component itself or from a replica.

The different transparency criteria that will lead to scalable systems are very difficult to achieve if distributed systems are built directly on network operating system primitives. To overcome these difficulties, we demand from middleware that they support access, location, migration and replication transparency.

Heterogeneity

The components of distributed systems may be procured off-the-shelf, may include legacy and new components. As a result they are often rather heterogeneous. This heterogeneity comes in different dimensions: hardware and operating system platforms, programming languages and indeed the middleware itself.

Hardware platforms use different encodings for atomic data types, such as numbers and characters. Mainframes use the EBCDIC character set, Unix servers may use 7-bit ASCII characters, while Windows-based PCs use 16-bit Unicode character encodings. Thus the character encoding of alphanumeric data that is sent across different types of platforms has to be adjusted. Likewise, mainframes and RISC servers, for example, use big-endian representations for numbers, i.e. the most significant byte encoding an integer, long or floating point number comes last. PCs, however, use a little-endian representation where the significance of bytes decreases. Thus, whenever a number is sent from a little-endian host to a big-endian host or vice versa, the order of bytes with which this number is encoded needs to be swapped. This heterogeneity should be resolved by the middleware rather than the application engineer.

When integrating legacy components with newly-built components, it often occurs that different programming languages need to be used. These programming languages may follow different paradigms. While legacy components

tend to be written in imperative languages, such as COBOL, PL/I or C, newer components are often implemented using object-oriented programming languages. Even different object-oriented languages have considerable differences in their object model, type system, approach to inheritance and late binding. These differences need to be resolved by the middleware.

As we shall see in the next section, there is not just one, but many approaches to middleware. The availability of different middleware solutions may present a selection problem, but sometimes there is no optimal single middleware, and multiple middleware systems have to be combined. This may be for a variety of reasons. Different middleware may be required due to availability of programming language bindings, particular forms of middleware may be more appropriate for particular hardware platforms (e.g. COM on Windows and CORBA on Mainframes). Finally, the different middleware systems will have different performance characteristics and depending on the deployment a different middleware may have to be used as a backbone than the middleware that is used for local components. Thus middleware will have to be *interoperable* with other implementations of the same middleware or even different types of middleware in order to facilitate distributed system construction.

3 MIDDLEWARE SOLUTIONS

In this section, we review the state of current middleware products. We identify the extent to which they address the above requirements and highlight their shortcomings. As it is impossible to review individual middleware products in this paper, we first present a classification, which allows us to abstract from particular product characteristics and which provides a conceptual framework for comparing the different approaches.

The four categories that we consider are *transactional*, *message-oriented*, *procedural*, and *object or component* middleware. We have chosen this classification based on the primitives that middleware products provide for the interaction between distributed components, which are distributed transactions, message passing, remote procedure calls and remote object requests.

Transactional Middleware

Transactional middleware supports transactions involving components that run on distributed hosts. Transaction-oriented middleware uses the two-phase commit protocol [3] to implement distributed transactions. The products in this category include IBM's CICS [22], BEA's Tuxedo [19] and Transarc's Encina.

Network Communication: Transactional middleware enables application engineers to define the services that server components offer, implement those server components and then write client components that request several of those services within a transaction. Client and server components can reside on different hosts and therefore requests are transported

via the network in a way that is transparent to client and server components.

Coordination: The client components can request services using synchronous or asynchronous communication. Transactional middleware supports various activation policies and allows services to be activated on demand and deactivated when they have been idle for some time. Activation can also be permanent, allowing the server component to always reside in memory.

Reliability: A client component can cluster more than one service request into a transaction, even if the server components reside on different machines. In order to implement these transactions, transactional middleware has to assume that the participating servers implement the two-phase commit protocol. If server components are built using database management systems, they can delegate implementation of the two-phase commit to these database management systems. For this implementation to be portable, a standard has been defined. The Distributed Transaction Processing (DTP) Protocol, which has been adopted by the Open Group, defines a programmatic interface for two-phase commit in its XA-protocol [43]. DTP is widely supported by relational and object-oriented database management systems. This means that distributed components that have been built using any of these database management systems can easily participate in distributed transactions. This makes them fault-tolerant, as they automatically recover to the end of all completed transactions.

Scalability: Most transaction monitors support load balancing, and replication of server components. Replication of servers is often based on replication capabilities that the database management systems provide upon which the server components rely.

Heterogeneity: Transactional middleware supports heterogeneity because the components can reside on different hardware and operating system platforms. Also different database management systems can participate in transactions, due to the standardized DTP protocol. Resolution of data heterogeneity is, however, not well-supported by transactional middleware, as the middleware does not provide primitives to express complex data structures that could be used as service request parameters and therefore also does not marshal them.

The above discussion has shown that transactional middleware can simplify the construction of distributed systems. Transactional middleware, however, has several weaknesses. Firstly, it creates an undue overhead if there is no need to use transactions, or transactions with ACID semantics are inappropriate. This is the case, for example, when the client performs long-lived activities. Secondly, marshalling and unmarshalling between the data structures that a client uses and the parameters that services require needs to be done manually in many products. Thirdly, although the API for the

two-phase commit is standardized, there is no standardized approach for defining the services that server components offer. This reduces the portability of a distributed system between different transaction monitors.

Message-Oriented Middleware

Message-oriented middleware (MOM) supports the communication between distributed system components by facilitating message exchange. Products in this category include IBM's MQSeries [16] and Sun's Java Message Queue [20].

Network Communication: Client components use MOM to send a message to a server component across the network. The message can be a notification about an event, or a request for a service execution from a server component. The content of such a message includes the service parameters. The server responds to a client request with a reply-message containing the result of the service execution.

Coordination: A strength of MOM is that this paradigm supports asynchronous message delivery very naturally. The client continues processing as soon as the middleware has taken the message. Eventually the server will send a message including the result and the client will be able to collect that message at an appropriate time. This achieves de-coupling of client and server and leads to more scalable systems. The weakness, at the same time, is that the implementation of synchronous requests is cumbersome as the synchronization needs to be implemented manually in the client. A further strength of MOM is that it supports group communication by distributing the same message to multiple receivers in a transparent way.

Reliability: MOM achieves fault-tolerance by implementing message queues that store messages temporarily on persistent storage. The sender writes the message into the message queue and if the receiver is unavailable due to a failure, the message queue retains the message until the receiver is available again.

Scalability: MOMs do not support access transparency very well, because client components use message queues for communication with remote components, while it does not make sense to use queues for local communication. This lack of access transparency disables migration and replication transparency, which complicates scalability. Moreover, queues need to be set up by administrators and the use of queues is hard-coded in both client and server components, which leads to rather inflexible and poorly adaptable architectures.

Heterogeneity: MOM does not support data heterogeneity very well either, as the application engineers have to write the code that marshals. With most products, there are different programming language bindings available.

In assessing the strengths and weaknesses of MOM, we can note that this class of middleware is particularly well-suited for implementing distributed event notification and

publish/subscribe-based architectures. The persistence of message queues means that this event notification can be achieved in fault tolerant ways so that components receive events when they restart after a failure. However, message-oriented middleware also has some weaknesses. It only supports at-least once reliability. Thus the same message could be delivered more than once. Moreover, MOM does not support transaction properties, such as atomic delivery of messages to all or none receivers. There is only limited support for scalability and heterogeneity.

Procedural Middleware

Remote Procedure Calls (RPCs) were devised by Sun Microsystems in the early 1980s as part of the Open Network Computing (ONC) platform. Sun provided remote procedure calls as part of all their operating systems and submitted RPCs as a standard to the X/Open consortium, which adopted it as part of the Distributed Computing Environment (DCE) [36]. RPCs are now available on most Unix implementations and also on Microsoft's Windows operating systems.

Network Communication: RPCs support the definition of server components as RPC programs. An RPC program exports a number of parameterized procedures and associated parameter types. Clients that reside on other hosts can invoke those procedures across the network. Procedural middleware implements these procedure calls by marshalling the parameters into a message that is sent to the host where the server component is located. The server component unmarshalls the message and executes the procedure and transmits marshalled results back to the client, if required. Marshalling and unmarshalling are implemented in client and server stubs, that are automatically created by a compiler from an RPC program definition.

Coordination: RPCs are synchronous interactions between exactly one client and one server. Asynchronous and multicast communication is not supported directly by procedural middleware. Procedural middleware provides different forms of activating server components. Activation policies define whether a remote procedure program is always available or has to be started on demand. For startup on demand, the RPC server is started by an `inetd` daemon as soon as a request arrives. The `inetd` requires an additional configuration table that provides for a mapping between remote procedure program names and the location of programs in the file system.

Reliability: RPCs are executed with at-most once semantics. The procedural middleware returns an exception if an RPC fails. Exactly-once semantics or transactions are not supported by RPC programs.

Scalability: The scalability of RPCs is rather limited. Unix and Windows RPCs do not have any replication mechanisms that could be used to scale RPC programs. Thus replication has to be addressed by the designer of the RPC-based system,

which means in practice that RPC-based systems are only deployed on a limited scale.

Heterogeneity: Procedural middleware can be used with different programming languages. Moreover, it can be used across different hardware and operating system platforms. Procedural middleware standards define standardized data representations that are used as the transport representation of requests and results. DCE, for example standardizes the Network Data Representation (NDR) for this purpose. When marshalling RPC parameters, the stubs translate hardware-specific data representations into the standardized form and the reverse mapping is performed during unmarshalling.

Procedural middleware is weaker than transactional middleware and MOM as it is not as fault tolerant and scalable. Moreover, the coordination primitives that are available in procedural middleware are more restricted as they only support synchronous invocation directly. Procedural middleware improve transactional middleware and MOM with respect to interface definitions from which implementations that automatically marshal and unmarshal service parameters and results. A disadvantage of procedural middleware is that this interface definition is not reflexive. This means that procedures exported by one RPC program cannot return another RPC program. Object and component middleware resolve this problem.

Object and Component Middleware

Object middleware evolved from RPCs. The development of object middleware mirrored similar evolutions in programming languages where object-oriented programming languages, such as C++ evolved from procedural programming languages such as C. The idea here is to make object-oriented principles, such as object identification through references and inheritance, available for the development of distributed systems. Systems in this class of middleware include the Common Object Request Broker Architecture (CORBA) of the OMG [34, 37], the latest versions of Microsoft's Component Object (COM) [5] and the Remote Method Invocation (RMI) capabilities that have been available since Java 1.1 [28]. More recent products in this category include middleware that supports distributed components, such as Enterprise Java Beans [30]. Unfortunately, we can only discuss and compare this important class of middleware briefly and refer to [8, 13] for more details.

Network Communication: Object middleware support distributed object requests, which mean that a client object requests the execution of an operation from a server object that may reside on another host. The client object has to have an object reference to the server object. Marshalling operation parameters and results is again achieved by stubs that are generated from an interface definition.

Coordination: The default synchronization primitives in object middleware are synchronous requests, which block the client object until the server object has returned the response.

However, the other synchronization primitives are supported, too. CORBA 3.0, for example, supports both deferred synchronous and asynchronous object requests. Object middleware supports different activation policies. These include whether server objects are active all the time or started on-demand. Threading policies are available that determine whether new threads are started if more than one operation is requested by concurrent clients, or whether they are queued and executed sequentially. CORBA also supports group communication through its Event and Notification services. This service can be used to implement push-style architectures.

Reliability: The default reliability for object requests is at most once. Object middleware support exceptions, which clients catch in order to detect that a failure occurred during execution of the request. CORBA messaging, or the Notification service [33] can be used to achieve exactly-once reliability. Object middleware also supports the concept of transactions. CORBA has an Object Transaction service [32] that can be used to cluster requests from several distributed objects into transactions. COM is integrated with Microsoft's Transaction Server [21], and the Java Transaction Service [7] provides the same capability for RMI.

Scalability: The support of object middleware for building scalable applications is still somewhat limited. Some CORBA implementations support load-balancing, for example by employing using name servers that return an object reference for a server on the least loaded host, or using factories that create server objects on the least loaded host, but support for replication is still rather limited.

Heterogeneity: Object middleware supports heterogeneity in many different ways. CORBA and COM both have multiple programming language bindings so that client and server objects do not need to be written in the same programming language. They both have a standardized data representation that they use to resolve heterogeneity of data across platforms. Java/RMI takes a different approach as heterogeneity is already resolved by the Java Virtual Machine in which both client and server objects reside. The different forms of object middleware inter-operate. CORBA defines the Internet Inter-Orb Protocol (IIOP) standard [34], which governs how different CORBA implementations exchange request data. Java/RMI leverages this protocol and uses it as a transport protocol for remote method invocations, which means that a Java client can perform a remote method invocation of a CORBA server and vice versa. CORBA also specifies an inter-working specification to Microsoft's COM.

Object middleware provides very powerful component models. They integrate most of the capabilities of transactional, message-oriented or procedural middleware. However, the scalability of object middleware is still rather limited and this disables use of the distributed object paradigm on a large-scale.

4 MIDDLEWARE STATE-OF-THE-ART

While middleware products are already successfully employed in industrial practice, they still have several shortcomings, which prevent their use in many application domains. These weaknesses lead to relatively inflexible systems that do not respond well to changing requirements; they do not really scale beyond local area networks; they are not yet dependable and are not suited to use in wireless networks.

In this section, we review the state-of-the-art of middleware research that addresses the current weaknesses that will influence the next-generation of middleware products. We discuss trading, reflection and application-level transport mechanisms that support the construction of more flexible software architectures. We present replication techniques that will lead to better scalability and fault-tolerance. We then discuss research into middleware that supports real-time applications and finally address middleware research results for mobile and pervasive computing.

Flexible Middleware

Trading: Most middleware products use naming for component identification: MOMs use named message queues, DCE has a Directory service, CORBA has a Naming service, COM uses monikers and Java/RMI uses the RMIRegistry to bind names to components. Before a client component can make a request, it has to resolve a name binding in order to obtain a reference to the server component. This means that clients need to uniquely identify their servers, albeit in a location-transparent way. In many application domains, it is unreasonable to assume that client components can identify the component from which they can obtain a service. Even if they can, this leads to inflexible architectures where client components cannot dynamically adapt to better service providers becoming available.

Trading has been suggested as an alternative to naming and it offers more flexibility. The ISO/ODP standard defines the principal characteristics of trading [2]. The idea is similar to the yellow pages of the telephone directory. Instead of using names, components are located based on service types. The trader registers the type of service that a server component offers and the particular *qualities of service* (QoS) that it guarantees. Clients can then query the trader for server components that provide a particular service type and demand the QoS guarantees from them. The trader matches such a service query with the service offers that it knows about and returns a component reference to the client. From then on the client and the server communicate without involvement of the trader.

The idea of trading has matured and is starting to be adopted in middleware products. The OMG has defined a Trading service [32] that adapts the ODP trader ideas to the distributed object paradigm and first implementations of this service are becoming available. Thus trading enables the dynamic connection of clients with server components based

on the service characteristics rather than the server's name.

Reflection: Another approach to more flexible execution environments for components is *reflection*. Reflection is a well-known paradigm in programming languages [17]. Programs use reflection mechanisms to discover the types or classes and define method invocations at run-time. Reflection is already supported to some extent by current middleware products. The interface repository and dynamic invocation interface of CORBA enable client programmers to discover the types of server components that are currently known and then dynamically create requests that invoke operations from these components.

Current research into reflective middleware [9] goes beyond reflective object and component models. It aims to support meta object protocols [29]. These protocols are used for *inspection* and *adaptation* of the middleware execution environment itself. In [12] it is suggested, for example, to use an *environment meta-model*. Inspection of the environment meta-model supports queries of the middleware's behaviour upon events, such as message arrival, enqueueing of requests, marshalling and unmarshalling, thread creation and scheduling of requests. Adaptation of the environment meta-model enables components to adjust the behaviour of the middleware to any of those events.

Application-level Transport Protocols: While marshalling and unmarshalling is mostly best done by the middleware, there are applications, where the middleware creates an undue overhead. One important application of reflection is therefore to marshalling. This is particularly the case when there is an application-specific data representation that is amenable for transmission through a network that and heterogeneity does not need to be resolved by the middleware.

In [14] we investigate the combined use of middleware and markup-languages, such as XML [14]. We suggest to transmit XML documents as uninterpreted byte strings using middleware. This combination is motivated by the fact that XML supports semantic translations between data structures and by the fact that existing markup language definitions, such as FpML [15] or FIXML [23] can be leveraged. On the other hand, the HTTP protocol with which XML was originally used is clearly inappropriate to meet reliability requirements. It can be expected that interoperability between application-level and middleware data-structures will become available in due course, because the OMG have started an adoption process for technology that will provide seamless interoperability between CORBA data structures and XML structured documents [35].

Scalable Middleware

Although middleware is successfully used in scalable applications on local-area networks, current middleware standards and products impose limitations that prevent their use in globally distributed systems. In particular, current middleware platforms do not support replication to the neces-

sary extent to achieve global distribution [31]. State of the art research addresses this problem through *non-transparent replication*.

Replication: Tanenbaum is addressing this problem for distributed object middleware in the Globe project [42]. The aim of Globe is to provide an object based middleware that scales to a billion users. To achieve this aim, Globe makes extensive use of replication. Unlike other replication mechanisms, such as Isis [4], Globe does not assume the existence of a universally applicable replication strategy. It rather suggests that replication policies have to be object-type specific, and therefore they have to be determined by server object designers. Thus, Globe assumes that each type of object its own strategy that proactively replicates objects.

Real-time Middleware

A good summary of the state of the art in real-time middleware has been produced in the EU funded CaberNet network of excellence by [1].

Most current middleware products are only of limited use in real-time and embedded systems because all requests have the same priority. Moreover the memory requirements of current middleware products prevent deployment in embedded systems. These problems have been addressed by various research groups. TAO [39] is a real-time CORBA prototype developed that supports request prioritization and the definition of scheduling policies. The CORBA 3.0 specification [41] builds on this research and standardizes real-time and minimal middleware.

Middleware for Mobile Computing

Current middleware products assume continuous availability of high-bandwidth network connections. These cannot be achieved with physically mobile hosts for various reasons. Wireless local area network protocols, such as WaveLAN, do not achieve reasonable bandwidth. However, they only operate if hosts are within reach of a few hundred metres from their base station. Network outages occur if mobile hosts roam across areas covered by different base stations or if they enter 'radio shadows'. Wide-area wireless network protocols, such as GSM have similar problems during cell handovers. In addition, their bandwidth is by orders of magnitude smaller; GSM achieves at most 9,600 baud. State-of-the-art wireless and wide-area protocols, such as GSRM and UTMS will improve this situation. However, they will not be available for another two years.

Several problems occur when current middleware products are used with these wireless network protocols. Firstly, they all treat unreachability of server or client components as exceptional situation and raise errors that client or server component programmers have to deal with. Secondly, the transport representation that is chosen for wired networks with bandwidth beyond 100Mbit does not need to be size-efficient. Middleware products are therefore optimized to simplify both, the translation between different heteroge-

neous data representations, and the routing of messages to their intended receivers. Such optimizations do not need to choose size efficient encodings for the network protocol and are therefore inappropriate when packets are sent through a 9,600 baud wireless connection.

Research into middleware for mobile computing aims to overcome these issues by providing coordination primitives, such as tuple spaces, that treat unreachability as normal rather than exceptional situations. Moreover, they use compressed transport representation to save bandwidth. A good overview into the state of the art for mobile middleware is given by [38] and we therefore avoid to delve into detail in this paper.

5 MIDDLEWARE AND SOFTWARE ENGINEERING RESEARCH

In this section, we analyze the consequences of the availability of middleware products and their evolution as a result of middleware research on the software engineering research agenda. We argue on the importance of non-functional requirements for building software systems with existing and upcoming middleware and identify a need for requirements engineering techniques that focus on non-functional requirements. We identify that software architecture research should produce methods that systematically guide engineers towards selecting the right middleware and employing it in such a way that it meets a set of non-functional requirements. We then highlight that the use of middleware is not transparent for system design and that design methods are needed that address this issue.

Two trends are important for the discussion of the impact of middleware on software engineering research. Firstly, middleware products are conceived to deliver immediate benefits in the construction of distributed systems. They are therefore rapidly adopted in industry. Secondly, middleware vendors have a proven track record to incorporate middleware research results into their products. An example is the ISO/ODP Trader, which was defined in 1993, adopted as a CORBA standard in 1997 and last year became available in the first CORBA products. There is therefore a good chance that some of the state-of-the-art research in the areas of flexible, scalable, real-time and mobile middleware will become state of the practice in 3-5 years.

Unless research into software engineering for distributed systems delivers principles, notations, methods and tools that are compatible with the capabilities that current middleware products provide and that middleware research will generate in the future, software engineering research results will only be of limited industrial significance. Industry will adopt the middleware that is known to deliver the benefits and ignore incompatible software engineering methods and tools. Middleware products and research, however, only support programming and largely ignore all other activities that are needed in software processes for distributed systems. We,

therefore, have a chance to achieve a symbiosis between software engineering and middleware. The aim of this section is to identify the software engineering research themes that will lead to the principles, notations, methods and tools that are needed to support all life cycle activities when building distributed systems using middleware.

Requirements Engineering

The challenges of co-ordination, reliability, scalability and heterogeneity in distributed system construction that we discussed in Section 2 and that engineers are faced with are of a *non-functional* nature. Software engineers thus have to define software architectures that meet these non-functional requirements. However, the relationship between non-functional requirements and software architectures is only very poorly understood. We first discuss the requirements engineering end of this relationship.

Existing requirements engineering methods tend to have a very strong focus on functional requirements. In particular the object-oriented and use-case driven approaches of Jacobsen [27] and more recently Rational [26] more or less completely ignore non-functional concerns. A goal-oriented approach, such as [10] seems to provide a much better basis, but needs to be augmented to specifically address non-functional concerns.

For non-functional goals to be a useful input to middleware-oriented architecting, these goals need to be *quantified*. For example, in order to engineer scalable architectures, engineers need to have quantitative requirements models for the required response time, peak loads and overall transaction or data volume that an architecture is expected to scale up to. Thus requirements engineering research needs to devise methods and tools that can be used to elicit and model non-functional requirements from a quantitative point of view.

Once a particular middleware system has been chosen for a software architecture, it is extremely expensive to revert that choice and adopt a different middleware or a different architecture. The choice is influenced by the non-functional requirements. Unfortunately, requirements tend to be unstable and change over time. Non-functional requirements often change with the setting in which the system is embedded, for example when new hardware or operating system platforms are added as a result of a merger, or when scalability requirements increase as a result of having to build web-based interfaces that customers use directly. Requirements engineering methods, therefore, not only have to identify the current requirements, but also elicit and estimate the ranges in which they can evolve during the planned life time of the distributed system.

Software Architecture

There is only very little work on the influence of middleware on software architectures, with [11] being a notable exception. Indeed, we believe that research on software architecture description languages has over-emphasized func-

tionality and not sufficiently addressed the specification of how global properties and non-functional requirements are achieved in an architecture. These requirements cannot be attributed to individual components or connectors and can therefore not be specified by current architectural description languages.

Distributed software engineering research needs to identify notations, methods and tools that support *architecting*. Research needs to provide methods that help software engineers to systematically derive software architectures that will meet a set of non-functional requirements and overcome the guesswork that is currently being done. This includes support for identifying the appropriate middleware or combinations of middlewares for the problem at hand. Moreover, software engineering research needs to define architecting processes that are capable of mitigating the risks of choosing the wrong middleware or architectures. These processes will need to rely on methods that quantitatively model the performance and scalability that a particular middleware-based architecture will achieve and use validation techniques, such as model checking, to validate that models actually do meet the requirements. The models need to be calibrated using metrics that have been collected by observing middleware performance in practice.

Many architecture description languages support the explicit modeling of connectors by means of which components communicate [40]. A main contribution of [11] is the observation that connectors are most often implemented using middleware primitives. We would like to add the observation that each middleware only supports a very limited set of connectors. Specifying the behaviour of connectors explicitly in an ADL is therefore modelling overkill that is only needed if architects opt out of using middleware at all. For most applications, the specification of each connector is completely unnecessary. Instead, software architecture research should develop middleware-oriented ADLs that have built-in support for all connectors provided by the middleware that practitioners actually use.

Design

In [13], we have argued that the use of middleware in a design is not, and never will be, entirely transparent to designers. There are a number of factors that, despite of the ISO/ODP transparency dimensions, necessitate designers to be aware of the involvement of middleware in the communication between components. These factors are:

- Network latency implies that the communication between two distributed components is by orders of magnitude slower than a local communication.
- Component activation and de-activation of stateful components lead to a need for implementing persistence of these components.
- Components need to be designed so that they can cope with the concurrent interactions that occur in a dis-

tributed environment.

- The components have a choice of the different synchronization primitives a particular middleware offers, and need to exploit them properly. In particular, they have to avoid the deadlocks or liveness problems that can occur as a result of using these synchronization primitives.

The software engineering community needs to develop middleware-oriented design notations, methods and tools that take the above concerns into account.

Discussing the state of the art middleware research above, we have highlighted a trend to give the programmer more influence on how the middleware behaves. Globe's replication strategies, TAO's scheduling policies and reflection capabilities that influence the middleware execution engine have to be used by the designer. This means, effectively, that the programmer gets to see more of the middleware and that distribution and heterogeneity become less transparent. If this is really necessary, and the middleware research community puts forward good reasons, programmers will have to be aided even more in the design of distributed components. Thus appropriate principles, notations, methods and tools for the design of replication strategies, scheduling policies and the use reflection capabilities are needed from software engineering research.

6 SUMMARY

We have discussed why the construction of distributed systems is difficult and indicated the support that software engineers can expect from current middleware products to simplify the task. We have then reviewed the current state of the art in middleware research and used this knowledge to derive a software engineering research agenda that will produce the principles, notations, methods and tools that are needed to support all activities during the life cycle of a software engineering process.

REFERENCES

- [1] J. Bates. The State of the Art in Distributed and Dependable Computing. Technical report, Laboratory for Communications Engineering, Cambridge University, <http://www.newcastle.research.ec.org/cabernet/sota/report>, Oct. 1998.
- [2] M. Bearman. ODP-Trader. In *Proc. of the IFIP TC6/WG6.1 Int. Conf. on Open Distributed Processing, Berlin, Germany*, pages 341–352. North-Holland, 1993.
- [3] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison Wesley, 1987.
- [4] K. P. Birman. *Building Secure and Reliable Network Applications*. Manning Publishing, 1997.
- [5] D. Box. *Essential COM*. Addison Wesley Longman, 1998.
- [6] J. Charles. Middleware Moves to the Forefront. *IEEE Computer*, pages 17–19, May 1999.

- [7] S. Cheung. *Java Transaction Service (JTS)*. Sun Microsystems, 901 San Antonio Road, Palo Alto, CA 94303, Mar. 1999.
- [8] P. Chung, Y. Huang, S. Yajnik, D. Liang, J. Shin, C.-Y. Wang, and Y.-M. Wang. DCOM and CORBA: Side by Side, Step by Step, and Layer by Layer. *C++ Report*, pages 18–29, January 1998.
- [9] P. Cointe, editor. *Meta-Level Architectures and Reflection: 2nd International Conference, Reflection '99, St. Malo, France*, volume 1616 of *Lecture Notes in Computer Science*. Springer, 1999.
- [10] A. Dardenne, A. van Lamswerde, and S. Fickas. Goal-directed Requirements Acquisition. *Science of Computer Programming*, 20:3–50, 1993.
- [11] E. di Nitto and D. Rosenblum. Exploiting ADLs to Specify Architectural Styles Induced by Middleware Infrastructures. In *Proc. of the 21st Int. Conf. on Software Engineering, Los Angeles, California*, pages 13–22. ACM Press, 1999.
- [12] F. Eliassen, A. Andersen, G. S. Blair, F. Costa, G. Coulson, V. Goebel, O. Hansen, T. Kristensen, T. Plagemann, H. O. Rafaelsen, K. B. Saikoski, and W. Yu. Next Generation Middleware: Requirements, Architecture and Prototypes. In *Proceedings of the 7th IEEE Workshop on Future Trends in Distributed Computing Systems*, pages 60–65. IEEE Computer Society Press, Dec. 1999.
- [13] W. Emmerich. *Engineering Distributed Objects*. John Wiley & Sons, Apr. 2000.
- [14] W. Emmerich, A. Finkelstein, and W. Schwarz. Markup Meets Middleware. In *7th Int. Workshop on Future Trends in Distributed Systems, Capetown, South Africa*, pages 261–266. IEEE Computer Society Press, 1999.
- [15] FpML. Introducing FpML: A New Standard for e-commerce. <http://www.fpml.org>, 1999.
- [16] L. Gilman and R. Schreiber. *Distributed Computing with IBM MQSeries*. Wiley, 1996.
- [17] A. Goldberg. *Smalltalk-80: The Language and its Implementation*. Addison Wesley, 1985.
- [18] J. N. Gray. Notes on Database Operating Systems. In R. Bayer, R. Graham, and G. Seegmüller, editors, *Operating systems: An advanced course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F., pages 393–481. Springer, 1978.
- [19] C. L. Hall. *Building Client/Server Applications Using TUXEDO*. Wiley, 1996.
- [20] M. Hapner, R. Burridge, and R. Sharma. Java Message Service Specification. Technical report, Sun Microsystems, <http://java.sun.com/products/jms>, Nov. 1999.
- [21] S. Hillier. *Microsoft Transaction Server Programming*. Microsoft Press, 1998.
- [22] E. S. Hudders. *CICS: A Guide to Internal Structure*. Wiley, 1994.
- [23] Infinity. Infinity Network Trade Model Overview. <http://www.infinity.com/ntm/pdf/ntmOverview.pdf>, 1999.
- [24] ISO 10746-1. Open Distributed Processing – Reference model. Technical report, International Standardization Organization, 1998.
- [25] ISO 7498-1. Information processing systems – Open Systems Interconnection – Basic Reference Model: The Basic Model. Technical report, International Standards Organisation, 1994.
- [26] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison Wesley, 1999.
- [27] I. Jacobson, M. Christerson, P. Jonsson, and G. Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley, 1992.
- [28] JavaSoft. *Java Remote Method Invocation Specification*, revision 1.50, jdk 1.2 edition, Oct. 1998.
- [29] G. Kiczales, J. d. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol*. MIT Press, 1991.
- [30] R. Monson-Haefel. *Enterprise Javabeans*. O'Reilly UK, 1999.
- [31] B. C. Neuman. Scale in Distributed Systems. In T. Casavant and M. Singhal, editors, *Readings in Distributed Computing Systems*. IEEE Computer Society press, 1994.
- [32] Object Management Group. *CORBA services: Common Object Services Specification, Revised Edition*. 492 Old Connecticut Path, Framingham, MA 01701, USA, December 1998.
- [33] Object Management Group. *Notification Service*. 492 Old Connecticut Path, Framingham, MA 01701, USA, Jan. 1998.
- [34] Object Management Group. *The Common Object Request Broker: Architecture and Specification Revision 2.2*. 492 Old Connecticut Path, Framingham, MA 01701, USA, February 1998.
- [35] Object Management Group. *XML/Value Request for Proposals*. 492 Old Connecticut Path, Framingham, MA 01701, USA, Aug. 1999.
- [36] Open Group, editor. *DCE 1.1: Remote Procedure Calls*. The Open Group, 1997.
- [37] R. Orfali, D. Harkey, and J. Edwards. *Instant CORBA*. Wiley, 1997.
- [38] G.-C. Roman, A. L. Murphy, and G. P. Picco. A Software Engineering Perspective on Mobility. In A. C. W. Finkelstein, editor, *Future of Software Engineering*. ACM Press, 2000.
- [39] D. Schmidt, C. Gill, and D. Levine. Evaluating Strategies for Real-Time CORBA Dynamic Scheduling. In *19th International IEEE Real-Time Symposium*. IEEE Computer Society Press, 1998.
- [40] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
- [41] J. Siegel. Component and Object Technology: A Preview of CORBA 3. *IEEE Computer*, pages 114–116, May 1999.
- [42] M. v. Steen, P. Homburg, and A. S. Tanenbaum. Globe: A Wide-Area Distributed System. *IEEE Concurrency*, pages 70–78, January-March 1999.
- [43] X/Open Group. Distributed Transaction Processing: The XA+ Specification, Version 2. X/Open Company, ISBN 1-85912-046-6, Reading, UK, 1994.