Middleware: Past and Present a Comparison

Hennadiy Pinus

ABSTRACT

The construction of distributed systems is a difficult task for programmers, which can be simplified with the use of middleware. Middleware can be seen as a layer between applications and operating systems. There are four main types of middleware: *transactional, message-oriented, procedural and object-oriented* middleware. The first three types of middleware can be seen as a past and the last one as a present middleware. The variety of middleware types leads to a choice problem for the software developers. In order to solve it, we research the classification of the middleware, pro and contra of each middleware type. This review is based on analyzing of middleware through the following requirements, proposed by [9]: network communication, coordination, reliability, scalability and heterogeneity. During this paper we try to find out, how good the middleware types support these requirements.

Keywords

RPC, MOM, marshalling, TP, middleware

1. INTRODUCTION

Nowadays a great number of transactions is realized via internet and networks. The "networks were introduced to connect PCs, workstations and mainframes, and there was a strong technical drive from the software and computer industry towards distributed computing." (see [8], p.1) *Distributed system(DS)* is "a computer system, where components of the system are held on physically separated, autonomous computers." (see [14], p.1) These computers are connected by a communication network, which enables the integration of components. The time available for an integration of the components is often too short, that's why DS seems to be a better choice than building a new system. The integrated components may include existing or old components (*legacy components*). However, it is considerably more difficult to construct a DS, than to build a centralized or completely new system. In a DS components have to communicate with each other on the multiple points via a network, thus increasing the likelihood of errors. The solution of the component integration into a DS can be found by using middleware [9, 14].

This paper is structured in the following way. At first we discuss the definition of the middleware (Section 2). Then we define the requirements that middleware has to support (Section 3). The classification of middleware is presented as next. For each type of the middleware we analyze, how it fulfils the requirements (Section 4). At the end of the work we make comparative conclusions concerning middleware types (Section 5).

2. WHAT IS MIDDLEWARE?

The term middleware first appeared in the late 1980s. It was used to describe network connection management software. In the middle 1990s, when network technology had achieved sufficient penetration and visibility, it found its widespread [4]. There are many definitions of middleware found in literature. In common, middleware can be defined as a software layer located above the OS and networking software and below applications. (see Figure 1) Middleware enables the

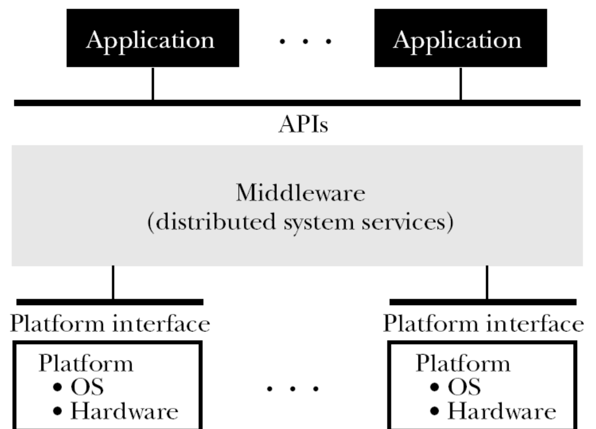

Figure 1: Middleware in Distributed System, taken from [7], p. 89

interaction and communication between different applications through Application Programming Interfaces (APIs) across the distributed components [7, 8, 13]. API "is the interface by which an application program accesses operating system and other services". (see [1]) Middleware can be seen as a common development and runtime environment

that enables the connection of components written in different languages and running on different operating systems [14]. The goal of middleware is to simplify the construction of the DS, where the application engineers can "abstract from the implementation of low-level details, such as concurrency control, transaction management and network communication." (see [9], p.120) Due to middleware developers can concentrate on application requirements.

3. MIDDLEWARE REQUIREMENTS

The following requirements were proposed by [9]: network communication, coordination, reliability, scalability and heterogeneity.

Network Communication: Distributed systems need a network communication, because the components are located on different hosts. The network communication is based on network protocols. The communication requires the transformation of the complex data structures into a suitable format, which can be transmitted using transport protocols. These transformations are called marshalling (component requesting service passes parameters to a message request) and unmarshalling. To enable automatical (un-) marshalling it is required, that all data involved in a request have to be described. It can be done with an *Interface Definition Language* (IDL) [4, 9].

Coordination: Coordination is required to control multiple communication points, which exist in distributed systems. There are several mechanisms, which can influence coordination of the components in distributed system. These include synchronization and activation (deactivation) policies. *Synchronization* is useful during the communication of the concurrent components on the same host. There are several ways to achieve synchronization. The one way, where a component is blocked till the other component completes execution of a requested service, is called *synchronous*. If the component that asks some service from another component remains unblocked and can continue to perform its operations, then this way is called *deferred synchronous* (if service request was initiated by client) or *asynchronous* (if initiated by server). Other coordination mechanisms are *activation and deactivation policies*. *Activation (deactivation)* allows to start (end) a component independently from the applications, which it executes. One of the reasons to use these policies is that "components may be idle for long periods, thus wasting resources if they were kept in virtual memory all the time." (see [9], p.121) The middleware needs to support mechanisms called *threading policies* in order to control, how the server reacts on the concurrent requests [9].

Reliability: There are several types of reliability proposed in the literature: best effort, at-most-once, at least-once and exactly-once. "*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. A*t-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." (see [9], p.121) The usage of the definite network protocol can also influence reliability, because different protocols have different reliability levels. Distributed system implementations need to include error detection and correction mechanisms to liquidate the unreliabilities caused by errors. If some components in the system are not available, then the reliability of the system suffers. The increase of the reliability in such situation is possible, if using a replica component. Therefore, replication of the components increases reliability. The programmers, however, need to know that increase of performance decreases reliability. It is a trade-off problem [9].

Scalability: Scalability defines, how well a hardware or a software system can adapt to increased demands [2]. The limited scalability in centralized systems can be overcome by DSs. The main task is to provide the changes in DS without changing its architecture or design. In order to achieve this task, it is desired that middleware respects different dimensions of transparency specified by the ISO Open Distributed Processing (ODP) reference model. For example, *access transparency* means that a component can access the services of the remote component, as if it were local. *Location transparency* means that the components are not aware of the physical location of the interacted components. *Migration transparency* allows components to change their location, which is not seen for a client requesting these components. *Replication transparency* means that requesting components don't care about the place of the needed service. It can either be main component or its replica. *Load balancing*, where requests are forwarded to a replica in order to release a loaded server, can use replication mechanisms. Hence, "components whose services are in high demand may have to exist in multiple copies." (see [9], p.122) A scalable system middleware needs to support access, location, migration and replication transparency [9].

Heterogeneity: Heterogeneity is the quality of being diverse and not comparable in kind [1]. The components of distributed systems can be of different types (legacy or new components) and written in different languages. For example, 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." (see [9], p.122) There are different dimensions of heterogeneity in DSs: "hardware and operating system platforms, programming languages and indeed the middleware itself." (see [9], p.122) These differences need to be resolved by the middleware.

In the next section we'll see, how good the different middleware types support the defined requirements.

4. MIDDLEWARE CLASSIFICATION

In the literature one can found four main types of middleware. These are: transactional, procedural, message-oriented and object-oriented middleware [5, 6, 9, 14]. The first three types of middleware can be seen as a past and the last one as a present middleware.

4.1 Transactional Middleware (TM)

Transactional middleware(TM) or transaction processing (TP) monitors were designed in order to support distributed synchronous transactions. The main function of a TP monitor is a coordination of requests between clients and servers that can process this requests. *Request* is a "message that asks the system to execute a transaction." (see [16], p.94) TM uses clustering ("a grouping of a number of similar things" [1]) of the service requests into transactions. "A transaction must support *ACID* properties - Atomic, Consistent, Isolated, and Durable. Atomic means "all or nothing" qual-

ity of transactions. The transaction either completes or it does not. Consistency should hold the system in a consistent state, independent of the status of a transaction. Isolation is ability of one transaction to work independently from other transactions that possibly run on the same TP monitor. Durability means the ability of a transaction to survive system failures, if a transaction is once committed and complete ." (see [11], p.2)
Typical products: IBM's CICS, BEA's Tuxedo, Transarc's Encina [9].
Network Communication: 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 [9].
Coordination: TP monitors can coordinate the distributed transactions through the *two-phase commit protocol* (2PC). This protocol is based on the "prepare to commit phase" and the "commit phase" [3]. Both, synchronous and asynchronous communication are supported by TM. The client components can request services using these communication types. TM has a support for various activation policies and services that can be activated and deactivated on demand, if they haven't been used for some time. The server components can always reside in memory, hence enabling a persistent activation. Many TP monitors support failover and possess restart capabilities, thus increasing application up time [4, 9].
Reliability: TM requires the 2PC to implement distributed transactions. The transaction can be committed, only if all processes involved in a transaction are ready to commit, otherwise the transaction is aborted [3, 9, 16]. TP monitors use transactions logs, which can undo changes [17]. The Failure/recovery service, which is supported by most TP monitors, increases fault-tolerance and reliability consequently. Message queues are also supported by TM, thus enabling reliability, when disk storage is used for queues. TM also supports database management systems (DBMS), which guarantee fault-tolerance [9, 17].
Scalability: TP monitors are rather scalable, because they support load balancing and replication of server components. Load balancing is important, because TP monitors have to cope with a lot of transactions in a limited time [9, 17]. "In order to sustain consistent response times, TP monitors are capable of starting additional process instances. This is an important feature for any enterprise environment that needs to have ensured scalability." (see [4], p. 11)
Heterogeneity: The heterogeneity support is realized on different levels. TM supports soft- and hardware heterogeneity, because the components can be located on different hardware and operating system platforms. TM, as mentioned above, has a DBMS support. DBMS components can participate in transactions due to the Distributed Transaction Processing (DTP) Protocol, adopted by the Open Group. But TM doesn't support data heterogeneity very well, because it can't express complex data structures and therefore can't marshall these structures [9].
Advantages: 1. Components are kept in consistent states (due to ACID properties of transaction) 2. TM is very reliable. 3. TP monitors perform better than message-oriented and procedural middleware. 4. TP monitors can dispatch, schedule and prioritize multiple application requests concurrently, thus reducing CPU overhead, response times and CPU cost for large applications [4, 9, 13].
Disadvantages: 1. TM has often unnecessary or undesirable guarantees according to ACID. "If a client is performing long-lived activities, then transactions could prevent other clients from being able to continue." (see [14], p.3) 2. Marshalling and unmarshalling have to be done manually in many products 3. The lack of common standard for defining the services that server components offer reduces the portability of a DS between different TP monitors . 4. TM runs on less amount of platforms (only UNIX and NT server) than other middleware types [4, 9, 14].
Where to use: TP monitors should be used, when "transactions need to be coordinated and synchronized over multiple databases." (see [5], p.5)

4.2 Message-oriented Middleware (MOM)

Message Oriented Middleware (MOM) is a middleware, that enables a communication through messages. According to [5] there are two different types of MOM: message queuing and message passing. *Message queuing* is defined as indirect communication model, where communication happens via a queue. Message from one program is sent to a specific queue, identified by name. After the message is stored in queue, it will be sent to a receiver. In *message passing* - a direct communication model - the information is sent to the interested parties. One flavor of message passing is publish-subscribe (pub/sub) middleware model. In pub/sub clients have the ability to subscribe to the interested subjects. After subscribing, the client will receive any message corresponding to a subscribed topic [5, 8, 10].
Typical products: IBM's MQSeries, Sun's Java Message Queue [9].
Network Communication: Network communication in MOM is based on messages. "*Messages* are strings of bytes that have meaning to the applications that exchange them. Besides application related data, messages might include control data relevant to the message queuing system only. This information is used to store, route, deliver, retrieve and track the payload data." (see [4], p.6) After receiving a message from a component, server replies with a message, which contains the results of the service execution. In comparison to RPCs, most messaging products have a good support for many additional communication protocols [4, 9].
Coordination: MOM supports both synchronous (via message passing) and asynchronous (via message queuing) communication. Asynchronous communication is achieved in the natural way. The message is sent to a server, without blocking a client. The client does not need to wait for a reply and can proceed with other actions. However, the synchronous communication needs to be implemented manually in the client. MOM supports the activation on demand. It is realized with *triggers*, "where an application program is started whenever a request message or a reply message has arrived on a local queue, and the application program is not already active." (see [4], p.8) It decreases the use of resources [4, 9, 14].
Reliability: MOM can be seen as fault tolerant, because it can use persistent queues, which are stored on the hard disc. This type of queue "is most appropriate where applications cannot be connected directly (for example, in mobile computing)." (see [5], p.6-7) Message queues are of two types: persistent and non-persistent and are managed by the queue manager. If server fails, it is guaranteed in the former case, that information will be restored after server restarts. Per-

sistent queues are to choose when the reliability is more important than performance, like in banking fund transfer. To increase reliability message queuing supports different Quality of Service *(QoS)* [4, 5, 9]. These QoS are defined in [4] as: "1. Reliable message delivery - during exchange of messages no network packets are lost. 2. Guaranteed message delivery - messages are delivered to the destination node either immediately (with no latency - network is available), or eventually (with latency - when the network is unavailable). In the latter case, middleware guarantees that messages are delivered as long as the network becomes available within a specified time period. 3. Assured, non-duplicate message delivery - if the messages are delivered, they are delivered only once." (see [4], p.8)
Scalability: "The publish-subscribe communications model provides location transparency, allowing a program to send the message with a subject as the destination property while the middleware routes the message to all programs that have subscribed to that subject." (see [5], p.7) Although location transparency is supported, MOMs have a limited support for access transparency. It happens, because queues are used for remote and not for local communication. If access transparency doesn't exist, it leads to a lack of migration and replication transparency, thus complicating 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." (see [9], p.123)
Heterogeneity: The support of data heterogeneity is rather limited, because marshalling is not automatically generated, and needs to be implemented by programmers [9].
Advantages: 1. MOM supports group communication, which is atomic. Either all clients receive a delivery or none. That's why a process doesn't have to worry about what to do, if some clients don't receive a message. 2. The use of persistent queues increases reliability in MOM products. 3. Support for transactional message queues is included in most MOM products, meaning that advanced delivery guarantees are supported. 4. MOM supports more network protocols than RPCs do [10, 16]. 5. MOM can send the message exactly-once due to QoS, thus increasing it's reliability.
Disadvantages: 1. MOM has limited scalability and heterogeneity support [9]. 2. There is a bad portability support, because MOM products don't support any standards. Applications that are made for one MOM product are not compatible to another MOM product [4].
Where to use: MOM can be used in the applications, where the network or all-components availability is not warranted [4].

4.3 Procedural Middleware (PM)

Remote Procedure Calls (RPCs) were developed by Sun Microsystems in the early 1980s. RPCs are represented on different operating systems, including most Unix and MS Windows systems. "Windows NT, for example, supports lightweight RPCs across processes and, with DCOM, full RPCs." (see [15], p.172)
Typical products: Open Software Foundation's Distributed Computing Environment DCE, Microsoft RPC Facility [9].
Network Communication: RPCs define server components as RPC programs. An RPC program contains parameterized procedures. Remote clients can invoke these procedures across the network using the network protocols. These protocols are low-level, such as Transmission Control Protocol/Internet Protocol (TCP/IP) or User Datagram Protocol (UDP). The communication happens in the following way. If a client wants to receive some services, then it makes a request to a server. This request consists of a message, which includes the marshalled parameters. On the other side, the server receives this message, unmarshalls the parameters, executes the requested service and sends the result back to the client [9, 13]. "In order to connect the client and server components of a distributed application using RPCs as the middleware link, it is required that every function that a client application can call should be represented by a *stub*, i.e. a placeholder, for the real function on the server." (see [4], p.5) We can note, that RPC is the category of middleware, where marshalling and unmarshalling are implemented automatically by the IDL compiler in stubs, hence making the life easier for developers [9, 13, 14].
Coordination: "Typical RPC-based communication is synchronous, i.e., an RPC client is blocked until the remote procedure has been executed or an error occurs." (see [10], p.19) RPCs don't support asynchronous communication. Procedural middleware provides different forms of activation policies. These include activation of RPC on demand and "RPC is always available".
Reliability: Procedural middleware possesses an at-most once reliability. If RPC fails, then an exception is returned. RPCs' communication, based on TCP/IP protocol, can be seen as reliable, because TCP/IP provides reliability [9, 12, 14]. "An application that uses TCP knows that data it sends is received at the other end, and that it is received correctly." (see [12], p.16)
Scalability: The scalability of RPCs remains rather limited, because RPCs lack replication mechanisms. As mentioned above the communication is based on stubs, which provide location transparency of the requested service to the client. It means, that client can invoke a remote procedure as if it were local [10]. As we know from the requirements section location transparency is a prerequisite to scalability [9].
Heterogeneity: The heterogeneity in RPCs is realized via IDL. It can define interfaces that represent relations between servers and clients. IDL is programming language independent, which means that client doesn't need to know the language that server supports, if IDL compiler can translate the client's request to a server [4].
Advantages: 1. RPC has a good heterogeneity support, because "RPC has bindings for multiple operating systems and programming languages." (see [14], p.3) 2. Marshalling and unmarshalling are automatically generated, thus simplifying the development of DSs [14].
Disadvantages: 1. RPCs don't support group communication [16]. 2. They have no direct support for asynchronous communication, replication and load balancing, therefore leading to a limited scalability. 3. Fault tolerance is worse than by other middleware types, because "many possible faults have to be caught and dealt with in the program." (see [14], p.4)
Where to use: According to [5], RPCs could be used in small, simple applications with primarily point-to-point communication. "RPCs are not a good choice to use as the building blocks for enterprise-wide applications where high performance and high reliability are needed." (see [5], p.6)

4.4 Object-oriented Middleware (OOM)

Object middleware, evolved from RPCs, extends them by adding object-oriented concepts. These concepts are: inheritance, object references and exceptions. OOM allows referencing of remote objects and can call operations on them [9, 10, 14].

Typical products: OMG's CORBA, Microsoft COM, Java RMI and Enterprise Java Beans [5, 9, 10].

Network Communication: Object middleware supports distributed object requests. It is possible for a client to request an operation on a server object on the other host. The one thing required for a client is a reference to the server object. Marshalling of the parameters for the network request is made automatically in the client and server stubs [9].

Coordination: OOM generally supports a synchronous communication. The client object remains blocked, waiting for the server object response. It doesn't mean that other synchronization types are not supported. "CORBA 3.0, for example, supports both deferred synchronous and asynchronous object requests." (see [9], p.125) OOM supports different activation and threading policies. In OOM server objects can be active all the time or started on demand. CORBA Concurrency Service enables threading and coordinates the concurrent access to shared resources. It also guarantees the consistency of the object, if it is accessed by multiple clients [4, 9, 13]. One should mention that CORBA "supports group communication through its Event and Notification services." (see [9], p.125)

Reliability: At most once reliability is set by default in OOM. To handle failures during component requests OOM uses exceptions. Noticeable is the fact that due to CORBA messaging exactly-once reliability can be achieved. Fault-Tolerant CORBA, for example, provides extra reliability in DS [9, 10, 14].

Scalability: Scalability remains limited. Some CORBA implementations support load-balancing. Enterprise Java Beans have a replication support, which increases the scalability [9].

Heterogeneity: OOM has a wide support for heterogeneity. For example, "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." (see [9], p.123) Java/RMI resolves the heterogeneity with its Java Virtual Machine [9].

Advantages: 1. Marshalling and unmarshalling are generated automatically in client and server stubs. 2. OOM supports both synchronous and asynchronous communication. 3. Most OOM products have a support for messaging and transactions. Hence, OOM can replace other three types of middleware in many different aspects. It makes OOM to a powerful and flexible middleware type [10, 14].

Disadvantages: 1. Lack of scalability.

Where to use: OOM "should be considered for applications where immediate scalability requirements are somewhat limited. These applications should be part of a long-term strategy towards object orientation." (see [4], p.22)

5. CONCLUSIONS

As we have seen, the variety of middleware types can be used in different situations (see "Where to use" subsections). Some of them can be combined to achieve the needed requirements. We can note, that all types of middleware have a limited support for scalability. The most reliable are TM and OOM. TP monitors perform better than MOM and RPCs. RPCs and OOM support an automatical marshalling, thus simplifying the network communication. Unlike MOM and OOM, RPC and TM don't support group communication. The use of IDL in RPCs and OOM makes them very heterogenous. Noticeable, that all types of middleware support the activation on demand, thus saving resources. OOM integrates the benefits of TM, MOM and RPCs and tends to replace them. At last, the choice of the middleware depends on the task to realize.

6. REFERENCES

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

[2] http://www.webopedia.com.

[3] http://www.cs.panam.edu/~meng/Course/CS6334/Note/master/node88.html.

[4] Middleware white paper, 1997. http://web.cefriel.it/~alfonso/WebBook/Documents/isgmidware.pdf.

[5] Talarian: Everything you need to know about middleware, 2000. http://searchwebservices.techtarget.com/searchWebServices/downloads/Talarian.pdf.

[6] D. E. Bakken. Middleware. http://www.eecs.wsu.edu/~bakken/middleware-article-bakken.pdf.

[7] P. A. Bernstein. Middleware: a model for distributed system services. *Communications of the ACM*, pages 86 – 98, 1996.

[8] R. P. Bob Hulsebosch, Wouter Teeuw. Middleware tintel state-of-the-art deliverable. 1999.

[9] W. Emmerich. Software engineering and middleware: A roadmap. *Communications of the ACM*, pages 117 – 129, 2000.

[10] C. Hartwich. A middleware architecture for transactional, object-oriented applications, 2003. http://page.mi.fu-berlin.de/~hartwich/diss-final.pdf.

[11] D. S. Linthicum. Application servers an eai. *eAI Journal*, July/August 2000.

[12] J. Moss. Understanding tcp/ip. *PC Network Advisor*, 1997.

[13] J. M. Myerson. *The Complete Book of Middleware*. Auerbach Publications, 2002.

[14] R. Nunn. Distributed software architectures using middleware. http://www.cs.ucl.ac.uk/staff/W.Emmerich/lectures/3C05-02-03/aswe18-essay.pdf.

[15] C. Szyperski. *Component Software*. Addison Wesley, 1999.

[16] A. S. Tanenbaum. *Distributed Operating Systems*. Prentice Hall, 1995.

[17] T. C. E. A. Workgroup. Middleware architecture report:, 2001. http://www.vita.virginia.gov/docs/ea/MiddlewareArchitectureV1-0-051801.pdf.

All URLs, mentioned in references, were last accessed on the 20.06.04.