1. Communication Middleware

Distributed processing implies that many separate components of a distributed application execute on different end-systems which are interconnected by a network. Distributed applications have gained a significant importance as low-priced personal computers, workstations and servers became available, and with the emergence of TCP/IP technology, which provides a ubiquitous means for the exchange of information between autonomous computers. This new technology enables geographically distributed companies to exchange information fast and efficiently, and therefore answers their needs to cooperate tightly. Fast access to data, resource sharing, load balancing, availability, and fault tolerance are the primary reasons for using distributed approaches. Generally, a request-response type of communication is established between two processes in a distributed system. The requesting side is called the client and the responding one is called server. Client/server applications and appropriate communication paradigms are related closely. Due to heterogeneous hardware and software of clients and servers, a direct (bit-by-bit) exchange of information between participants is often not possible. However, transparently hiding away these differences from applications themselves is feasible by providing appropriate high-level interfaces and interoperable protocols.

The basic mechanisms for information exchange are send and receive operations where a sender specifies the destination and a message to be sent [CDKi94]. The exchange may take place in an asynchronous manner, with a send operation that is non-blocking, allowing the sender to proceed after having emitted the message. Alternatively, a synchronous exchange may be used, where the send operation is blocking, leaving the sender in a wait state to synchronize with the expected response to be received. Besides location-independent identifiers for destinations within messages, where these identifiers are mapped onto explicit lower layer addresses for message delivery, reliability aspects are important. Loss, duplication, corruption, or non-timely delivery of messages may occur while two processes are communicating.
The model of Remote Procedure Calls (RPC) as depicted in Figure 1 is a well established communication paradigm for supporting client/server applications with different semantics with respect to reliability.

Developing client/server applications is a complex task and can be performed by explicitly programming the application and communications between clients and servers. However, this interrelates deep application knowledge and communication knowledge for the developer at the same time. Following the software engineering approach of the “separation of concerns”, client/server application aspects should remain with the application developer and networking aspects are provided by a suitable programming interface which itself is offered by the communication middleware, a software layer residing between the implementation of the networking functions, and the application code.

Basically, a communication middleware provides a platform to develop distributed applications; additionally, it simplifies their design, since communication-relevant details are hidden. Services offered by middleware are at least twofold in providing communication and networking access which is discussed in detail in Paragraph 1.2.3. Often the term “middleware” is used as a synonym for a communication subsystem which defines an architecture providing communication services, extending the pure interface view to an entire system and architectural view. Middleware for distributed objects, such as CORBA [OMG95] or DCE [DEC92], have been designed to support interoperability, portability, and object-oriented specification methods. We will investigate both CORBA and DCE in detail further below. Todays developments of distributed information systems and middleware offer full-fledged architectures, including a variety of services, such as file, directory, name, security, or time services, and they support a broad range of underlying networks and communication protocols.

In addition to these generic middleware aspects, research in communication support for modern distributed applications has focused on two different areas. On the one hand, the functionality of communication subsystems providing a variety of different communication protocols and features has been extended far beyond traditionally layered architectures. These approaches are often Quality-of-Service-oriented (QoS) and offer facilities for exchanging a broad range of multimedia data with different requirements. On the other hand, existing communication architectures, e.g., the Internet and OSI protocol suites, form a major component of a communication subsystems and have undergone considerable redesign efforts with the goal of accommodating advanced technology, fault-tolerant communication paradigms, and flexible service concepts. These approaches need to be combined, as real-time requirements of multimedia applications can be fulfilled only, if suitable design principles are applied to both network-oriented communication functions and application-oriented functions in end-systems.
At the boundary between communication subsystems and applications we need a well-defined, comprehensive and easy-to-use interface for application programmers (applications programmers’ interface, API) which is independent of the underlying communication subsystem, providing abstract communication functions, and assisting the efficient exchange of control and user data between remote applications and between the application and the local communication subsystem. The API defines the crucial user or application access point to middleware for communications. It must transparently hide away details concerning communication issues, such as specific communication protocol features, network performance, or network topology information. However, a well designed and fine granular approach to specify communication requirements in terms of QoS is required. Interfaces and behaviors of objects are specified by interface definition languages, forming a subset of formal methods such as discussed within Section 2.

1.1 Traditional Approaches

Candidate applications using communication middleware services encompass, among others, real-time control and management applications, such as industrial plant control or virtual reality systems; remote surgery; and continuous media applications, such as desktop video-conferencing systems, picture phones, or tele-seminar applications. In general, client/server and networked multimedia applications combine basic communication tasks (low-level) with advanced service, maintenance, and management tasks (high-level). However, not every application suits the client/server model, e.g. applications with a one-to-many communication pattern. Therefore, extended approaches are required to overcome the missing flexibility inherent in RPC-based approaches. This leads to a new view of communications in which sophisticated services tailored to specific fields of applications are made available to the applications programmer. Future distributed systems must be robust, scalable, and reusable, and offer inherent service flexibility and high-performance. Therefore, communication middleware is a suitable and universal model for allocating high- and low-level functionality needed on top of basic network services. In addition, portability of communication middleware is a central requirement and, in turn, eases application portability.

As the performance and functionality of communication networks increased over the last twenty years, a broad range of low-level services has become available. Traditional communication systems have been specified and modeled with a hierarchical approach; the best-known reference is the Open Systems Interconnection (OSI) Basic Reference Model, developed by the International Organization for Standardization (ISO) [ISO7498], which uses a hierarchy of seven layers which accommodates all functions needed in a communication system, assigning them to a specific layer. The Internet model, is similar in principle, but very different in detail, using only four layers and being less formal in its specification. Implementation of these traditional approaches may be classified as examples of an early, but low-level communication middleware. However, they are completely bound to a particular communication protocol stack.

Today, traditional communication systems have various shortcomings, especially concerning the support of distributed applications on high-performance networks. Therefore, alternative approaches have been developed, focussing on (1) middleware and communication protocol design, (2) efficient software implementation, and (3) hardware support, and resulting in the development of high-speed or light-weight protocols. Furthermore, the provision of flexible communication services by adjustable protocols according to application requirements based on QoS parameters has been studied, e.g., [CCHu94], [Dant94], or [ZSTa93]. Flexible protocol
architectures were designed to minimize protocol processing efforts and have been dealt with in various projects, *e.g.*, Dynamic Configuration of Protocols (Da CaPo) [PPVW92], the Function-based Communication Subsystem (F-CSS) [ZSTa93], or ADAPTIVE [ScSu93]. On higher levels and protocol-independent layers general message-based inter-process communication mechanisms have been replaced by Remote Procedure Calls (RPC) due to their simplicity of use and transparency [CDKi94]. Today, RPC-based middleware tends to be superseded by distributed object-oriented technologies, such as CORBA, due to their more powerful semantics and increased flexibility for different application domains. However, available implementations of these technologies still suffer from poor performance, *e.g.*, concerning their marshalling functions or stub compilers for high-speed networking environments [GoSc97].

### 1.2 Communication Paradigms

After an introduction to distributed application development schemes the Inter-Process Communication (IPC) and Remote Procedure Call (RPC) communication paradigms are discussed and a classification for high- and low-level middleware is presented.

#### 1.2.1 Basic Communication Paradigms for Distributed Applications

Several basic mechanisms exist to support a developer of distributed applications.

- **The shared memory paradigm.** Processes may co-operate by accessing (reading from / writing to a shared memory. While this approach is simple to implement if a physical shared memory is actually available, it becomes difficult and potentially expensive if the physical memory available to the processes is distributed, i.e. each process only has direct access to its own physical memory. Many approaches implementing distributed shared memory have been reported in the literature (see, *e.g.*, [FMPK95], [ZSB94]). Problems to be solved are, among others, the effects of data access latency.

- **The message-passing paradigm.** In this paradigm two basic operations are offered to the user: *send* and *receive*. They send operation takes as parameter the identification of the addressed process and a reference to or the value of the message content. The receive operation delivers the reference to or the content of the message and the identification of the sender. Asynchronous or synchronous variants exist, where in the synchronous variant the sender blocks until the receiver has read the message, in contrast to the asynchronous case, in which the sender is not blocked.

  Various other variants can be found in the literature, such as unreliable and reliable message passing and point-to-multi-point or multi-point-to-multi-point operations which can be used among groups or processes.

- **The Input/Output paradigm.** Using this paradigm, processes exchange data in very much the same way as they read and write data from/to peripheral devices or files. The basic functions used are *open*, *read*, *write* and *close* or equivalent functions, where the *open* call establishes an association between the calling process and the called process, and the *read* and *write* calls are used for actually sending and receiving data. The *close* call breaks the association previously established. The format of the data exchanged may be stream-oriented or message-oriented, which often maps one-to-one on connection-oriented network services or connection-less network services, respectively. The *socket* interface initially developed for the Berkeley line of UNIX systems (see [Stevens90]) is probably the most well-known representative of this communication paradigm.
• Use of a distributed operating system which provides the basic mechanisms for the development of concurrent applications; with this approach, the programmer may take advantage of parallel programming primitives and operating system-specific system calls, granting access to communication functions. An important drawback of this scheme is a potential lack of heterogeneity allowed between participating nodes; many designs for distributed operating systems assume high-speed and low-latency channels between participating nodes which can often be provided in a local area environment only.

While implementation of the basic communication paradigms abound and have been used for years, actually using them for applications programming is not very attractive, as the operations provided are too low level. Developing distributed applications using these primitives is comparable to programming on the assembly language level, instead of using a high level programming language. Remote procedure calls, as described in the following subsection, raises the level of abstraction available to the programmer by one step.

1.2.2 Remote Procedure Calls (RPC)

The basic idea of using remote procedure calls [CDKi94] is simple: The programmer may invoke a simple procedure to actually have a remote process execute an operation. After calling the procedure, the calling process is suspended until the remote execution completes and the results of the computation (computed values and a result code) are available. In its basic form, and assuming that no errors occur, the calling process cannot distinguish the execution of a remote procedure call from local procedure call, as the call interface of the remote procedure call are identical to the local one. The basic operation of the execution of an RPC is illustrated in Figure 1. It is apparent that the RPC approach represents one specific instance of the more general client/server model, since the calling party is the client, which sends a request to the called party, the server.

While the most simple variant blocks the calling process until the call completes, a multithreaded operating system would allow for concurrent processing of multiple calls. Non-blocking approaches are called asynchronous RPC, where clients may continue processing without delaying for the RPC result. Using asynchronous RPC in an application requires a more sophisticated control of RPC in progress, as the completion of an RPC should trigger a state change in the calling process, enabling it to react to the completion of the RPC.

In any case, applications need not care about details of the data transfer between the client and the server. Many RPC packages extend their services further by providing

• an interface definition language (IDL) allowing the application programmer to specify the interface of remote procedures, and the statements to be executed on the server when a remote procedure is called,

• an IDL compiler, which compiles specifications given in IDL to procedure stubs and skeletons which can be included (as code written in a standard programming language, e.g. C) in the client’s and server’s application processes, respectively. The compiler automatically generates the marshalling and unmarshalling routines required to transform parameters and results stored in the local representation of the server or the client to a common representation or the representation used by the other side,

• and a binding service which can be called to bind a specific set of callable procedures to a client process.
In real uses of RPC, many kinds of failures may occur. Request or reply messages may be lost, clients or servers may crash at any point while a remote procedure call is in progress, and messages may be garbled or arrive in the wrong sequence. It is for these reasons that the client’s perception of a remote procedure being executed just like a local one cannot be realized in practice. The way out of this dilemma is to accept that errors may occur and be visible to the application. Thus the application programmer needs to be aware of such errors and design the logic of the application such that errors can be detected and taken care of, or RPC errors which cannot be detected will not create errors in the application. To ease the programmer’s task, four common different failure semantics have been defined. RPC implementations are expected to guarantee one of these failure semantics, whatever happens to the client, server or the communication infrastructure between them.

- **maybe semantics**: The remote operation may or may not be executed during an RPC. The RPC implementation grants just the completion of the call (i.e. the client will not hang forever), but the client will have to query the server again to find out how the operation actually was executed.

- **at-least-once semantics**: The RPC implementation guarantees that the operation is executed at least once, but possibly several times. In this case the RPC protocol specifies that a client should retry to initiate an operation until he receives a response. More than one execution may occur if the server executed a first request and then crashed before it could record that the operation was already executed. Retries of the client may then cause the same operation to be executed a second time.

- **at-most-once semantics**: The RPC implementation guarantees that the operation is executed at most once, but perhaps not at all. The RPC protocol will in this case give up without retries if a response is not received within reasonable time. The at-most-once semantics are similar to the maybe semantics, but stronger in that they guarantee that the operation will not be executed several times.

- **exactly-once semantics**: This are the semantics the programmer would wish to work with, as they emulate the semantics of a local procedure call to their best. However, it is impossible to grant these strong semantics in a general case. Exactly-once semantics may be granted for specific, application-dependent cases and normally is quite expensive to implement.

An excellent discussion of issues with RPC definition and implementation is given in Tanenbaum’s book [Tanen95].

### 1.2.3 Classification

Middleware defines the access level and the degree of transparency for distributed applications to communication functionality [PlSt97]. Besides the provision of services their specification in terms of Quality-of-Service (QoS) is required to specialize and optimize communications. Depending on the view, QoS may be quantifiable or non-quantifiable. In any case, these different views of QoS, such as user-oriented, application-dependent, or provider-defined, have to be integrated into a single comprehensive approach. Features of a flexible communication service support and detailed requirement specifications have to be provided by communication middleware allowing for easy-to-use access to these services. For that reason, an Application Programming Interface (API) independent from the communication system is important, as it represents the access point to communication middleware, handling flexible association set-ups, QoS transfers between applications and the subsystem, and an efficient user data transfer. RPC-based or object-oriented technologies may offer this API, e.g. in the form of an IDL. A particular compiler produces skeletons and stubs for shielding the developer from
communication details. According to this access point and interface two levels of communications middleware are distinguished:

1. **Low-level middleware** provides communication support, directly residing on top of services offered by the network. The main task of this level is to complement the service offered by the network such that its quality fits the needs of the application; specifically, the service provided by this level should be quantifiable in terms of QoS specifications, *e.g.*, it should accept throughput, delay, and delay jitter specifications, and – if at all possible – enforce these specifications.

   QoS guarantees can be provided only, if the entity dealing with such guarantees has the capability to schedule communication events in real-time. This implies that this entity needs to interact intimately with the operating system kernel of the end-system, i.e., the operating system scheduling of user processes and threads can be influenced by communication entities. IPC and RPC approaches are part of the low-level middleware. However, since some enhanced RPC versions offer extended semantics, sophisticated solutions already may belong to the high-level middleware.

2. **High-level middleware** directly supports the application, providing application-oriented and even application-specific services. It always incorporates a communication component, being based on a low-level middleware. Classic approaches involve the client/server communication and access methods to data bases.

   The main mission of the advanced and emerging high-level middleware part is, however, to establish an API which “speaks the language of the application”, both in terms of syntax and semantics, and to offer a variety of functionality, such as system, information, presentation, or computational services. For instance, low-level middleware may be implemented in a traditional programming style (for reasons of performance), while the application environment uses an object-oriented paradigm. In this setup, it will be the task of the high-level middleware to offer a suitable object-oriented API and required services to the application. As for semantic aspects, QoS specifications in terms of *bit/s* for throughput and *ms* for delay may not be so meaningful for the humble application programmer; thus, it will be the responsibility of the middleware designer to provide for a mapping of application-oriented QoS specifications to QoS metrics as used by the low-level middleware.

   The performance argument counts for high-level middleware as well; however, multimedia user data are often directed from the network via the low-level middleware directly to devices or vice versa, such that camera-, microphone-, or speaker-data streams are not processed by high-level middleware or the application process. Therefore, a little fraction of control data only needs to be handled in an object-oriented manner, thus resulting in less performance critical tasks to be scheduled.

   In a specific system, the high-level portion of the middleware may be tailored to the application environment, to a degree that will render such special middleware unsuitable for other application environments. Thus, the dividing line between these approaches and the application is not as clear as one may wish. Nevertheless, advanced middleware approaches offer a range of services as mentioned above to ease an application-driven selection of required services.

High- and low-level middleware tend to be described as open system platforms covering at least five functional elements [MMLa96]: (1) server behavior, (2) service definitions represented in IDLs for interface specifications, (3) user interface design, (4) QoS characteristics, and (5) communication protocols. While the former three elements are part of
the high-level middleware, the last one belongs to the low-level middleware. Obviously, QoS characteristics need attention on both levels.

1.3 Middleware Examples

Many paradigms and development schemes have been used to design and implement communication middleware. The following examples provide an overview of important approaches briefly discussing their main concepts and advantages.

1.3.1 Distributed Computing Environment – DCE

The Distributed Computing Environment DCE evolved to be an open standard within the Open Software Foundation (OSF) [OSF98]. DCE offers a high-level middleware approach, including an RPC, a directory, and a security service, to support open, distributed processing [DEC92], [Lock94].

The DCE architecture is depicted in Figure 2. Distributed applications may access each of the components separately. The Cell Directory Service offers a reliable and hierarchical association between applications and naming information to provide a location-independent server structure. The security service is based on Kerberos for authentication and based on DES (Data Encryption Standard) for encryption purposes, being interoperable with a variety of additional products and applications. A Distributed Time Service synchronizes every system clock within a single cell of clients and servers and may synchronize the system clock with an external reference time. These services are based on the DCE-RPC communication model which recommends a multithreading operating system.

Due to the mature DCE specification and various implementations, this middleware it readily available as a product for various computing platforms, also providing good performance. Current implementations include every important UNIX platform, IBM operating systems, and Microsoft Windows; all implementations are fully interoperable [OSF98].

1.3.2 Common Object Request Broker Architecture – CORBA

Due to the lack of a standard, high-level, and open architecture for distributed applications the Object Management Group (OMG) was founded in 1989. The OMG defined a model based on
object-oriented technologies including terminology definitions, an abstract object model, and a reference architecture, which provides a high-level middleware.

The Common Object Request Broker Architecture CORBA is defined as an open standard interface for distributed object computing [OMG95] including the key component of the Object Management Architecture. The component model of CORBA is depicted in Figure 3. Clients may invoke remote objects and their operations with arguments on any object being implemented according to CORBA specifications. A set of components within the architecture provides location independence of these objects which may be executed locally or remotely. The main advantages of this approach covers the transparent access to a suitable object to perform a desired operation without worrying about its location and communication issues. The Object Request Broker (ORB) Interface provides exactly this entity to de-couple application details from communication and object implementation details which are provided in the ORB core, defining the communication part of CORBA. Marshalling and conversion functions are offered to deal with argument lists. CORBA Interface Description Language (IDL) stubs and skeletons transform application object formats into low-level communication formats and vice versa and initiate an RPC-based communication. While the IDL stub is located at the client’s side, the IDL skeleton provides access to the interface method of the object to be executed. To make use of various programming environments and languages for the lower level, a CORBA IDL compiler provides automated support for this transformation. A client may use a Dynamic Invocation Interface (DII) instead, if specific IDL stubs are not available or shall not be linked into the process. This is a suitable approach for less frequently used objects or new ones to be tested. Similarly, the Dynamic Skeleton Interface (DSI) delivers requests from an ORB to the object implementation. As for the object invocation in general, the client does not need to know about the DII or the IDL stub, the request is issued identically. In addition, CORBA allows for the specification of a variety of distributed system services, such as naming, events, replications, and security [OMG95].

Current implementations of CORBA, particular the ORBs, include Orbix from Iona, Object Broker from DEC, Distributed System Object Model (DSOM) from IBM, NEO from Sun, or ORB+ from Hewlett Packard.

1.3.3 Open Distributed Processing – ODP

The reference model for Open Distributed Processing ODP [ISO10746] has been developed and standardized to overcome the lack of open models for distributed applications. However, it extends the pure architectural view of the equivalent OSI Basic Reference Model for
communications by integrating the software life cycle of distributed applications. Therefore, five viewpoints have been defined within ODP:

- **Enterprise Viewpoint**: Defines the integration of a distributed application into an enterprise, by expressing system boundaries and policies, including various tools, such as roles, activities, and organizational structures.

- **Information Viewpoint**: Models the meaning of information flows between departments or offices including the automated and manual information processing tasks, e.g., in terms of editing or transmission.

- **Computational Viewpoint**: Applications are abstractly decomposed as distributable, but communicating objects which define the required functionality and its software structure.

- **Engineering Viewpoint**: Objects are refined in terms of processes, memory, and networks to describe components and structures needed for distribution.

- **Technology Viewpoint**: Required hardware and resources are determined to lead to a detailed, conforming implementation and installation.

Aside from a highly structured reference model for these viewpoints and different levels of abstraction, functionality required to support distributed applications is not part of the ODP reference model. However, formal methods as described within Section 2. are recommended to specify functionality and behavior of a designed systems. In addition, QoS support is integrated during current activities, e.g., [ISO13236], in the ODP model.

### 1.3.4 ANSAware

ANSAware provides an implementation of an ODP-based development platform for distributed applications [Lind93]. This tool-oriented ANSA architecture automatically transforms and inserts functionality to achieve requested application features which have been programmed in a certain language. The run-time system is quite similar to the DCE one, including a fixed number of threads to serve concurrency, RPCs with “at-most-once” semantics, and an ANSA-IDL with the general possibility of specifying procedure signatures. A trader (binder) provides location transparency, but not access transparency, since RPCs have to be used in a special format to deal with exception handling. Marshalling within ANSAware can not process linked lists containing pointers, however, a wide variety of standard data types can be processed. Engineering functionality encounters besides the mentioned services an IDL and a distributed processing language which is embedded in C. Current implementations of ANSAware run on a variety of UNIX, VMS, and Windows/DOS operating system platforms.

### 1.3.5 Telecommunications Information Networking Architecture – TINA-C

The Telecommunications Information Networking Architecture Consortium TINA-C is developing an approach to provide operational independence of different platforms and to enable federation by open interfaces for an open information networking architecture [TINA98]. This architecture recognizes network operators, hardware builders, and software developers as separate roles, but defines an Distributed Processing Environment (DPE) for their cooperation in terms of locating and communicating between objects in different domains and environments. DPE hides away the distribution complexity and enables the location-transparent development of information systems. Furthermore, DPE is based on the Kernel Transport Network inside and encapsulates network signalling, which de-correlate applications from any signalling tasks. The role “user domain” determines customers utilizing a variety of
terminals for accessing information services. Information service brokers and content providers are included in the “service provider domain” role, providing brokerage services and information content suppliers. The “network provider domain” role includes operators offering transport services and interconnection amongst different domains. Utilizing object-oriented technologies, TINA-C applied developments from CORBA, ODP, ATM, and other areas, resulting in the support of multi-service networks, applying agent technology for resource negotiations independent of client’s direct interaction, and defining the DPE in details. Figure 4 depicts the TINA-C enterprise model, where software components (agents) are located in every domain and running on different hardware components (terminal, hosts). Resources support the provision of information services and TINA-C applications are defined in terms computational objects interacting via their interfaces.

Reducing the complexity of telecommunications architectures, the TINA-C object model allows for the specification of objects, interfaces, and behaviors. The Object Description Language (ODL) of TINA-C contains object descriptions including multiple optional interfaces, object behavior, object environment constraints, and traditional interface descriptions. The CORBA-IDL is utilized for these interface descriptions, however, TINA-C extensions to ODL allow for the specification of behavior and constraints. Finally, Quality-of-Service support has been integrated directly into the TINA-C object model.

1.3.6 Da CaPo++

The Da CaPo++ approach provides a framework for the support of distributed multimedia-applications [SBCC97]. It is centered around the concept of dynamically configurable communication protocols [PPVW92] and consists of low-level middleware – the Da CaPo Communication Subsystem –, an Application Programming Interface (API), and high-level
middleware including a hierarchy of application components and applications [SSRe97]. The Da CaPo++ architecture is depicted in Figure 5.

![Figure 5. Architecture of the Da CaPo++ Middleware Package](image)

The distinctive functionality of Da CaPo++ is based on the communication subsystem being capable of dynamically configuring end-to-end communication protocols such that QoS requirements specified by the application are met explicitly. The role of the API is to map application requirements from the application-specific domain to the communications domain and to build higher level communication abstractions like sessions consisting of multiple flows. A specific application module at the boundary of the API provides user and control data access to the communication protocol and multimedia devices. Local system resources, such as memory, buffer space, and processing time, and network resources, e.g., available bandwidth or delay, are considered additionally application-specified QoS to determine important properties of the run-time environment or the infrastructure access, e.g., in terms of currently supported network services such as unicast or multicast IP or ATM. The run-time system is specifically responsible for protocol processing tasks in terms of modules forming the basic element to perform required protocol processing. The best suitable protocol configuration consisting out of protocol functions, e.g., a selective retransmission function, a compression function, and a segmentation function, is determined automatically. The final result of this configuration process comprises a protocol that is optimized to fulfil needs of the application and is offered as an adapted low level middleware.

### 1.3.7 Component Object Model – COM

The Component Object Model COM provides an object infrastructure technology, defining a binary object model. It involves higher-level services, such as persistent storage, uniform data transfer, and intelligent names [Roge97]. Local objects and various technology infrastructures as well as user-oriented features, such as application integration and automation of tasks, have been included. This desktop-centric and interface-based programming model does not cover remote and networked objects. As a binary model, any interface to a uniquely identified object is defined in terms of entry points, which are enlisted as entry vectors, compared to Dynamic Link Libraries (DLL). This process eases object definition by simply listing all entry points in any language. However, it complicates networking, since location transparencies are not available due to the local validity of entry points, e.g., on a given desktop, only. The powerful inheritance capabilities of object-oriented technologies are circumvented by a special query interface to allow for the determination of each object being asked, whether it is of the particular type being looked for. Simply spoken, OLE implements COM in terms of pre-defined COM objects offering the above mentioned services, as developed by Microsoft. The
distributed version DCOM (Distributed Component Object Model) is a Microsoft proprietary architecture, suitable for Windows utilizing ActiveX components. Bridging between CORBA and ActiveX has been endorsed from Microsoft.

1.4 Evaluation and the Future

Benefits from distributed application development platforms and communication middleware have to be twofold. (1) Developers and programmers need strong guidelines to focus on the distributed application developments. (2) Instead of focussing on the technological end-system and network heterogeneity, platforms need to offer an open interface, e.g., including support of late binding, class hierarchies, replication, and migration. In general, these benefits concern two market segments, the provision of open commercial services [Cron94] and enterprise-internal rationalization processes.

Distributed object-oriented technologies offer strong advantages compared to traditional approaches. Programs are free of side effects due to object encapsulation. These objects may be adapted easily to different application domains instead of changing dependencies of heavy or light weight processes. In addition, class hierarchies and late binding approaches are suitable for open, distributed systems. Communication aspects between applications, processes, or objects reside in the background, but are supported by automated stub compilers and preprocessors to generate code for different target systems and many languages.

Concerning the discussed middleware examples above, RPC-based communication middleware offers a seamless evolution from local procedure call concepts. Additionally, standardized development and run-time systems support a broad acceptance, such as with DCE. The CORBA communication middleware provides flexibility in terms of an unlimited range of supported client/server applications. It is portable, since a variety of different implementation and language environments can be used. Additional services, such as the object location via the ORB interface, the object marshalling by IDL stubs or DII, or the object invocation are independent of the environment. A drawback of current CORBA implementations, however, is their difficulty to adapt to high-performance networks and low-latency application requirements [GoSc97]. Although the basic communication paradigm is a request/response protocol between an client and a server for DCE and CORBA, this type of traffic model may be inadequate in certain situations [BiRe94]. TINA-C is aligned to the OMG approach, in particular, ODL is a strict superset of OMG-IDL and the object model includes extensions within TINA-C that are likely to be adopted by OMG as well. While CORBA follows the networked language-based approach, COM adopted the binary object model and is limited to the desktop, even though both models tend to extend their views to local and remote concepts, respectively. DCOM encompasses the distributed version of COM, but still requires a bridge to interoperate with CORBA-based approaches. Additionally, the Da CaPo++ approach is focussed on an efficient end-system protocol support. However, distributed application support in the Internet is provided by the Hypertext Transport Protocol (HTTP) and the basic JAVA technology which utilizes the Remote Method Invocation (RMI) for communications between applets (JAVA objects).

Future developments for communication middleware need to focus on the provision of fully implemented standard platforms for traditional and mission-critical distributed computing environments, e.g., including functional-rich and interoperable languages, security concepts, Quality-of-Service support, advanced communication protocol support, a variety of operating system platforms, and efficient development tools. In particular, the integrated support of high-
speed and multimedia applications within communication middleware requires a concerted effort of platform developers, standardizing committees, and users to determine an efficient, open, and functional-appropriate solution.

2. Formal Methods

Up to now, it has been discussed how to support distributed applications using state-of-the-art middleware platforms. But are these systems correct so that the intended functional behavior is implemented? Or can these systems interoperate? In this section formal methods for the specification and validation of distributed systems are discussed. Dating back to the early days of distributed systems, when standardization of basic concepts, services and protocols started with the definition of the Open Systems Interconnection Basic Reference Model [ISO7498], work has been conducted on formal methods for the specification, validation, and testing of distributed systems, particularly protocols and their implementations. From the very beginning it has always been emphasized that open and interoperable distributed systems require complete, consistent, concise, unambiguous, and precise specifications of services and protocols [ISO8807], [BoBr87], so that the reliability of information exchange (cf. Paragraph 1.2.2) can be assessed. The work on formal methods within ISO resulted in the standardization of ESTELLE - a formal description technique based on an extended state transition model [ISO9074], [BuDe87], and LOTOS - a formal description technique based on the temporal ordering of observational behavior [ISO8807]. Concurrently, SDL - functional Specification and Description Language [ITU-Z.100], [EHSa97], evolved to be the preferred specification method within ITU. Out of these three, only SDL has found its way into practical use.

Besides the development of formal methods for the specification of communication services and protocols, a methodology and framework for conformance testing was internationally standardized [ISO9646]. Driven by practical needs, i.e., the proliferation of some of the very first OSI application services, protocols and related product implementations, it was considered important that implementations and systems could be validated for compliance to their defining standards or recommendations.

Although the developments in specification methods and conformance testing progressed independently of each other, they finally merged into a joint activity called formal methods in conformance testing. The essential idea behind this was to put the conformance testing activities on a formal basis with the advantage of having these activities also being supported by tools.

Formal methods for the specification and validation of telecommunication systems are available, are applied, and are gaining more and more importance. Standardization bodies, like ISO and ETSI - European Telecommunications Standards Institute - have committed themselves to use these methods in standardization of services and protocols, and in conformance test standards.

2.1 Specification Methods

A number of specification methods for concurrent and distributed systems have been invented. Some are presented and discussed in this section, particularly those which have found their ways into standardization. Formal methods are not solely related to OSI, but also have been of importance in, for instance, ODP (cf. Paragraph 1.3.2) and TINA-C (cf. Paragraph 1.3.5).
2.1.1 Finite State Machines

Finite State Machines (FSM) [Holz91], [Brau84] determine a simple model which is applied to specify the control structure of distributed systems. An FSM consists of a finite non-empty set of states and a set of state transitions or transitions for short. Transitions are labelled by pairs of input and output symbols. An input event triggers a state transition, if for the current state a transition is defined which is labelled with a corresponding input symbol. While performing the transition, the input is consumed and an output event is generated.

An FSM is often represented by a directed graph. Vertices correspond to states, and if there is a transition from state $s$ to $s'$ labelled by input and output symbols $i/o$ then there is a directed edge labelled $i/o$ in the graph connecting states $s$ and $s'$.

The system (or protocol) behavior, i.e., how a entity performs its functions by processing inputs and outputs, is referred to as the system’s core behavior [SaDa88]. In general, the core behavior does not cover all possible combinations of pairs of state and input, as in a given state only a subset of all inputs are valid. Nonetheless, a complete specification has to deal with these unforeseen erroneous state-input combinations. A practical approach is that for a state, all invalid inputs are ignored and null outputs are produced. The FSM remains in the same state after performing such a transition. In [SaDa88] this behavior is called non-core behavior. Note that in ISO standards and ITU recommendations the non-core behavior results in a protocol error.

In the modeling process of distributed systems, input and output symbols refer to actions that entities are performing in their communication with the environment and between themselves. These actions are requests for the execution of a service and responses to requests when a service has been executed successfully. An FSM specification of the behavior of an entity, thus, provides an understanding of the essential behavior of the entity in performing its communication tasks. However, an FSM cannot specify all aspects of the behavior of an entity in a distributed system, such as parameters of interactions with the environment or sequence numbers for numbering messages. Therefore, to overcome these problems extended finite state machines have been adopted as the basic model for ESTELLE and SDL.

2.1.2 Extended Finite State Machines - ESTELLE and SDL

Extended Finite State Machines (EFSM) belong to a model that extends FSMs in many directions. An EFSM may have local variables, inputs and output with values, transition guards, and an input may result in a number of different outputs. While performing a state transition, an EFSM may manipulate variables. ESTELLE [ISO9074] and SDL [ITU-Z.100] are two standardized formal description techniques based on the extended machine model.

The following explanations of SDL hold for ESTELLE with slight modification, too. The general model of SDL is a system that consists of a number of concurrently running EFSMs. EFSMs are referred to as processes and, since several copies of a single process may co-exist in the system, occurrences of a process are called process instances. Process instances communicate with each other and with the system’s environment. Communication is asynchronous by sending and receiving messages, called signals. All signals for a process instance are stored temporarily in an unbounded queue, called input port, until the process instance inputs the signal performing a state transition.
Execution of a process instance starts in a specifically identified state, called start state, from which the process instance proceeds performing transitions. When the process instance receives a stimulus, i.e., consumes a signal from its input port, it performs a transition possibly resulting in an output which may, as an input, have values assigned. After executing the transition the process instance enters a new state.

Input symbols indicate the variables that are assigned the values carried by the consumed signal. While performing a transition the received values may be assigned to local variables of the process instance, so that they can be used in a later state. Furthermore, a transition may have several branches. The one which is chosen for execution may be determined by the values of local variables and the values of the consumed signal.

As a further extension to FSM, ESTELLE and SDL have the ability to deal with time. In SDL, this is achieved by using timers. Timers can be started and reset, and if a timer expires it produces a time-out signal placed in the input port of the process instance that has been using the timer. These time-out signals may be consumed as regular inputs so triggering execution of a transition. Notice, it is not part of the SDL specification to relate time-out values to real-world time values. This is an implementation matter.

A system is further structured in block which are either further re-structured into blocks or into processes. Blocks and blocks and the environment are interconnected by channels. Process instances within blocks are connected by signal routes. All communication flows along signal routes and channels. Channels introduce a possibly finite delay on all signals they carry. The interprocess communication structure of processes is completely determined by the static outline of channels and signal routes.

Although ESTELLE and SDL are based on EFSMs, there exists a rather subtle difference: SDL defines a static communication structure, while in ESTELLE channels dynamically can be connected and disconnected to process instances.

### 2.1.3 Process Algebras - LOTOS

Between SDL and ESTELLE on one hand and LOTOS - Language of Temporal Ordering Specification [ISO8807] - on the other, there exists a significant difference with respect to the underlying communication paradigm: In LOTOS processes communicate synchronously. Similar to SDL, LOTOS descriptions specify the sequences of observable behavior of a system while performing communication tasks.

LOTOS uses the concepts of process, event, and behavior expression as modeling concepts. Systems and their components are represented by processes. A process hides its internal structure from the environment. But the environment can interact with a process at specific interaction points, called gates. Events are structured and consist of a gate identifier and a list of value expressions. Processes synchronize over gates and communicate by performing events provided the gate is the same and all events have matching lists of value expressions; matching value expressions are value declaration and variable declarations.

Value expressions are terms over a signature which constitutes a basic concept in the theory of algebraic specification of data types [EhMa85]. A signature consists of symbols for sorts (sets of values), constants and operations. Terms are built by applying

- (1) operations to constants, or
• (2) operations to terms without variables, and
• (3) with variables.

Given the Boolean constants true and false, operations not and or from a Boolean to a Boolean or a Boolean and a Boolean to a Boolean, respectively, and Boolean variable x then
• not(true) is an example for (1),
• not(true) or false is an example for (2), and
• not(x) is an example for (3).

Terms are interpreted in an algebra, i.e., a structure consisting of a carrier set for each sort and an operations for every constant and every operation.

The part of LOTOS dealing with the definition of behavior expression roots in process algebras, mainly Calculus of Communicating Systems [Miln80], [Miln89] and Communicating Sequential Processes [Hoar85]. A behavior expression of a process is defined from basic or elementary processes, such as stop (an inactive process) or exit (a terminating process) and operators. Operators include event prefixing, choices, parallel composition and hiding, to only mention a few. By applying event prefixing, sequential processes are built that interact with their environment performing a sequence of events. Choices can be used to build behavior expression with a branching structure, where the decision which branch to follow is determined by the first event performed. This very much relates to SDL, where several transitions may leave a state and the one is taken for which an input is available in the input port. Parallel behavior expressions may evolve independently of each other. Hiding is an operator applied to behavior expression, to make certain interactions at specific gates internal, i.e., not observable and controllable from the environment.

Since LOTOS is based on a well-established theory, it provides the basis for analysis of specifications. In particular, a hierarchy of well-defined equivalence relations has been established giving the developers of specifications the possibility for verification and the developers of implementation to refine specification into equivalent ones which include more implementation details.

2.1.4 Evolution of Formal Methods

It should be mentioned that besides the discussed formal description techniques, a number of others have been developed from which two are briefly discussed: message sequence charts [ITU-Z.120], [RGG96], [Mauw96] recently having found its way into standardization and closely related to SDL, and temporal logic which have been invented quite long ago [RGG96].

Message Sequence Charts (MSC) are a means for the graphical visualization of selected system runs of communication systems. Since 1990 MSCs evolved from an informal notation to a formal description language with complete syntax and semantics definition. Figure 6 (a) gives an example of a basic MSC. It describes the message flow between the system environment and the instances A and B. The diagram frame denotes the system environment. Instances are represented by vertical axes and messages are described by horizontal arrows. An arrow origin and the corresponding arrow head denote sending and consumption of a message. In addition to the message name, parameters may be assigned to a message. All events along an instance axis are totally ordered. The order of events on different instance axes is mediated by the message, i.e., a message must have been sent before it can be received. The rectangle
Update describes an instance action, i.e., a local activity of instance A. The hexagon State1 which is on top of instance axes A and B is a condition. It denotes state State1 which both instances have in common. The hour glass denotes the setting of a timer. The corresponding reset and time-out constructs can be found in Figure 6 (b). A cross denotes the reset of a timer and an hour glass with an arrow describes a time-out. Other basic MSC constructs are instance creation, instance termination, and the ordering of events along an instance axis, called coregion. In this example not only basic MSC constructs have been used, but in addition structural concepts like MSC references and inline expressions have been applied. An MSC reference can be seen as a placeholder for an MSC diagram (e.g., reference BasicConcepts in Figure 6 (b)). Inline expressions are used to relate MSC sections by means of operators. Altogether this supports the specification of complex system behaviors. Inline expressions are described by rectangles and operators, alt in Figure 6 (b). Different MSCs section are separated by dashed lines. In Figure 6 (b) the operator alt denotes the alternative operator which defines alternative system runs.

Already from this brief description it becomes obvious that MSCs are used with the advantage to represent graphically some of the most essential system runs being of particular interest. Notice that MSCs are not a substitute for a specification technique such as SDL or LOTOS, but should be seen as an add-on giving a glance of what happens inside a system performing communication tasks.

**Temporal Logic** is claimed to be a convenient method for specifying concurrent systems. Temporal logic extends classical logic by modal operators for describing the occurrence of events in time. In particular, temporal logic formulae describe properties a system should met. In the literature [MaPn91] two main classes of properties are distinguished: safety and liveness properties. Safety properties specify that “bad things do not happen” which means that essentially a temporal logic formula should hold in all states of a system run. Liveness properties state that “eventually something good will happen”.

![FIGURE 6. MSC Examples](image-url)
Temporal logic can be used in conjunction with the other formal description techniques mentioned. In particular, temporal logic is quite easily applicable for conveniently specifying at a high level of abstraction requirements of a system; for instance, that a request issued by a client is eventually answered by a server with a reply. Given a LOTOS specification of such a client/server system, the specification can be checked to imply this requirement (and of course others). For certain classes of LOTOS specification and temporal logic formulae this can even be checked efficiently by tools [KBGu93].

2.2 Conformance Testing

From a practical point of view conformance testing is the most likely used method to establish a certain degree of confidence that an implementation of a communication system meets its specification. Conformance testing, as understood by the OSI Conformance Testing Framework and Methodology [ISO9646], [Linn89], [Sari89], is black-box testing of the observable behavior of a distributed system. The conformance testing framework and methodology covers all aspects of testing from test suite design, test methods, test notation, and even aspects of test realization, test execution, and requirements on test laboratories.

In alignment with the OSI BRM, OSI conformance testing defines a number of abstract test methods which can be distinguished by how the test system can control and observe the system under test. Control and observation means that the test system sends or receives service primitives or more generally messages to or from the system under test. Control and observation can be close, i.e., the test system directly interacts with the system under test at its lower and upper interfaces, or can be loose, i.e., the test system interacts with the system under test from remote without direct access to the implementation.

The test system itself can be a complex distributed system comprised of several parallel running test components which communicate with each other and the system under test. The behavior of test components is given in the test case specification. Test cases are parts of a test suite; the latter comprises all test cases for a specific protocol. Each test case is specified to test a specific communication function of the system under test. A test run, i.e., the execution of a test case on a test system, yields a verdict that either confirms that the system correctly supports the function or does not support the function. If all the outcomes of all test runs show that all functions are correctly supported by the system under test, the system is claimed a conforming system.

For the description of test cases, a specific test notation - TTCN (Tree and Tabular Combined Notation) has been standardized [ISO9646-3], [PrMo92], [WaPl92], [BaGi94], [KrWa96]. TTCN test cases describe the behavior of a test system. A test case behavior description consists of statements and verdicts. Verdicts are statements on the conformance of a systems described above. Statements are grouped into statement sequences (similar to event prefixing in LOTOS) or sets of alternatives (similar to choices in LOTOS). The semantics definition of TTCN uses the concept of a snapshot. This means that whenever a set of alternatives is to be evaluated the current test system state is updated and frozen until all alternatives have been evaluated. Evaluation of alternatives is done in a fixed sequential order, and the alternative that can be executed is executed. The snapshot semantics implies that test runs are deterministic and thus reproducible. The latter is particularly important, as after changes of the system under test have been done, every test run should be repeated and should produce a unique verdict.
Although defined for testing OSI compliant systems, [ISO9646] has been and is applied in other areas as well, e.g. testing of ATM signalling [ScRe96] and testing of CORBA supported systems [BaGe97]. Whereas the applications of conformance testing mentioned are related to testing functional properties of distributed systems, OSI conformance testing concepts also have been applied in performance testing of ATM, HTTP [SSRe97] and real-time system testing [Tret92]. Since a few years, also Quality-of-Service testing has become an issue [MRBu93], [GrWa95] which combines testing functional properties and testing timing constraints.

2.3 Formal Methods in Conformance Testing

Although research into formal methods and conformance testing has started separately and for quite a number of years evolved independently of each other, in the late ‘80 ITU and ISO set up a working group on Formal Methods in Conformance Testing [FMCT96]. The group focuses on a definition of conformance and conforming implementation in the context of the standardized formal description techniques, thus, bridging the gap between system specifications and testing.

Fundamental to the development on formal methods in conformance testing has been the invention of implementation relations [BiRe94]. Given a formal specification of a system and given an implementation of the system, if both are in the implementation relation then the system is a correct implementation of the specification. However, different implementation relations are obvious but not all of them are an intuitive formalization of conformance. [BiRe94] has developed a proposal of a conformance relation and its associated testing theory. The idea behind the conformance relation is that an implementation does not deadlock unexpectedly after having performed a sequence of interactions which is also possible for the specification. An implementation unexpectedly deadlocks, if no further action is possible for the implementation although accordingly to the specification some action should be possible. Since only specified sequences of interactions are used during testing, conformance testing clearly does not assess robustness of an implementation.

Furthermore, [BiRe94] shows that for the conformance relation a tester can be (automatically) generated from the specification which has the following properties; Firstly, if specification and implementation are in the conformance relation, the implementation passes all tests defined by the tester. Secondly, the tester is exhaustive with respect to the conformance relation. Exhaustive in general implies that an infinite number of tests are defined which all have to be executed against the implementation. However, for practical reasons and in particular due to limited resources, only a subset of all test cases can be executed. It is still a research topic how this subset should be determined. To achieve a solution, test coverage metrics have been used assessing the “quality” of a test due to several criteria, such as cost and test distances between test sequences. These criteria attempt to value the gain in a test sequence compared to others [BALL89], [Tret92], [CVZh97].

Although this discussion only covers only parts of the overall theory of formal methods in conformance testing, the developed theory is useful in solving practical problems. Even more, the work on formal methods has proven that a smooth, but well-founded and validated transition from specifications to conformance tests is feasible. Thus, automatic test generation and implementation will be the next action items on the research agenda.
2.4 Formal Methods - And What Next?

The above discussion surveys almost two decades of research and developments. It is quite opportune to ask whether all work has gained practical impact. [Rudi92], [Holz94], [Dahb95] report on quite a number of “success stories”: a rigorous approach to protocol and distributed system specification helps in identifying errors in all phases of system engineering. However, it still needs to be proven that over the long term applying formal methods also prove economically valuable in terms of time-to-market and return on investment.

At least a first attempt has been started towards rational use of formal methods. ETSI, ISO and ITU decided to produce standards and recommendations that also contain a formal description of the protocol specification [ITU-X.762], [MAC97].

As formal methods are out and are used, new and upcoming advanced distributed applications, however, pose new requirements on the capabilities of formal methods. Whereas SDL, LOTOS and TTCN focus on specifying and testing functional or input/output properties, advanced systems like multimedia systems require methods and techniques to specify and to test non-functional properties like performance, Quality-of-Service, robustness, and hard real-time requirements, too. Hard real-time requirements are found in safety-critical systems, like power plant and process control systems. Those systems behave correctly if they correctly implement their functional behavior and if the results computed are available within some hard time limits.

The good news is that all of the formal methods discussed above and a number of existing others have been extended to enable specification and testing of non-functional properties. Unfortunately, no generally applicable approach has been found yet.

In addition to the cited literature the following material is a good source for further information: [Turn93] for a detailed introduction to formal description techniques; [MaPn91], [MaPn95] for an essay on temporal logic of reactive systems, their specification and verification; and [Holz91] is a book on a pragmatic approach to the design and validation of protocols which is also supported by tools.

Last but not least, for the discussed formal methods and parts of the design process of distributed systems, tools are available on the market: editors, validators, compilers and test generation tools.

3. References


