The Broker Architectural Framework

Michael Stal

Siemens AG, Corporate Research and Development
Otto-Hahn-Ring 6
D-81730 München
phone: +49 89 636 49380
fax: +49 89 636 40757
e-mail: Michael.Stal@zfe.siemens.de

Abstract

According to Andrew S. Tanenbaum distributed systems have a significant drawback: “Distributed systems need radically different software than do centralized systems”. This is the major technical reason for consortias such as the Object Management Group (OMG) or the Open Software Foundation (OSF) and companies such as Microsoft developing their own technologies for distributed computing.

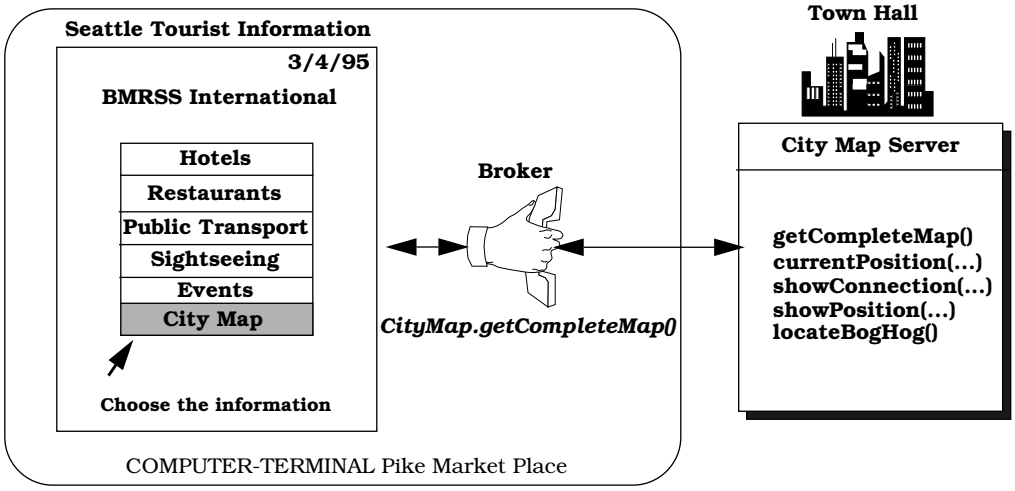
Platforms such as Microsoft OLE 2.x (OLE = Object Linking and Embedding), OSF DCE (Distributed Computing environment) and OMG CORBA (Common Object Request Broker Architecture) share the use of a common software architecture from which we have abstracted the Broker architectural framework. There are three groups of developers who will benefit from our description of the Broker pattern:

- Developers working with an existing Broker system who are interested in understanding the architecture of such systems,
- Developers who want to implement lean versions of a Broker system, without all the bells and whistles of a full-blown OLE or CORBA,
- Developers who plan to implement a full-fledged Broker system and need an in-depth description of the Broker architecture.

Broker

The *Broker* architectural framework may be applied to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication (such as forwarding requests), as well as transmitting results and exceptions.

Example Suppose, you are developing a city information system designed using a wide area network. Some computers in the network host one or more services that maintain information on events, restaurants, hotels, historical monuments, or public transportation. Computer-terminals are connected to the network. Throughout the city tourists can retrieve all the information they are interested in from these terminals using a WWW (World Wide Web) browser (such as Mosaic or Netscape). The data is not all maintained in the terminals but distributed across the network. Users interact with front-end software running on each terminal. This software supports on-line retrieve of requested information from the appropriate servers and the display of the data on the screen.



Since we expect the system to change and grow continuously, services should be decoupled from each other. In addition, the terminal software should be able to access services without knowing their location. This allows us to move, replicate, or migrate services. ❑

Context Your environment is a possibly heterogeneous, distributed sytem with independent cooperating components.

Problem Composing a complex software system as a set of decoupled and interoperating components rather than as a monolithic application results in better flexibility, maintainability and changeability. By partitioning functionality into independent components the system becomes potentially distributable and scalable. However, when distributed components must communicate with other components, some means of interprocess communication is required. Services for adding, removing, exchanging, activating, or locating components are also needed. When applications use these services they should not depend on system-specific details to guarantee

portability and distribution even within a heterogeneous network. Hence, use the Broker architecture when you need to balance the following forces:

- Components should be able to access services of other components through remote, location-transparent, service invocations.
- It must be possible to exchange, add, or remove components at runtime.

Solution When two independent components - a client and a server - need to interoperate, they have to communicate with each other. If they handle communication themselves, the resulting software system faces several dependencies and limitations. For instance, the system becomes dependent on the communication mechanism used, clients need to know the location of servers, and in many cases the solution is limited to only one programming language. To achieve a better decoupling of clients and servers, a broker component is introduced as an additional layer. Servers register themselves with the broker and make their services available to clients through method interfaces. Clients access the functionality of servers by sending requests via the broker component. The tasks of a broker include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

Using the Broker pattern an application can access distributed services simply by sending message calls to the appropriate object instead of focussing on low-level interprocess communication. All system specific tasks such as forwarding messages or locating objects are handled by a broker. From a developer's view, there is essentially no difference between developing software for centralized systems and developing for distributed ones. The user of an object only sees the interface the object offers. He does not need to know anything about implementation details of an object or its physical location. In addition, the Broker architecture is flexible in that it allows dynamic change, addition, deletion, and relocation of objects.

The Broker pattern reduces the complexity of developing distributed applications, because it makes distribution transparent to the developer. It achieves this goal by introducing an object model where distributed services are encapsulated within objects. Thus, Broker systems offer the integration of two core technologies: distribution and object technology. Moreover, they extend object models from single applications to distributed applications consisting of decoupled components that can run on heterogeneous machines and may be written in different programming languages.

Structure The *Broker* architectural framework defines six kinds of participating components: *clients*, *servers*, *brokers*, *bridges*, *client-side proxies* and *server-side proxie*s.

- A *server*[1] implements objects that expose their functionality through interfaces consisting of operations and attributes. These interfaces are either made available through an interface definition language or a binary standard (see the implementation section for a comparison of these approaches). Interfaces typically group semantically related functionality. There are two kinds of servers: servers

1. In this pattern description *servers* are responsible for implementing services. In an object-oriented approach every service is realized by one or more *objects*. Whenever the term *server object* is used, we want to emphasize the fact that this server appears to other components as an object in the object-oriented sense.

that implement services for use by many application domains and servers offering functionality for a single specific application domain or task.

- By *clients* we mean applications that access the services of at least one server. To call remote services, clients forward requests to the local broker. After execution of an operation they receive responses or exceptions from their broker.

In the Broker architectural framework the interaction between clients and servers is based upon a dynamic model, which means that servers may also act as clients. This dynamic interaction model differs from the traditional notion of Client-Server Computing in that the roles of clients and servers are not statically defined. From the viewpoint of an implementation, clients could be considered as applications and servers as libraries. Note, that clients do not need to know the location of the servers they access. This is important, because it allows the addition of new services and the movement of existing services to other locations, even at runtime.

Class Client	***Collaborators*** Client-side Proxy, Broker
Responsibility • Implements user functionality • Sends requests to servers through a client-side proxy	

Class Server	***Collaborators*** Server-side Proxy, Broker
Responsibility • Implements services • Registers itself with the local broker • Sends responses and exceptions back to the client through a server-side proxy	

- A *broker* is a messenger responsible for the transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client.

  A broker must have some means of locating the receiver of a request from its system-unique identifier.

  Furthermore, the APIs (Application Programming Interface) a broker offers to clients and servers include operations for registering servers and invoking requests.

  When a request arrives for a server that is maintained by the local broker, the broker passes the request directly to the server. If the server is currently inactive, the broker has to activate the server. All responses and exceptions from a service execution are forwarded by the broker to the client that sent the request. If the specified server is hosted by another broker, the local broker must find a route to this broker and must forward the request along this route. Hence, there is a need for brokers to interoperate with each other.

Depending on the requirements of the whole software system, additional services - such as name services[2] or marshalling support[3] - may be integrated into the broker.

Class Broker	***Collaborators*** Client, Server, Client-side Proxy, Server-side Proxy, Bridge
Responsibility • (Un-)registers servers • Offers APIs • Transfers messages • Error recovery • Interoperates with other brokers through bridges	

- *Client-side proxies* represent a layer between clients and the broker component. This additional layer provides transparency in that a non-local object appears to the client as a local one. In detail, the proxies allow the hiding of implementation details from the clients. For example: the interprocess communication mechanism used for message transfers between clients and brokers, the creation and deletion of memory blocks, and the marshallling of parameters and results. In many cases, client-side proxies translate the object model specified as part of the Broker architectural framework to the object model of the programming language used to implement the client.
- *Server-side proxies* are generally analogous to Client-side proxies. The difference is that they are responsible for receiving requests, unpacking incoming messages, unmarshalling the parameters, and calling the appropriate service.

Class Client-side Proxy	***Collaborators*** Client, Broker
Responsibility • Interoperates with the local broker • Encapsulates system-specific functionality • Mediates between the client and the broker	

Class Server-side Proxy	***Collaborators*** Server, Broker
Responsibility • Interoperates with the local broker • Calls services within the server • Encapsulates system-specific functionality • Mediates between the server and the broker	

2. Name services provide associations between names and objects. To resolve a name, a name service determines a server associated with the given name. In the context of Broker systems, names are only meaningful relative to a name space.
3. Marshalling is the semantic-invariant conversion of data into a machine-independent format such as ASN.1 (Abstract Syntax Notation) or ONC XDR (eXternal Data Representation). Unmarshalling performs the opposite transformation.

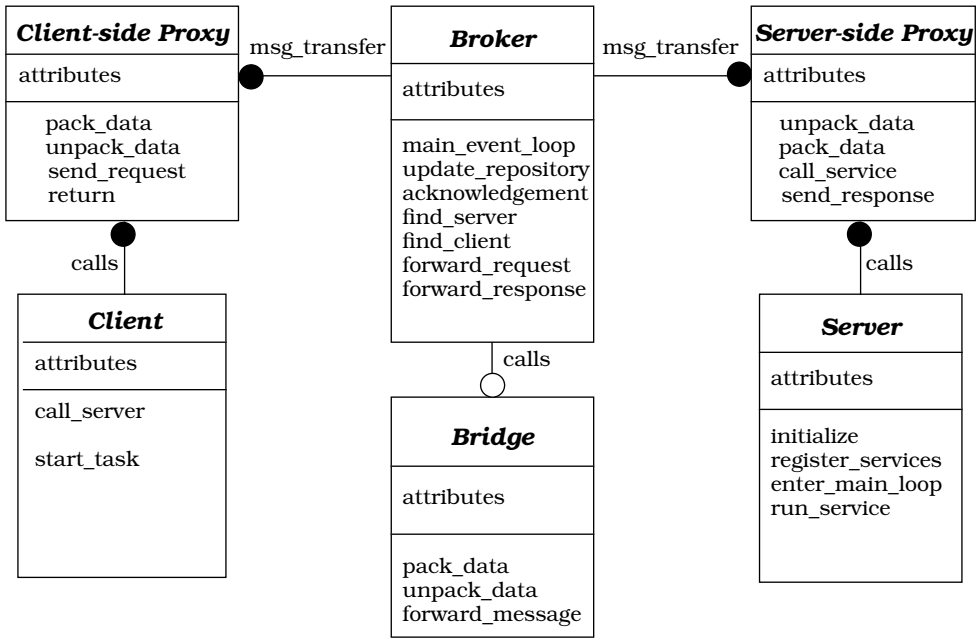
- *Bridges*[4] are optional components for hiding implementation details when two different brokers interoperate. Suppose, a broker system executes on a heterogeneous network system. If requests must be transmitted over the network, different brokers have to communicate independent of the different network and operating systems in use. A bridge builds a layer that encapsulates all these system-specific details.

Class Bridge	***Collaborators*** Broker, Bridge
Responsibility • Interoperates with remote brokers • Encapsulates network-specific functionality • Mediates between the local broker and the bridge of a remote broker	

There are two different kinds of Broker systems: Broker systems using direct communication and those using indirect communication. For better performance some broker implementations only establish the initial communication link between a client and a server, while the rest of the communication is done directly between the participating components - messages, exceptions and responses are transferred between client-side proxies and server-side proxies without using the broker as an intermediate layer. This direct communication approach requires servers and clients to use and understand the same protocol. In this pattern description we are focussing on the indirect broker variant, where all messages are passed through the broker. However, we will include a description of the direct communication variant additionally whenever appropriate.

4. We call these components *Bridges* following the terminology of the OMG in the CORBA2-specification.

The following OMT-diagram presents the objects involved in a Broker system::

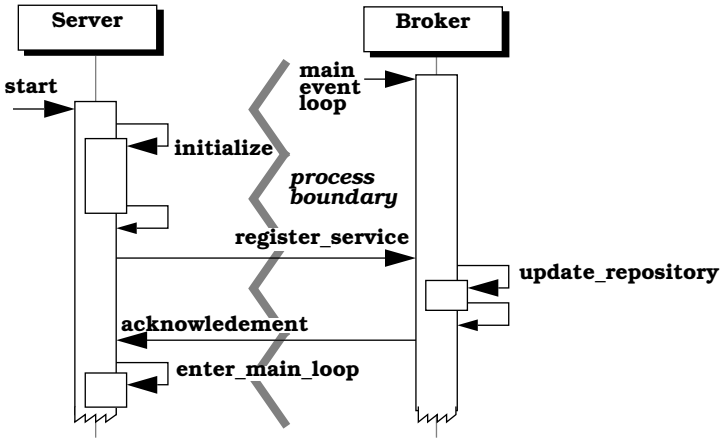


Dynamics Part I:

The first scenario illustrates the behaviour of a Broker architecture when a server registers itself with the local broker component:

- Something starts the broker component. The broker enters its event loop and waits for incoming messages.
- The user starts the server application. First, the server executes some initialization code. After initialization is complete, it registers with the broker.
- The broker receives the incoming registration request from the server. It extracts all necessary information from the message and stores it into one or more repositories. These repositories are used to locate and activate servers. Then, the broker sends an acknowledgement to the server.
- After receiving the acknowledgement from the broker, the server enters its main loop waiting for incoming client requests.

.

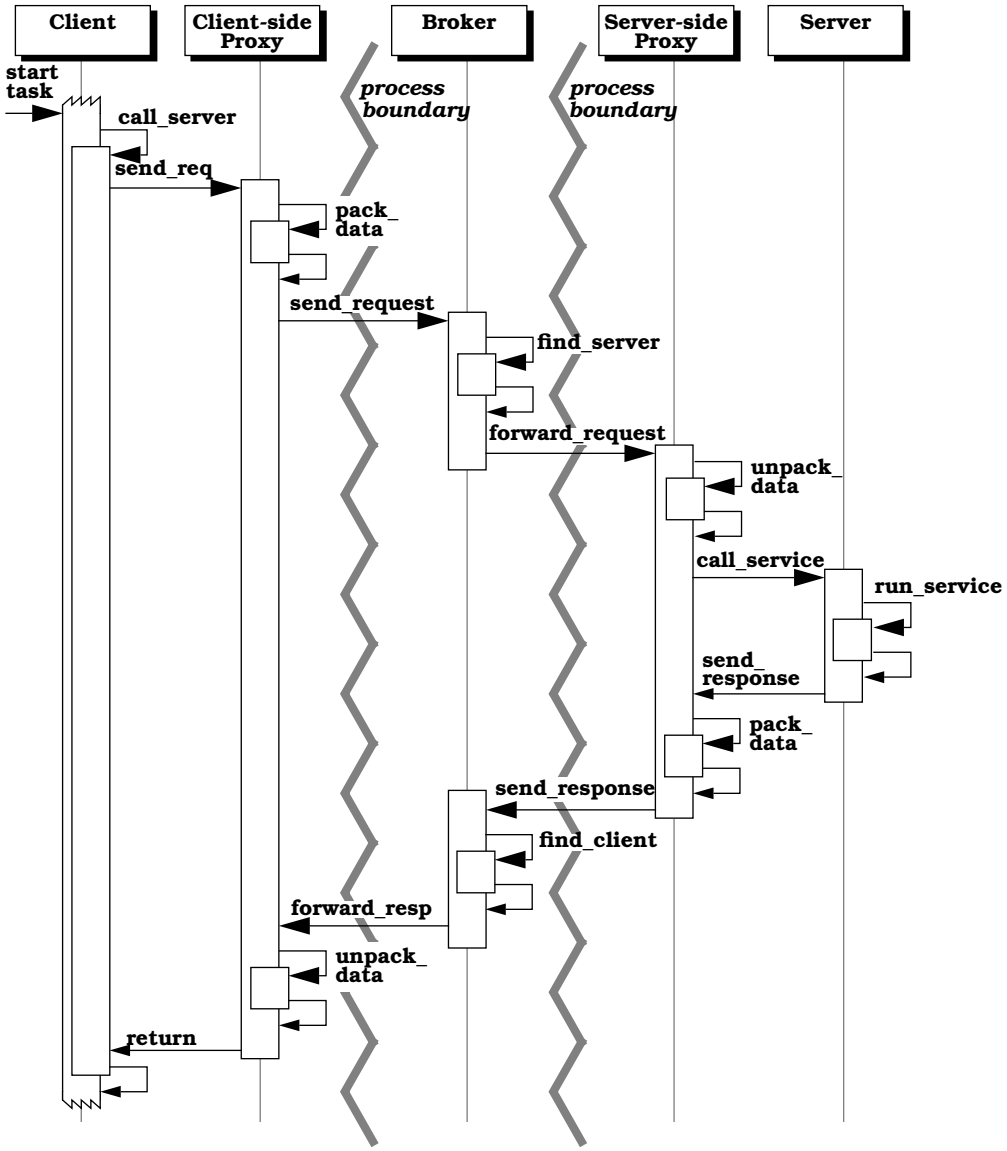


Part II:

A second scenario illustrates the behaviour of a Broker architecture when a client sends a request to a local server:

- The client application is started. During program execution the client invokes a method on a remote service/object.
- The client-side proxy packages all parameters and other relevant information into a message and forwards this message to the local broker component.
- The broker looks up the location of the server in its repositories. Since the server is available locally (*for the remote case see the following scenario*), the broker forwards the message to the corresponding server-side proxy.
- The server-side proxy unpacks all parameters and other information such as the method it is expected to call. Then, it calls the appropriate service.
- After the service execution is complete, the result is returned to the server-side proxy, which packages the result and other relevant information into a message and passes it to the broker.
- The broker forwards the response to the client-side proxy.

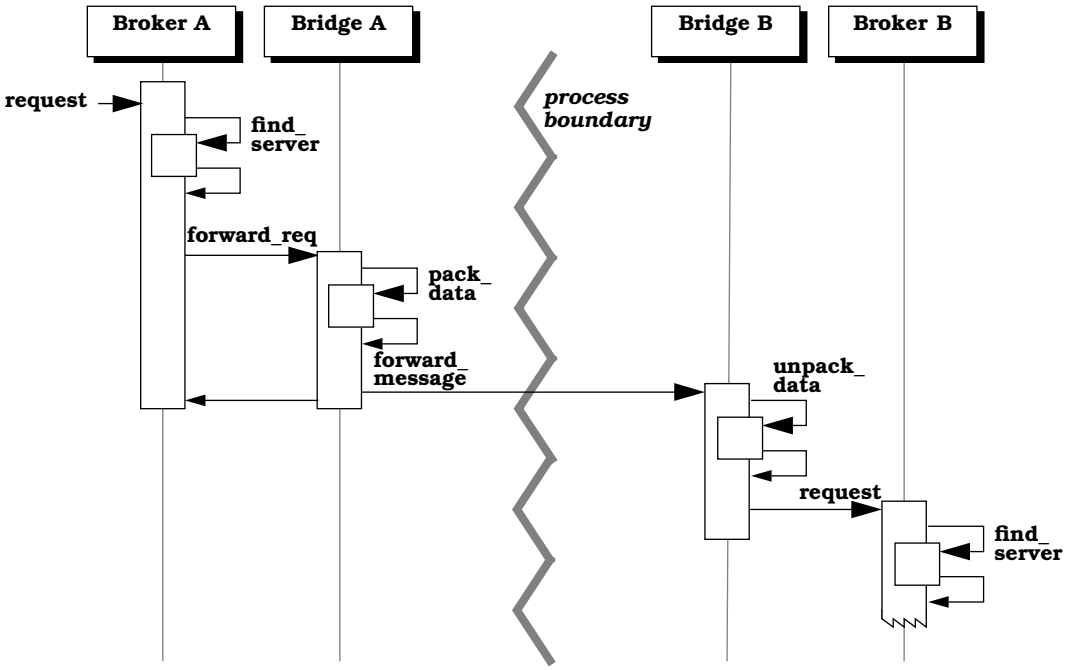
- When the client-side proxy receives the response, it unpacks the result and returns. Finally, the client process continues with its computation.



Part III:

The third scenario presents the interaction of different brokers with the help of bridge components:

- Broker A receives an incoming request. First, it locates the server responsible for executing the specified service by looking it up in its repositories. Since the corresponding server is available on another network node, the broker has to forward the request to a remote broker.
- The message is passed from Broker A to Bridge A, which is responsible for converting the message from the protocol defined by Broker A to a network specific but common protocol that the two participating bridges understand. After message conversion, Bridge A transmits the message to Bridge B.
- Bridge B maps the incoming request from the network specific format to a Broker B specific format.
- Broker B performs all the actions necessary when a request arrives (see the first step in this scenario).



Implementation The following steps are necessary to create a Broker architectural framework:

1 First, define an object model or use an existing one. The selection of an object model has a major impact on all other parts of the system under development. Each object model must specify entities such as object names, objects, requests, values, exceptions, supported types, type extensions, interfaces, operations. In this first step you should only consider semantic issues. If the object model has to be extensible, prepare the system for future enhancements. For instance, specify a basic object model and how it can be refined systematically using extensions.

Another key issue in designing an object model is to describe the underlying computational model. This model must provide mechanisms for executing requests. That is, definitions of the state of server objects, definitions of methods, how methods

are selected for execution, how server objects are generated and destroyed, etc. Keep in mind that neither the state of server objects nor their method implementations should be directly accessible to clients. Clients may only change or read the server's state indirectly by passing requests to the local broker. With this separation of interfaces and server implementations the so-called remoting of interfaces becomes possible: clients use proxy server interfaces that are completely decoupled from the server implementations, as well as from the concrete implementations of the server interfaces.

2 In the next step, decide which kind of component-interoperability the system should offer. You can design for interoperability either by specifying a binary standard or by introducing a high-level interface definition language (IDL). An IDL-file contains a description of the interfaces a server offers to its clients. The binary approach needs support for the binary representations from your programming language system. For instance, binary method tables are available in Microsoft OLE. These tables consist of pointers to method implementations and enable clients to call methods indirectly by using pointers. The access to OLE-objects is only supported by compilers or interpreters that know the physical structure of these tables. In contrast to the binary approach, the IDL-approach is more flexible in that an IDL-mapping may be implemented for any programming language of your choice. Sometimes, both approaches are used in combination as in IBM SOM (System Object Model).

   An IDL-compiler uses an IDL-file as input and generates programming language code or binary code as output. One part of this generated code is required by the server for communicating with its local broker, another part of the output is used by the client for communicating with its local broker. Moreover, the broker may use the IDL-specification to maintain type information about existing server implementations. Whenever interoperability is provided as a binary standard, every semantic concept of the object model must be associated with a binary representation. However, if you supply an interface definition language for interoperability, you must map the semantic concepts to programming language representations. In either case all data types that services may use must be mapped to appropriate representations.

   One question remains: When should a Broker system expose interfaces with an interface definition language and when by a binary standard instead? The reason for the first approach is to gain more flexibility with respect to the broker implementation: every implementation of the Broker archictecture may define its own protocol for the interaction between the broker and other components. Here, it is the task of the interface definition language to provide a mapping onto the local broker protocol. When following a binary approach, you need to define binary representations such as method tables for invoking remote services. This often leads to greater efficiency, but requires all brokers to implement the same kind of protocol for communicating with clients and servers.

3 Specify the application programming interfaces (APIs) the broker component provides for clients and servers. On the client-side, functionality must be available for constructing requests, passing them to the broker and receiving responses. In this context, decide whether clients should be only able to invoke server operations statically, thus allowing clients to bind the invocations at compile-time. If you want to allow dynamic invocations of servers, this decision leads to a direct impact on the size or number of APIs. For instance, you need some functionality for asking the

broker about existing server objects. This may be implemented with the help of a meta-level schema.

You will have to offer some operations to clients, so that they are capable of constructing requests at run-time. The server implementations use API-functions primarily for registering with the broker. Brokers use repositories to maintain the information. These repositories may be available as external files, so that servers can register themselves even before system start-up. Another approach is to implement the repository as an internal part of the broker component, e.g., using the Repository variant of the Blackboard architectural framework. Using this approach, the broker must offer an API allowing servers to register at run-time. Here, the broker component is responsible for associating server object identifiers with server object implementations. Thus, the server-side API of the broker must generate system-unique identifiers. If clients, servers and the broker are running as distinct processes, the API functions need to be based upon an efficient mechanism for interprocess communication between clients, servers and their local broker.

4 Use proxy objects to hide implementation details from clients and servers. On the client side, a local proxy object represents the remote server object called by the client. The same holds for the server-side, where a proxy is used for playing the role of the client. Proxy objects have the following responsibilities:

- Client-side proxies package procedure calls into messages and forward these messages to the local broker component. In addition, they receive responses and exceptions from the local broker and pass them to the calling client. For this purpose, you must specify an internal message protocol for the communication between proxy and broker.
- Server-side proxies receive requests from the local broker and call the methods in the interface implementation of the corresponding server. Moreover, they forward server responses and exceptions to the local broker after packaging them according to an internal message protocol.

Note, that proxies are always part of the client or server process.

Proxies hide implementation details by using an internal interprocess communication mechanism for communicating with the broker component. They may also implement the marshalling and unmarshalling of parameters and results into/from a system-independent format.

If you are following the IDL-approach for interoperability, proxy objects are automatically available, because they can be generated by an IDL-compiler. On the other hand, if you are using a binary approach, the creation and deletion of proxy objects may happen dynamically.

5 Design the broker component intertwined with the preceeding implementation steps 3 and 4. In the following we describe how to develop a broker component that acts as a messenger for every message passed from a client to a server, and vice versa. Some implementations do not transmit messages via the broker so as to increase the performance of the whole system. In these systems most of the work is done by the proxies, while the broker is still responsible for establishing the initial communication link between clients and servers. A direct communication between client and server is only possible when both of them can use the same protocol. We denote such systems as Direct Communication Broker systems (*see Variants section*).

In specifying the broker, you should systematically iterate through the following steps:

- Specify a detailed on-the-wire protocol for interacting with client-side proxies and server-side proxies. Plan the mapping of requests, responses, and exceptions to your internal message protocol. The internal message protocol handles the mapping of higher-level structures such as parameter values, method names, return values to corresponding structures specified by the underlying interprocess communication mechanism.
- A local broker must be available for every machine in the network where there are clients or servers. If requests, responses or exceptions are transferred from one network node to another, the corresponding local brokers have to communicate with each other using an on-the-wire protocol. Bridges are used to hide details such as network protocols and operating system specifics from the broker. A bridge is a special component that encapsulates the mechanisms for (network) communication between brokers. That is, the message transfer between brokers is based upon the communication between the corresponding bridges. The broker must also maintain a repository to locate the remote brokers or gateways to which it forwards messages. You may encode the routing information for finding remote brokers as a part of the server or client identifier. Broadcast communication is another (potentially inefficient) way to locate the network node where a server or client resides. If the proxies (see step 4) do not provide mechanisms for marshalling and unmarshalling parameters and results, the broker component must include that functionality.
- In the case your system supports asynchronous communication between clients and servers, you will need to provide message buffers within the broker or within the proxies for the temporary storage of messages.
- The broker must contain a directory service for associating local server identifiers with the physical location of the corresponding servers. For instance, if the underlying interprocess communication protocol is based upon TCP/IP, an Internet port number could be used as the physical server location.
- When your architecture requires system-unique identifiers to be dynamically generated during server registration, the broker must offer a name service for instantiating such names.
- If your system supports dynamic method invocation (see step 3), the broker needs some means for maintaining type information about existing servers. A client may access this information using the broker APIs in order to construct a request dynamically.
- Consider the case when something fails. In a distributed system two levels of errors may occur. First, a component such as a server may run into an error condition. This is the same kind of synchronous error you encounter when executing conventional non-distributed applications. Second, the intercommunication between two independent processes may fail. Here the situation is more complicated since the communicating components are running asynchronously. Hence, plan the broker's actions when the communication with clients, other brokers or servers breaks. For instance, some brokers resend a request or response several times until they succeed. If an at-most-once semantic[5] is used, you have to

make sure that a request is only executed once even if it is resent. Do not forget the case where a client tries to access a server which either does not exist or which the client is not allowed to access. Error handling is an important topic when implementing a distributed system. If you forget to handle errors in a systematic way, testing and debugging of client-applications and servers will become an extremely tedious job.

6 Whenever you are implementing interoperability by providing an interface definition language, you need to build an IDL-compiler for every programming language you support. An IDL-compiler translates the server interface definitions to programming language code. When many programming languages are in use, it is recommended that the compiler is developed as a framework allowing the developer to add his own code generators.

Example The following applies the Broker architectural framework to implementing the information system we introduced in the motivating example:

- The software consists of WWW-services that are distributed over the network, as well as applications, i.e., clients, running on graphics terminals. Users interacts with these applications to retrieve the information they need, a complete list of available hotels, for example.
- On every network node a broker component is responsible for forwarding requests from clients to the appropriate server and results from the server back to the client. All servers are identified by system-unique names such as "ftp://munich/Hotel_Information_Service". Clients do not need to know anything about the location of the services they access. It is the task of the brokers to locate servers using location-databases and forward requests to these servers. The database in our example comprises a mapping between service names and the physical network address where the appropriate server resides. In the system, a service may be provided by more than one server in order to improve its availability. If one service in the network is not accessible, the broker may forward the request to another machine providing the same kind of service (see the *Trader System* variant). For this purpose, a Trading service is available in the broker system.
- Bridges are used as layers between the participating brokers. This is why the whole system is capable of running on a heterogeneous network consisting of PCs and workstations using different broker implementations. We use cheap PCs running under MS Windows 95 for visualizing the information and fast Unix-workstations for providing the database services.

❑

Variants There are several possible variants of the Broker architectural framework. The following variants can be combined:

- *Direct Communication Broker System*: For efficiency reasons you may sometimes choose to relax the restriction that clients can only forward requests through the local broker. Instead, clients send their requests directly to the remote broker where the server resides. Another possibility is allowing clients communicate with

---

5. When supporting at-most-once semantics your system has to guarantee that any request either fails or is executed only once. If you implement other *semantics* instead such as "at least once" or "exactly once", the same request may be resent and executed several times. This strategy is only applicable to idempotent services, where overall consistency is not damaged by executing a service more than once. A typical example for an idempotent service is a function that assigns an initial value to a variable.

servers directly. Here, the broker tells the clients which communication channel the server provides. Then, the client can establish a direct link to the requested server. In these systems, the proxies and not the broker are responsible for handling most of the communication activities.

- *Message Passing Broker System*: This variant is suitable for systems that focus on the transmission of data instead of implementing a Remote Procedure Call abstraction[6]. Using this variant, servers do not offer services that clients can invoke, but use the type of message to determine what they must do. In this context, a message is a sequence of raw data together with additional information that specifies the type of a message, its structure and other relevant attributes.
- *Trader System*: Usually, a client request is forwarded to exactly one uniquely-identified server. In some circumstances, the broker should can which server(s) can provide the service and forward the request to an appropriate server. When you implement this variant, services and not servers are the targets to which clients send their requests. Hence, client-side proxies use service identifiers instead of server identifiers to access server functionality. Moreover, the same request might be forwarded to more than one server implementing the same service.
- *Adapter Broker System*: To enhance flexibility, the interface of the broker component to the servers can be hidden by using an additional layer. This adapter layer is a part of the broker and is responsible for registering servers and interacting with servers. By supplying more than one adapter, you can support different server implementation strategies with respect to server granularity and server location. For instance, if all objects that an application accesses are located on the same machine and are implemented as library objects, a special adapter could be used to link the objects directly to the application. Another example is the use of an object-oriented database for maintaining objects. Since the database is responsible for providing methods and storing objects, there may be no need for registering objects explicitly. In such a scenario, we could provide a special database adapter.
- *Callback Broker System*: Instead of implementing an active communication model where clients are the producers of requests and servers are the consumers of requests, you may also use a reactive model - the reactive model is event driven, and there is no distinction between clients and servers. Whenever an event arrives, the broker invokes a callback method of the corresponding component. As a result, events may happen that the broker transmits to other components that are interested in them.

Selected known uses

- The Broker architectural framework was used to specify the Common Object Request Broker Architecture defined by the Object Management Group. CORBA is an object-oriented technology for distributing objects on heterogeneous systems. An interface definition language IDL is available to support the interoperability of clients and server objects [OMG92]. Many CORBA implementations realize the *Direct Communication Broker System* variant.

---

6. Brokers offering RPC (Remote Procedure Call) interfaces are typically built using message passing interfaces.

- IBM SOM/DSOM (System Object Model, Distributed System Object Model) represents a CORBA-compliant Broker system. In contrast to many other CORBA-implementations, it implements interoperability by combining CORBA-IDL with a binary protocol. SOM's binary approach supports of subclasses from existing binary parent classes.
- Microsoft's OLE 2.x technology (OLE stands for Object Linking & Embedding) provides another example for the use of the Broker architectural framework. While CORBA guarantees interoperability using an interface definition language, OLE 2.x defines a binary standard for exposing and accessing server interfaces [Brockschmidt94].
- In a Siemens in-house project for building a telecommunication switching system based upon ATM (Asynchronous Transfer Mode), the Broker architectural framework has been successfully applied [ATM93]. This system uses the *Message Passing Broker System* variant.

Consequences The Broker architectural framework has some important benefits:

Changeability and extensibility of components: The Broker architectural framework offers a high degree of changeability and extensibility. When servers change but their interfaces remain the same, this will have no functional impact on clients. Modifying the internal implementation of the broker but not the APIs it provides, has no effects on clients and servers except for performance changes. Changes in the communication mechanisms used for the interaction between servers and the broker, between clients and the broker, and between brokers will eventually require you to recompile clients, servers or brokers. However, you will not need to touch the source code. The use of proxies and bridges is an important reason for this ease of implementing changeability.

Portability of a Broker system: The Broker system hides operating system and network system details from clients and servers by the use of indirection layers such as APIs, proxies and bridges.

Therefore, in most cases it is sufficient to port the broker component and its APIs to a new platform and to recompile clients and servers. Structuring the broker component into layers is recommended, e.g., according to the Layered architectural framework. If the lowermost layers hide system specific details from the rest of the broker. you will only need to port these lowermost layers instead of completely porting the broker component.

Interoperability between different broker systems: Different broker systems may interoperate if they understand a common protocol for exchanging messages. This protocol is implemented and handled by bridges, which are responsible for translating the broker specific protocol into the common protocol, and vice versa.

Reusability: In a Broker architecture software-reuse is supported. Whenever you build new client applications you can base the functionality of your application upon existing services. Thus, you may reduce the code of the new client application by reusing services that are already available. For example, suppose you are going to develop a new business application. When components that offer services such as text editing, visualization, printing, database access, or spreadsheets are already available, you will not need to implement these services yourself. Instead, it may be enough to integrate these existing services into your software.

Location Transparency: Since the broker is responsible for locating servers by an unique identifier, clients do not need to know where the servers are located. On the other hand, servers do not care about the location of calling clients, since they receive all requests from the local broker component. Hence, the Broker architectural framework provides location transparency.

The Broker architectural framework has the following drawbacks:

Restricted efficiency: Applications using the Broker architectural framework are usually slower than applications whose component-distribution is statically known. Moreover, systems that directly depend on a concrete mechanism for interprocess communication gain better performance than a Broker architecture that introduces indirection layers in order to be portable, flexible and changeable.

Less fault tolerance: Compared with a non-distributed software system, a Broker system may offer less fault tolerance. Suppose, that during program execution either a server or a broker fails. In this case, all applications that depend on the server or broker will be unable to continue successfully. However, replication can be used to increase reliability.

The following is another aspect that should be considered:

Testing and Debugging: A client application developed using tested services is more robust and easier to test. On the other hand, debugging and testing a broker system is a tedious job because of the many components involved. For instance, if the cooperation between a client and a server fails, there are two possible reasons for the failure: Either the server itself has entered an error state or there is a problem somewhere on the communication path between client and server.

See also *Forwarder-Receiver design pattern*: This pattern provides transparent interprocess communication for software systems with a peer-to-peer interaction model between components.

References

[GOF95]: Gamma, Helm, Johnson, Vlissides: Design Patterns, Addison-Wesley, 1995

[OMG92]: The Object Management Group: The Common Object Request Broker: Architecture and Specification, OMG Document Number 91.12.1, 1992

[Brockschmidt95]: Brockschmidt, Inside OLE 2, 2nd Edition, Microsoft Press 1995

[Rymer94]: J. Rymer: OMG's UNO Object System Interoperability - at Last, Distributed Computing Monitor, Vol. 9, No. 12

[Tanenbaum92]: Andrew Tanenbaum: Modern Operating Systems, Prentice-Hall, 1992

[JS95]: Jell, Stal: Comparing Microsoft OLE 2 and OMG CORBA, Conferencez OOP, Munich, and Object Expo, London, SIGS Conferences, 1995

[BMRSS96]: Buschmann, Meunier, Rohnert, Sommerlad, Stal: Pattern-Oriented Software Architecture, Wiley and Sons, to be published in 2/1996

[C94]: F.R. Campagnoni: IBM's System Object Model, Dr. Dobbs Journal, pp.24, Special Report, Winter 1994/95

Douglas C. Schmidt: personal mail.