A CORBA-based Architecture for Electronic Auction Applications

Markus Aleksy, Axel Korthaus, Martin Schader
University of Mannheim, Germany
{aleksy|korthaus|mscha}@wifo3.uni-mannheim.de

Abstract

In this paper we present a CORBA-based architecture for electronic auction applications. The core of this architecture is represented by two services, which considerably facilitate the implementation of an electronic auction house by performing some of the main tasks that typically have to be implemented by a developer in this context, such as distributing and recollecting bidding data. Besides explaining this core architecture we describe additional components that are needed for the implementation of a complete auction house. Finally, we discuss further aspects concerning a flexible implementation of an auction application such as the support for different kinds of auctions and fault tolerance.

1. Introduction

An architecture for an electronic auction house has to be very flexible. Not only should the architecture be able to support various kinds of auctions and to preserve the anonymity of the bidders at any time, but it must also be able to handle different kinds of technical and business problems such as the crashdown of a bidder's computer or the late joining of a bidder. Interoperability with other systems is another important aspect. An auction application should be able to exchange information with existing legacy systems such as Enterprise Resource Planning (ERP) systems. By accessing an enterprise's ERP systems, e.g., product information available from existing databases can be used, and the business processes concerned with the auction process in general can be optimized. In [1], the authors describe how to access SAP R/3 systems from Java- and CORBA-based applications. A number of important aspects relevant to electronic auctions are discussed in [6] and [7].

We chose the Common Object Request Broker Architecture (CORBA) [8] as the basic communication architecture for our prototype. The CORBA standard is widespread in the field of object-oriented and distributed systems. It brings about independence from computer architectures and programming languages as well as the possibility for the user to choose an Object Request Broker (ORB) product vendor-independently. A prerequisite to this last characteristic was the introduction of a mechanism for uniquely referencing objects on the basis of so-called *Interoperable Object References* (IORs) and of a standardized transmission protocol, called *Internet-Inter-ORB-Protocol* (IIOP), in CORBA 2.0. Thus, applications that were developed using different programming languages can be made to interoperate.

For the description of interfaces belonging to classes that offer their services, the Object Management Group (OMG) has specified the *Interface Definition Language* (IDL). The IDL is a declarative language, i.e., it is used to describe data types and interfaces by specifying their attributes, operations and exceptions but not their actual implementation algorithms. The IDL is the basis for the achievement of programming language independence, and the mapping to a concrete programming language does not take place before an IDL interface is being compiled using an IDL compiler for the target language. Besides the language mappings described in the OMG standard, which include the mappings from IDL to Ada, C, C++, COBOL, Java, and Smalltalk, there are a number of non-standard language mappings to further programming languages such as Eiffel, Lisp, Objective-C, and Perl, which only exist in certain ORB products.

2. Architecture of an e-auction system

Analyzing the general process of an electronic auction, it turns out that two specific kinds of communication techniques may be needed:

- group communication and
- asynchronous communication.

Group communication is required to be able to efficiently send a bid, e.g., the initial bid or the currently highest bid, to any given number of bidders who submit new bids independently of each other. In this context, asynchronous communication is important because the initial bid has to be propagated to each bidder without blocking the sender (the auctioneer) on the one hand, and individual bids put by the bidders can arrive asynchronously, i.e., at different points in time, and have to be collected and evaluated subsequently on the other hand.

The basic communication model specified by CORBA describes a kind of communication that can be characterized as being synchronous and blocking. There are no constituent parts in the standard referring to any kind of group communication. Thus, the programmer is confronted with the problem that some of the techniques needed for the implementation of an auction application are not sufficiently or even not at all specified in the CORBA standard. For this reason, we have developed two CORBA services that remedy these shortcomings.

3. The reusable core architecture

The reusable core architecture we propose, which can be used not only in the context of auction applications, comprises the following components:

- an Object Group Service,
- a Join Service,
- one or more Masters, and
- one or more Workers.

The general idea of a CORBA-based Object Group Service (OGS) is based on the work by Felber [5], who uses this approach in order to facilitate the replication of data. Our design and the corresponding implementation, on the other hand, are aimed at the parallel processing of CORBA calls, i.e., a message sent by a master is delivered to any number of workers concurrently, and the workers process the data included in the message independently of each other. To that end, our solution supports the transmission of the complete data sets to all the workers as well as the application of several distinct data dispatching policies. A detailed description of the OGS and the supported data dispatching policies, their design rationale and the OGS's advantages over the CORBA Event Service [9] can be found in [2] and had to be left out here for reasons of scope. The interested reader is kindly asked to consult this earlier paper.

By the design of our OGS it becomes possible to distribute the source data independently of the data types and data structures used. The only restriction that has to be observed by the programmer is the need to organize the data as a CORBA sequence, i.e., a vector of variable length, if a dispatching policy is to be applied that differs from the policy of sending all data to all workers. However, the single data elements in the sequence can have any simple or complex structure. Basic data types are just as allowable as complex data types containing others such as basic data types, arrays, sequences or user-defined, possibly layered data structures. At run-time, the OGS dynamically determines the data types contained in the CORBA sequences and copies the data into new subsequences of the same data type. How many of those subsequences have to be created depends on the number of workers involved and the number of data sets to be dispatched.

One positive aspect of this approach is that it enables the developer to even dispatch sequences of data the types of which were not foreseen at the time of construction of the OGS. Therefore, the OGS is capable of supporting a broad scope of current and future fields of application. The price for this flexibility gain is a certain loss of performance, because the *marshalling* and *unmarshalling* of CORBA type `any` requires more time than that of simple data types.

To be able to use the OGS, an OGS client has to implement the `Master` interface. It only contains the operation `receive()`, which is called by the Join Service (JS) and is handed over a collection of the results produced by the workers. The servers representing workers, on the other hand, have to implement the `Worker` interface including operation `send()`, which has to be provided not only with the actual message data but also with the Interoperable Object Reference (IOR) of the JS. This is necessary to be able to apply a kind of callback technique. First, the master sends a message (including its IOR) to a certain group by calling `send()`. Then, the group forwards the call to each of its members (the workers) and informs the JS of the fact that a number of worker results are to be expected. Later, the results produced by the workers arrive at the JS which collects them and finally calls method `receive()` on the master in order to inform the master of the complete end result of the processing.

The purpose of IDL interface `Group` is to specify functionality for forwarding a call issued by a client to each member of the respective group. It also contains operations for the attachment and detachment of servers. In order to be able to manage different groups, we defined the `GroupManager` interface. It is equipped with operations that can be used to create, list, retrieve, and delete groups.

The JS represents the counterpart of the OGS. Like the OGS, the JS supports different policies:

- `COMPLETE` and
- `PARTIAL`.

Choosing the `COMPLETE` policy has the effect, that the JS will wait for the results of all the workers in any case. This means that if one of the workers crashes during processing, no result will be produced and, consequently, the waiting thread will be blocked infinitely.

This drawback can be avoided by applying the `PARTIAL` policy. In that case, the JS has to be provided with an additional time value. The JS then maximally waits for this specified amount of time, and if not all results have arrived from the workers before the deadline, it reports only those results that have already been delivered. Should all the workers call back before the timeout period has passed, the complete end result is delivered immediately.

A detailed description of the whole architecture can be found in [2]. Figure 1 shows the collaboration of the architecture's main components.

For applications with only a small amount of masters and workers the OGS can be started using the command line option `--use-internal-JS`. If this option is selected, the OGS falls back on a built-in JS for collecting the single results of the workers instead of using an external one.

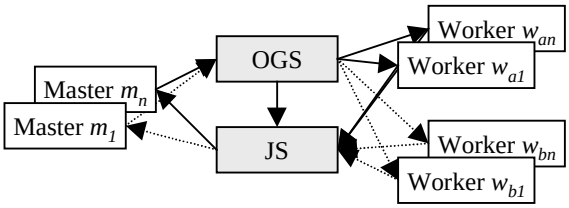

Figure 1: Structure of the core architecture

The same behavior occurs if no external JS can be found. In this case, the architecture can be compared with the "Master/Slave" design pattern described in [4].

4. Scalability of the core architecture

The core architecture allows a *from-one-to-many/from-many-to-one* communication model. It must be considered, however, that with an increasing number of masters and workers or large amounts of data to be dispatched the two services can become considerable bottlenecks. To avoid this situation it is possible to start more than one instance of each of the two services. Using a Naming Service [10] or a Trading Service [12], respectively, a master can look up the IORs of different OGSs and can subsequently decide to which one it wants to send its request. Figure 2 illustrates this process.

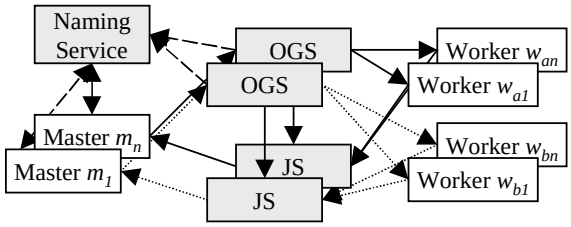

Figure 2: Scalability of the core architecture

Furthermore, the OGS can prescribe on the group level to which JS instance the workers have to deliver their results. In our case, the selection of the responsible JS instance is based on a round-robin load balancing strategy. Within an application both OGSs that only communicate with one single, internal or external JS, and OGSs that use several JS instances can be used at the same time.

5. Extending the core architecture for the design of an e-auction system

The core architecture provides the prerequisites needed for the implementation of an electronic auction house. The concept of a group corresponds to an auction. The bidders are represented by workers and the tasks of an auctioneer are performed by a master.

Since auctioneers and bidders are special kinds of masters and workers, the corresponding interfaces have to be extended. For that reason, auctioneers and bidders are represented by their own interfaces. The `Auctioneer` interface extends the `Master` interface and defines additional operations such as `start_auction()`. The `Bidder` interface, on the other hand, extends the `Worker` interface and adds an operation called `inform()`. The purpose of this operation is to inform the bidders of the result of the auction.

Another component specifically developed for the implementation of electronic auction houses represents an auction manager. `AuctionManager` is a central component responsible for a number of management tasks such as registering bidders, maintaining user data and product descriptions, creating and deleting groups and managing several auctions at the same time. Although only this component accesses the core data relevant to the electronic auction house directly, the data maintenance process can be per-

formed in a decentralized way. To that end, clients may access the management operations of the `AuctionManager` component (e.g., `addProduct()` or `removeProduct()`).

Thus, our CORBA architecture for electronic auction houses supplements the services discussed before with the following components:

- any number of auctions, represented by groups,
- an auctioneer (`Auctioneer`) for each of the auctions executed in parallel, who informs the bidders of the current state of the auction with the help of the OGS and receives collections of new bids from the JS,
- usually several bidders (`Bidder`) per auction who register with a group (an auction) and send their bids to the JS, and
- an auction manager (`AuctionManager`) responsible for the management of data.

Figure 3 presents a screenshot of the `Auctioneer` GUI taken from our implementation. It shows a list of all available auctions, represented by the respective product to be sold. There are several buttons for the different services an auctioneer has to perform, such as the creation and destruction of auctions. To register an auction means to start the management of the auction process.

Figure 3: Main GUI of the prototypical `Auctioneer` component of our implementation

On the right side of Figure 3, an auction description can be seen, i.e., the specification of the product to be sold by the auction is displayed.

Figure 4 displays a dialog window of the `Auctioneer` GUI which shows the details of a running auction, while Figure 5 displays a dialog window of the `Bidder` GUI, containing a textfield for the entry of a new bid, and the remaining time for that bid as well as information about the currently highest bid.

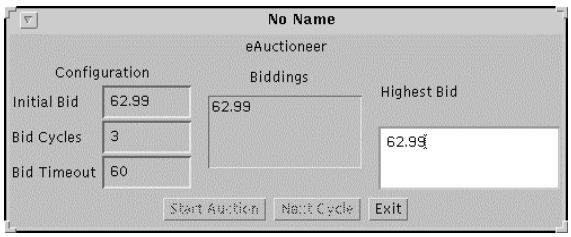

Figure 4: Dialog box of the `Auctioneer` component showing a running auction

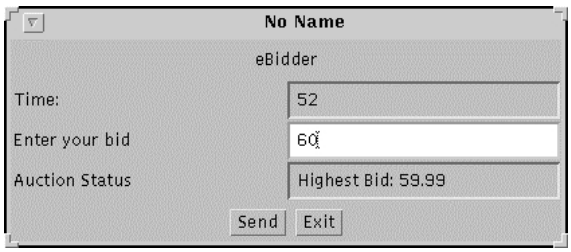

Figure 5: Dialog box of the `Bidder` component showing a running auction

6. An OGS scenario modeled with UML

Figure 6 shows a UML Deployment Diagram corresponding to our implementation. It is an instance level diagram depicting a snapshot of a system implementing the OGS and JS approach at run-time.

The different computational nodes, e.g., the computer on which the master is running, the network etc., are symbolized by 3D boxes. The CORBA objects implementing auctioneer, bidders, the OGS, the JS and the Auction Manager are modeled as UML components with explicit interfaces (shown in "lollipop" notation) which represent their IDL interfaces.

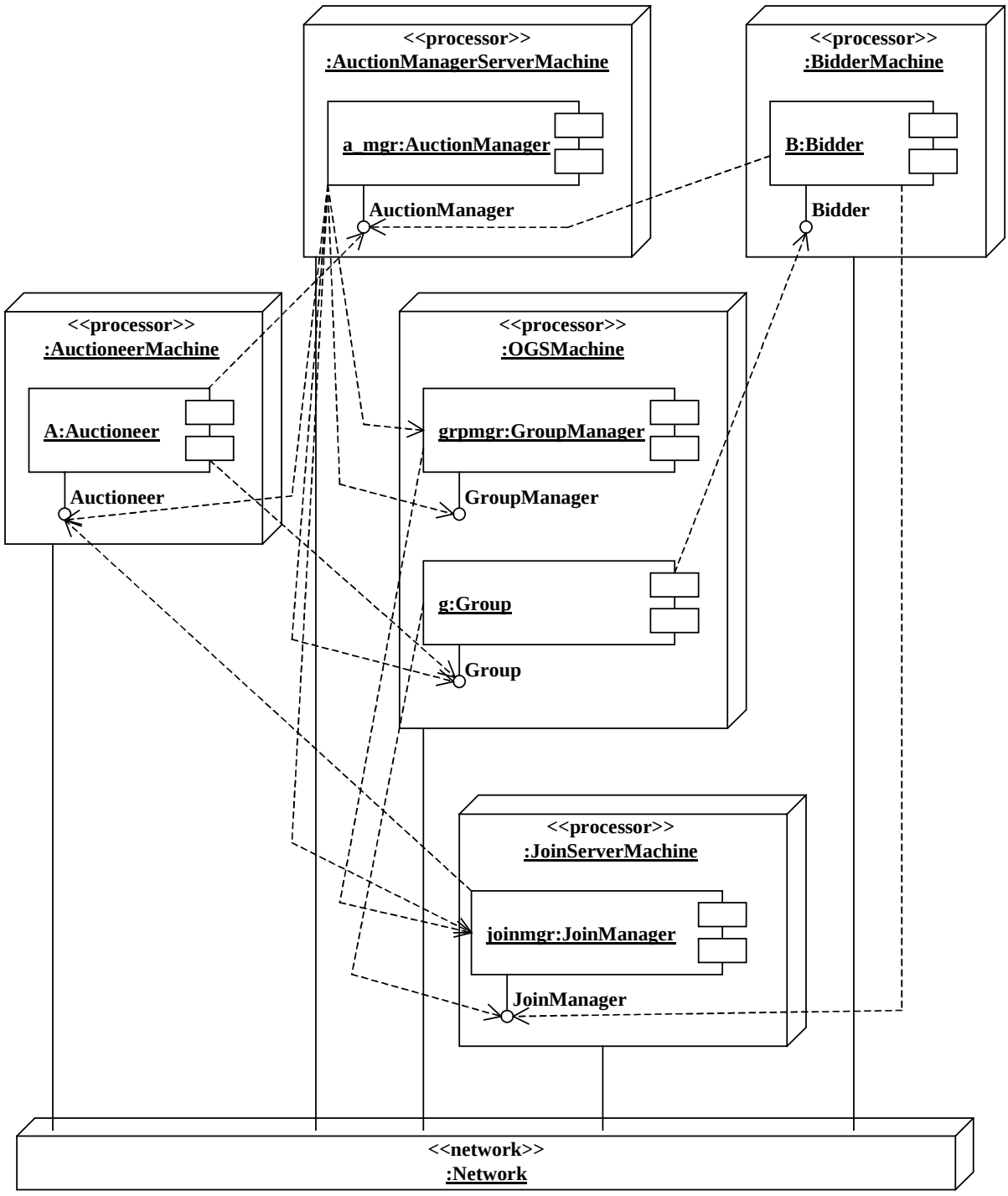

Figure 6: UML deployment diagram of the e-auction system

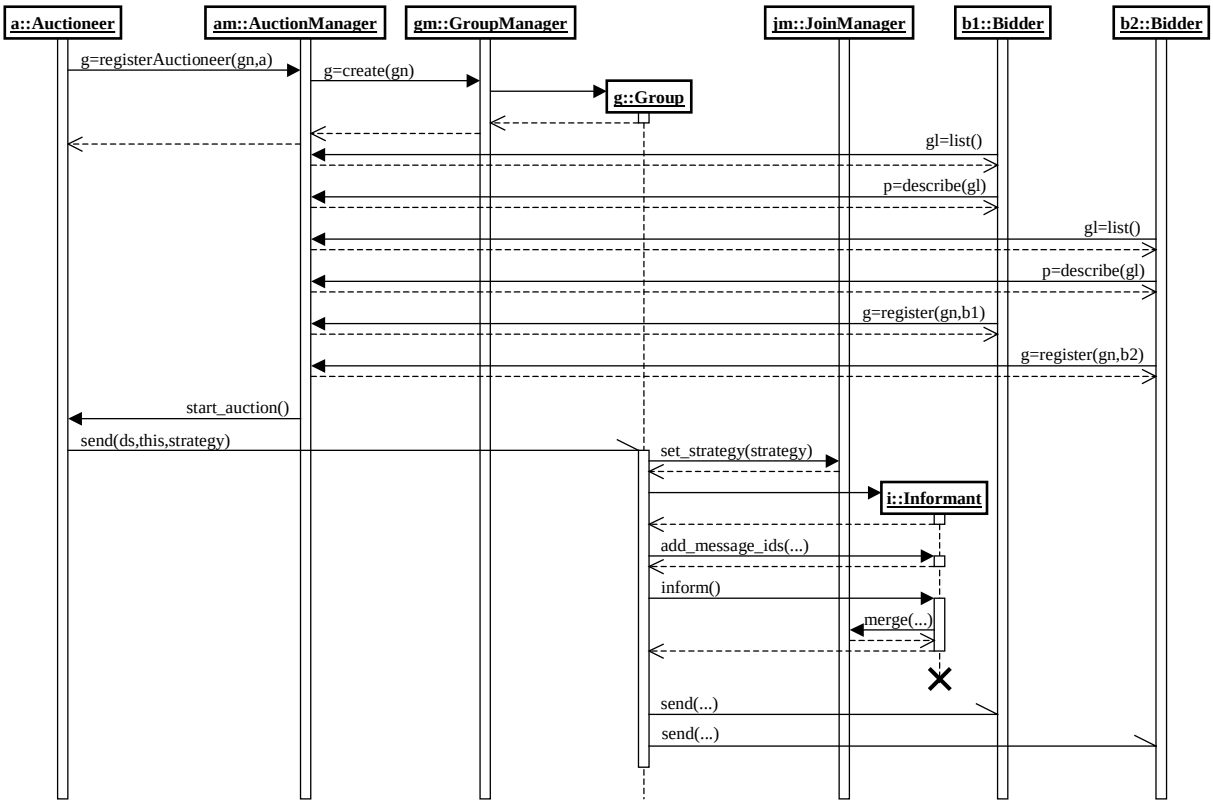

Figure 7: A partial example scenario of the e-auction system at work

The solid lines between the nodes show hardware connections used for the actual flows of information. Dependencies between the components are expressed by dashed lines, e.g., a component uses an interface of another component. The UML sequence diagram shown in Figure 7 gives an idea of how synchronous and asynchronous communication is used within our approach.

7. Different kinds of auctions

If an electronic auction house is to be used for various business purposes it must support different types of auctions.

The most commonly practiced kinds of auctions are:

- *Sealed bid auctions*:
  If this type of auction is applied bidders have to submit their bids by a certain deadline. The auctioneer keeps these bids confidential and announces the winner not before the deadline has passed. A sealed bid auction can last for one or more rounds. In a multiple round sealed bid auction the bids are publicized at the end of each auction round and a new set of bids is solicited.
- *Increasing-price auctions* (*English auctions*):
  In an English auction an initial bid is announced for the good to be sold. Afterwards, the bidders try to outbid each other. The very bidder who places the highest bid gets the knocking down.
- *Decreasing-price auctions* (*Dutch auctions*):
  Dutch auctions start with a relatively high price which is decreased successively. The first bidder to be prepared to pay the current price gets the knocking down.

Each type of auction described above, excluding single round sealed auctions, needs facilities for a multi-level data distribution which are provided by our core architecture. In the case of a multiple round sealed bid auction the auctioneer sends an initial bid to the OGS (1), and the OGS forwards this bid to each of the bidders (2). Their bids (3) are collected by the JS and sent back to the auctioneer (4). Subsequently, the auctioneer can spread those intermediate bids via the OGS again (5), (6). Once again, the results of the bidders (7) are received by the JS and delivered to the

auctioneer. This process can be iterated as many times as necessary.

The following figure shows a schematic illustration of this process:

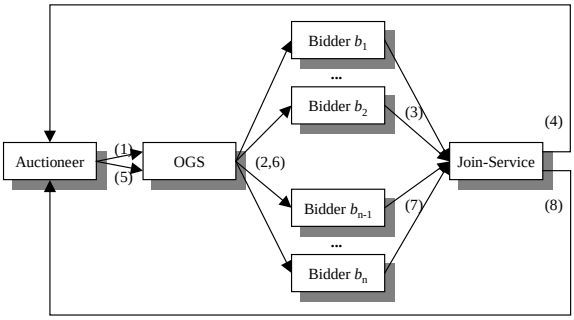

Figure 8: Recursive use of OGS and JS

An equivalent scenario applies to English auctions, except for the fact, that a timeout value has to be set for the auctioneer instead of a deadline.

With respect to the multi-level dispatching of data, Dutch auctions do not differ very much from the two other types of auctions described before. Here, the auctioneer transmits the initial bid on a regular basis, each time subtracting a certain amount until one of the bidders is willing to pay the current price.

To ensure that all participants operate on a common time basis it is necessary to synchronize the local clocks of the different computers. This can be done for instance by using the CORBA Time Service [11] in a standardized manner.

8. Fault tolerance and flexibility

Independently of the kind of auction at hand it may occur that one or more bidders do not submit any bids at all during an iteration. If no bid arrives this may have one of serveral different causes, e.g.:

- the bidder may not be interested in the auction any more and stops its application prematurely,
- the bidder may have technical problems such as crash of the computer, the network connection, the operating system or the application and thus cannot participate in the auction,
- the bidder may want to observe the trend of the auction before issuing a bid on his own.

For this reason, the auctioneer application should only use the JS's PARTIAL policy (see section 3), because this automatically solves potential problems that might result from the situations described above. With this policy the JS does not wait for any new bids to arrive any more after a specific deadline has passed and only transfers those bids to the auctioneer that arrived at the JS before the deadline.

Another interesting aspect is the handling of bidders who want to join a running auction (*late join*). Since the auctioneer application sends the initial bid or the currently highest bid to the OGS and gets new bids from the JS, the number of bidders is transparent to the application. Consequently, a new bidder may even join a running auction without any effects on the auctioneer application.

The auctioneer application is not restricted to managing a single auction only. If it is implemented in a programming language which supports multi-threading it can supervise more than one auction concurrently. Although this might lead to a better utilization of the system, it requires a much more complex implementation of the auctioneer application.

9. The process of an electronic auction from the bidder's perspective

After the start of its application a bidder can find out the available auctions by using operation `list()` which is part of the `AuctionManager` interface. If the bidder is interested in an auction, he can learn more details about the product to be sold by calling operation `describe()` defined in the same interface (see Fig. 4). The product description is allowed to contain textual elements as well as images. The following listing showing a part of the corresponding IDL interface illustrates this:

```
typedef sequence<string> desc;
typedef sequence<octet> image;

struct ProductInfo {
  double initial_bid;
  desc description;
  image illustration;
};
```

In the example, two new data types `desc` and `image` are defined. Data type `desc` is a sequence of `string`s containing the description of the good to be sold by auction. Data type `image` is a sequence of `octet`s which have a size of 8 bits and are not converted when they are transmitted. This feature is needed because the data is a binary representation of an image of the respective product, e.g., in GIF or JPEG format.

The data structure named `ProductInfo` contains the two data types mentioned before and a variable for an initial bid. In this way, a bidder is provided with all the information he needs for participating.

If the product meets the need of the bidder it will have to register itself with the auction manager by calling operation `registerBidder()` of interface `AuctionManager`. At the beginning of the auction, the initial bid is sent to the bidder automatically, independently of the kind of auction to be performed. Should the bidder want to submit a bid, it is transferred to the JS using operation `receive()`. Then, the JS collects the bids of all the bidders and sends the result to the auctioneer. This step might be iterated several times.

9.1. Connecting the e-auction system to the Internet

Using CORBA, it is possible to offer bidder applications either as standalone applications that have to be installed locally or in the form of an ORBlet-based solution that has to be downloaded automatically from a HTTP server at run-time. ORBlets are Java applets which use CORBA as their underlying communication middleware and know how to communicate with other CORBA-based components via IIOP. The main advantages of the use of ORBlets are platform independence, simplicity and usability (provided the user knows how to handle a web browser), and easier software distribution, e.g., if updates have to be made available, because only one new version has to be installed which can be downloaded by any number of clients automatically from the server.

If the user does not have a Java-compliant web browser or if he has disabled the Java feature, a different solution has to be found. One approach to this problem could be a CORBA- and servlet-based server receiving HTTP requests and converting them into CORBA requests. Alternatively, Java Server Pages (JSP) can be used. However, the way back, i.e., informing the HTTP bidder of new bids, is problematic, because there is no real-time solution to this problem in the web context.

10. Critical review

To the developer of an e-auction system, the choice of CORBA as the underlying middleware provides several advantages. First, the technology allows the realization of internet-wide real-time auctions, which would not be possible with a pure HTTP-based approach. Second, CORBA facilitates the integration of the auction application with existing legacy applications, such as ERP systems, so that exchange of information between those applications can be achieved automatically. Although the collaboration with pure HTTP-based clients is possible with our approach, this kind of usage shows the disadvantage that clients of that kind cannot participate in real-time auctions.

The core architecture presented provides several interesting benefits. The combined OGS and JS approach supports the parallel processing of data independently of the programming language in use. The concurrency of the processing is transparent to the user, i.e., he can implement bidders and auctioneers as sequential programs. Especially developers who implement their applications with a programming language that does not provide multithreading mechanisms are, nevertheless, enabled to produce parallel applications. Furthermore, our approach supports developers who do not have any or much experience in parallel programming. They do not have to care about such tasks as data distribution or synchronization of incoming calls, because the architecture encapsulates these mechanisms and simplifies their use by abstracting them within single invocations and returns of the end results.

By using the CORBA data type `any` as the basis for data distribution, it is possible to use both services for a multitude of applications.

However, the flexibility provided by the architecture is not restricted to the type of data that can be exchanged. Offering different policies, the two services support even unforeseen situations that might occur during an auction, such as the crashing of a bidder application or bidders that want to join an already running auction, and facilitate solutions to these situations.

Another example of families of applications that can make use of our core architecture are meta search engines. In that case, a master issues a request to a meta search engine (implemented by the OGS), which forwards the request to different search engine services (workers) working independently of each other. These services send the request to corresponding search engines such as `www.altavista.com`, `www.google.com`, `www.lycos.com`, and so on. The replies produced by the search engines can arrive at different points in time and have to be collected (by the JS) and evaluated (by the master). Again, asynchronism and group communication play an important role in this application context. Applying our core architecture helps to hide these technical details from the developer in a comfortable and simple way.

References

[1] Aleksy, M., Korthaus, A. (1999): "Interoperability of Java-Based Applications and SAP's Business Framework - State of the Art and Desirable Developments, in: Proceedings of International Symposium on Distributed Object and Applications (DOA '99), 5.-6. Sep. 1999, Edinburgh, Scotland, IEEE, pp. 190-200

[2] Aleksy, M., Korthaus, A. (2000): "A CORBA-Based Object Group Service and a Join Service Providing a Transparent Solution for Parallel Programming", in: Proceedings of the International Symposium Software Engineering for Parallel and Distributed Systems, 10-11 June 2000, Limerick, Ireland, IEEE, Los Alamitos, California, pp. 123-134

[3] Aleksy, M., Korthaus, A. (2000): "Implementation Techniques and an Object Group Service for CORBA-Based Applications in the Field of Parallel Processing", in: Proceedings of the Seventh International Conference on Parallel and Distributed Systems, 4-7 July 2000, Iwate, Japan, IEEE, Los Alamitos, California, pp. 65-72

[4] Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996): A System of Patterns – Pattern-Oriented Software Architecture, John Wiley & Sons, Chichester

[5] Felber, P., Garbinato, B., Guerraoui R. (1996): "The design of a CORBA group communication service"; in: Proceedings of the 15th IEEE Symposium on Reliable Distributed Systems, Niagara-on-the-Lake, pp. 150-159, ftp://ftp-lse.epfl.ch/pub/felber/papers/SRDS-96.ps

[6] Harkavy, M., Tygar, J. D., Kikuchi, H. (1998): "Electronic Auctions with Private Bids"; in: Proceedings of the 3rd USENIX Workshop on Electronic Commerce, 31 August-3 September 1998, Boston, USA, http://www.usenix.org/publications/library/proceedings/ec98/full-papers/harkavy/harkavy.pdf

[7] Kumar, M., Feldman, S. I. (1998): "Internet Auctions"; in: Proceedings of the 3rd USENIX Workshop on Electronic Commerce, 31 August-3 September 1998, Boston, USA http://www.usenix.org/publications/library/proceedings/ec98/full-papers/kumar_auctions/kumar_auctions.pdf

[8] OMG (2000): "CORBA/IIOP 2.4 Specification"; OMG Technical Document Number 00-10-01, ftp://ftp.omg.org/pub/docs/formal/00-10-01.pdf

[9] OMG (2000): "Event Service Specification"; OMG Technical Document Number 00-06-15, ftp://www.omg.org/pubs/docs/format/00-06-15.pdf

[10] OMG (2000): " Naming Service Specification"; OMG Technical Document Number 00-11-01, ftp://ftp.omg.org/pub/docs/formal/00-11-01.pdf

[11] OMG (2000): "Time Service Specification"; OMG Technical Document Number 00-06-27, ftp://ftp.omg.org/pub/docs/formal/00-06-27.pdf

[12] OMG (2000): "Trading Object Service Specification"; OMG Technical Document Number 00-06-27, ftp://ftp.omg.org/ pub/docs/formal/00-06-27.pdf

[13] Schmidt D. C., Vinoski S. (1998): "An Introduction to CORBA Messaging"; in C++ Report, SIGS, vol. 10, no. 10

[14] Schmidt D. C., Vinoski S. (1999): "Programming Asynchronous Method Invocations with CORBA Messaging"; in C++ Report, SIGS, vol. 11, no. 2