Malaysian Journal of Computer Science, Vol. 14 No. 2, December 2000, pp. 95-105

SEALED-BID AUCTION PROTOCOL IMPLEMENTATION OVER CORBA ARCHITECTURE

Mohammad Zahidur Rahman and Sai Peck Lee
Faculty of Computer Science & Information Technology
University of Malaya
Kuala Lumpur, Malaysia
saipeck@fsktm.um.edu.my

ABSTRACT

Verifiable secret and polynomial sharing (VSPS) scheme can be adopted in the development of a sealed-bid auction protocol for secure sealed-bid auction service. In this paper, issues related to the implementation of VSPS scheme for sealed-bid auction service is discussed. Distributed object computing over Common Request Broker Architecture (CORBA) is deployed in the protocol. To allow the access of an auction server simultaneously by more than one bidder, the concept of concurrent processing is introduced.

Keywords: ***Distributed computing, Object-Orientation, Electronic auction***

1.0 INTRODUCTION

The sealed-bid auction protocol is used for auctioning services or goods, which requires an extended time to prepare the counter bid, and where it is not possible to gather all the prospective bidders at a common place at a specific time. The trend in the last decade is that trading over the Internet was increasingly common though there were some security glitches yet to be taken care of. The designed sealed-bid auction protocol in [8] is based on the object-oriented Common Request Broker Architecture (CORBA) [1]. CORBA encompasses a series of standards and protocols for inter-process communication in a heterogeneous environment. The CORBA application is based on objects. Objects reside on different machines throughout the distributed environment. A client object can communicate with a server object through an object reference. This object reference is resolved by the object request broker (ORB). When a request is reached to the ORB, the request is passed to an object adapter. The object adapter forms a link between an object's implementation and its presence on the ORB.

In order to reap the benefit of the distributed nature of CORBA, several auction servers are distributed over the network such that the availability of all servers is not compromised. The security of the distributed servers is achieved by using the secret sharing scheme, known as verifiable secret and polynomial sharing (VSPS) [2]. The interfaces between the auction server objects and the bidder objects are designed using CORBA IDL (interface definition language). As the IDL of these objects defined always remain constant, any change or improvement to any bidder object and/or auction server object will not be detected. The portable object adapter (POA), a refined concept of basic object adapter (BOA), is proposed in the new CORBA specification [4]. The advantage of using POA is that it enables porting one version of the server object to a new version without requiring to notify the clients who are using that server object. In this paper, we give some lights on the software development aspects of the auction protocol, both in secret sharing scheme and CORBA implementation.

The online auctions currently available use the features of the hyper text markup protocol (HTTP). The web-based auction systems suffer from the disadvantages like requirement of reloading and resubmitting of data, which takes quite some time to perform. To perform a data transfer, the web server has to recreate a HTML page and then sends the whole file back to the browser. On the contrary, the Internet Inter-ORB Protocol (IIOP) of CORBA consists of a common data representation (CDR), an interoperable object reference (IOR) format, interoperable TypeCodes for all data types and Inter-ORB Protocol (IOP) message contents, formats, and semantics mapped to TCP/IP. This configuration results much faster data transfer in comparison with a web-based application.

Section 2.0 briefly describes the auction protocol implemented. In Section 3.0, we discuss about the VSPS related problems. The usage of portable object adapter (POA) and the naming service of CORBA are discussed in Section 4.0. In Section 5.0, the concurrency related problems are addressed and shown how they can be solved.

2.0 OVERVIEW OF PROTOCOL FOR SEALED-BID AUCTION SERVICE

Our auction protocol stated in [8] starts from the registration of bidders. There should be some registration policy under which the prospective bidders register themselves and get a certificate. The registration is important in the serious real-life, sealed-bid auction. Some sealed-bid auctions require bidders to submit a bid deposit in the form of a bank cheque or some other means. In some cases, the bidder should support bid with some documents such as tax returns, etc. The bank or government agency normally issues these instruments. The presence of a registration server can ease the operation of the auction protocol in future when these types of services mature in electronic form.

After a predefined time, the registration of bidders will be closed. Bidding can start immediately or after some predefined time when all good auction servers are ready to start auction service. At the beginning of the bidding service, each bidder sends his/her own certificate to all auction servers as the prerequisite for bidding. When a certificate is verified by all the auction servers, an independent private channel based on the Internet will be established between the bidder and each of the auction servers. The context diagram of our sealed-bid auction scheme is shown in Fig. 1. The private channel between the bidder and an auction server is built over the Internet using available cryptographic technology.

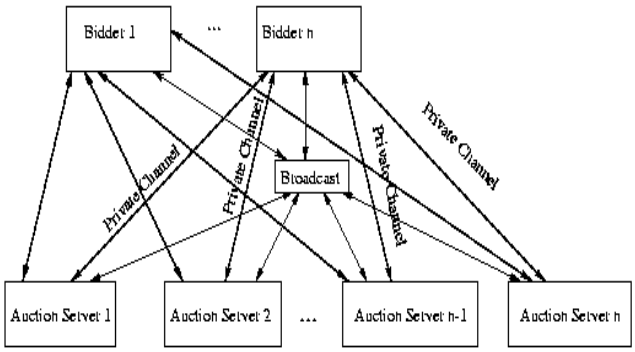

Fig. 1: Sealed-bid auction scheme

A session key and some parameters like n, the number of auction servers, and t, the degree of polynomial, will be passed by each auction server to the bidder. Then the bidder will choose an auction item, price quote and auction ID to form a secret bid. The bidder now randomly chooses a polynomial $f(x)$ of degree t with a free term equals to the secret bid, and another random polynomial $r(x)$ of degree t. He/she then computes the share values of the secret bid, $\alpha_i = f(i)$ and $r_i = r(i)$, based on the algorithms of the polynomials chosen, where i is for all n participating auction servers. Before sending to ith auction server via the private channel, the bidder encrypts ith pair of α_i and ρ_i with the session key of ith participating auction server. The commitments of α_i and ρ_i, where $i = 0$ *to* n, are broadcasted to all participating auction servers openly. According to the current protocol, the commitment of a secret is the one-way hash of that secret. The individual ith auction server compares the received pair (α_i and ρ_i), which was passed via the private channel, with the corresponding broadcasted commitment. If the validity fails, the auction server informs all other participants including the bidder that some problem has occurred. Each auction server also verifies that the broadcasted commitments lie within a committed polynomial. If a bidder receives a broadcasted complaint from the ith auction server, he/she openly broadcasts α_i, ρ_i and the corresponding commitments to defend his/her integrity. If the bidder does not follow certain steps, he/she will be disqualified; otherwise it should be concluded that the bid from the bidder is shared perfectly.

In the sealed-bid auction, a negotiation process ends with the seller closing the auction at a time based on pre-agreed rules, such as at a previously agreed time, or after a certain duration of inactivity, or a combination of the two. When the auction time is over, all the auction servers close the receiving of bids. Depending on the auction rules, after a predefined time, the auction servers send each other all the shares that they had received in earlier sessions to reconstruct each bid. Before reconstructing a bid, the auction servers have to check that the shares received are valid by checking with the commitments published during the bidding session. When all the bids have been reconstructed, the winner is declared according to the predefined rule. There is a deal if there is at least one bidder who has the highest bid that exceeds the reserved price, if the seller has specified one.

3.0 VSPS IMPLEMENTATION ASPECTS

The secret bid sharing in the proposed sealed-bid auction protocol uses the simplified verifiable secret and polynomial sharing (VSPS) scheme. Before starting the shares calculation, the bidder should choose two large prime numbers, p and q, such that $p=mq+1$, where μ is a small integer. These two prime numbers are to be known to all the parties involved. This requirement is necessary because we need the numerical group Z_p^* which contains a subgroup of large prime order. Recalling from the basic algebra that Z_p^* has a cardinality of $p-1$; and for each prime divisor of the cardinality of a group, we can come to the conclusion that there is a subgroup of that order. Another two integer variables g and h, such that $h = g^z \mod p$, are required for calculating verifiable hash, where z should be unknown to all parties. This raises a question: who should know it? There is one possibility of a presence of an Arbitrator. Normally p, q and g are generated by the bidder, then the Arbitrator calculates h and publishes it. Another possibility is to take a random number r (for example, the hash of the current date and time) and set $h=r^m \mod p$ (this will make sure that h is an element of order q). As h is pseudo-randomly generated, nobody knows z. If q divides $p-1$ but q^2 does not divide $p-1$, an easy way to do this is to choose two random numbers r_1 and r_2 in Z_p^*, then set $g=r_1^{(p-1/q)}$ and $h=r_2^{(p-1/q)}$. Thus g and h as generators of the subgroup of order q in Z_p^* is accomplished. For the implementation of the secret sharing scheme, the latter solution is used.

According to the VSPS algorithm, for constructing verifiable commitments, two polynomials are used in the preparation of shares. These polynomials are to be computed over modulo q and not over modulo p. For the reconstruction of a bid, the shares from the first polynomial $(f(x))$ are fitted over the polynomial and the interpolated free term will be the reconstructed secret. The process of reconstruction uses the inverse of a matrix. The verification of hash also uses the inverse of a matrix. The inverse of a matrix should not be computed over the reals or the rationals, but all the computations should be done over the modulo of the large prime q. For these mathematical implementations, LiDIA [5], an object-oriented mathematical framework for large numbers, is used.

4.0 SEALED-BID AUCTION PROTOCOL IMPLEMENTATION OVER CORBA ARCHITECTURE

A common idea is derived from the middleware approach which introduces a new layer into the program that keeps the complexity away from the developer by hiding as many details of distributed programming as necessary. To the developer, the middleware presents seemingly local objects, and invocations on the local proxy cause the necessary data to be transparently sent to the recipient. This allows clients to invoke operations on distributed objects without concern for object location, programming language, operating system (OS) platform, communication protocols and interconnections as well as the hardware [7]. One popular example of such middleware is the Common Object Request Broker Architecture (CORBA), which is part of the Object Management Architecture as specified by the Object Management Group (OMG). CORBA uses the Object Request Broker (ORB) as the glue between individual pieces, which is responsible for directing a client's method invocation to the appropriate object implementation.

While CORBA can indeed hide many details of client/server programming from the client, experience has shown that a much tighter involvement with the ORB is necessary on the server side. For this, CORBA uses the concept of object adapters which mediate between the ORB and the server on how to represent servants to the outside world. The server can choose an object adapter that best fulfills its requirements. CORBA is scalable and its location forwarding mechanism provides basic support for coarse-grained server mobility, but a vendor-specific forwarding service (i.e. an Implementation Repository) is necessary on the server side to employ the feature [6].

A programming concept with CORBA is that a remote method invocation causes a request to be sent from the client to the server, and the client usually waits synchronously until the reply is received [9, 14]. CORBA uses a declarative language, the Interface Definition Language (IDL), to describe an object's interface. This description is used by an IDL compiler to generate stubs for the client side and skeletons on the server side. Using this generated code, a remote method invocation looks like an invocation on a local object, at least in an object-oriented programming language like C++. The encoding of parameters is hardware-independent, which is referred to as CDR (Common Data Representation).

The nature of the proposed sealed-bid auction protocol demands a greater speed by spreading complex tasks over many computers working in parallel and increased fault tolerance can be achieved by using more than one server as described in the VSPS scheme. According to the protocol, a bidder contacts several server objects as defined by the auction rules to perform the auction service. The offering and using of services are of particular interest for the Internet today. Companies are not satisfied with presenting information on the World Wide Web only, but also keen to present their services online to the world-wide user community. The main challenges focus on the operating

system (OS) platform portability, connection management and service initialisation, event demultiplexing and event handler dispatching, synchronisation, fault tolerance and fault detection. CORBA helps to overcome these challenges. As such the proposed sealed-bid auction protocol is implemented over CORBA architecture. In the following subsections, we discuss several CORBA concepts and their usage in auction system implementation.

4.1 The Portable Object Adapter (POA)

The concept of portable object adapter (POA) is augmented with the basic object adapter (BOA) in the new CORBA specification 2.3 [4]. The POA is designed keeping in mind the portability and flexibility of objects. The POA comes with its own set of concepts. Many POA instances can exist in a server, organised in a hierarchical structure. The RootPOA is created by the ORB and subsequent POAs can be children of existing ones which were created by the working server. Each POA maintains its own Active Object Map so that it can map currently active objects to servants. Each object is identified by a unique Object ID within its name space and is activated by a particular POA instance.

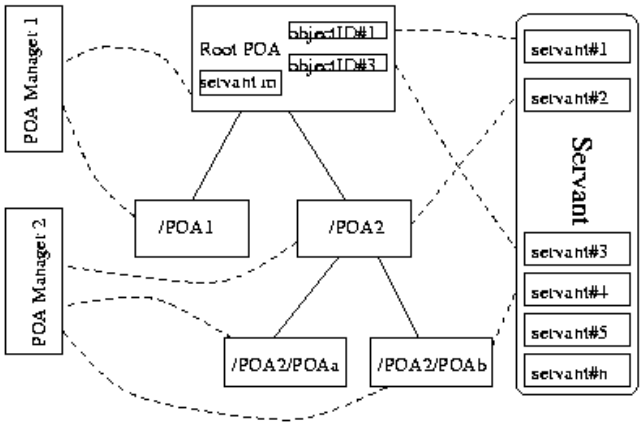

Fig. 2: Portable Object Adapters (POA) and its interaction with managers and servants

The POA managers are responsible for synchronising the different POAs, i.e. managers control the readiness of one or more POAs to receive requests. The other responsibility of the managers are to control and monitor the life cycle of the servants. *RootPOA* is created by ORB as shown in Fig. 2. *POA1* and *POA2* are children of *RootPOA* and *POAa* and *POAb* are children of *POA2*. The new four POAs (i.e. *POA1*, *POA2*, *POAa* and *POAb*) are now registered to servants, which adjust different aspects of a POA's behaviour. For example, *RootPOA* is registered with two servants *servant#1* and *servant#3*, which are kept in the active object map. During the life cycle of a POA, the POA changes its states according to the stimulation it receives. The state transition of a POA is shown in Fig. 3.

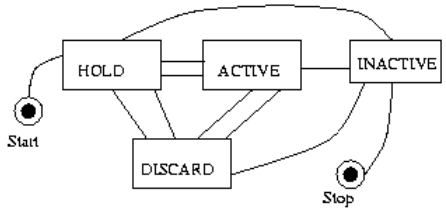

Fig. 3: State transition diagram of a POA

4.1.1 Policies Related to a POA

Each of the POAs has its own separate policies. Policies are selected by the user upon the POA creation and cannot be changed over its lifetime. Since objects are associated with a fixed POA instance, some policies can also be said to be that of an object, for example, the lifespan policy. According to the CORBA 2.3 specification, the following are the available policies regarding the portable object adapters.

Thread — It can be set to either "Single Thread" if the servants are not thread-aware and that requests must be serialised, or set to "ORB controlled" if servants are reentrant and requests can be processed regardless of ongoing invocations.

Lifespan — Lifespan of an object can either be a "transient" or a "persistent". The lifespan of transient objects (i.e. objects that are registered in a POA with the transient lifespan policy) is limited by the lifespan of the POA instance they were activated in. Once their POA is destroyed, for example, as part of server shutdown, object references to transient objects become permanently invalid. Persistent objects can outlive their original POA and even their original server. Missing POAs for persistent objects can be recreated, and if their server is shut down, it may be restarted at a later time and continue serving the object. The usage of the term "persistency" here is very different from its meaning as a Basic Object Adapter's (BOA) activation policy. All BOA servers satisfy the POA's persistent lifespan policy, because the servers could be stopped and restarted.

Object — An object can be activated with the POA more than once to serve more than one object. ID Assignment Policy defines whether Object IDs are generated by the POA, or selected by the user, for example, to associate objects with "human-readable" names or to hold identity information. If a single servant is registered more than once to serve multiple objects, it could use a user-selected Object ID (which would be different for multiple activations) at runtime to discriminate between them.

Servant — When an object is activated, the association between the object (or rather, Object ID) and the servant is stored in the Active Object Map. This behaviour can be changed if desired, for example, if a default servant is prepared to handle requests for newly-activated objects.

Request Object Map — When trying to serve a request, whether a servant manager is available, or whether the request should be delegated to a default servant, Request Object Map policy selects the option.

Implicit — Some operations on a servant require its activation, for example, the request of an object reference. Depending on this policy, performing such an operation on an inactive servant can either cause an error, or transparent activation.

4.1.2 Servant Manager

The usage of a servant manager is focused in the following cases. Like adapter activators for POAs, a servant manager can be used to activate servants on demand after a partial shutdown or after a server restart. The servant manager receives the request's Object ID and could use that information to read back the object's state from the persistent storage. Another interesting feature possible with servant managers are virtual objects, i.e. object references that refer to non-existent servants. References to virtual objects can be passed to a client; on the server side, a servant manager is then registered with the POA to incarnate the virtual object on demand.

Servant managers come in two different flavours with slightly different behaviour and terminology, depending on the servant retention policy. If this policy's value is "retain," a servant activator is asked to incarnate a new servant, which will, after the invocation, be entered into the Active Object Map itself - this would be sensible in the sketched file service example, since the newly incarnated file object would be needed more than once. If an object is deactivated, either explicitly or because of a server shutdown, the servant activator's *etherealize* method is called to get rid of the servant - at which point the object's state could be written to the persistent storage.

The POA servant managers are of two types: one activates a new servant called Servant Activator. POA's Active Object Map retains the information of the servant to serve further requests on the same object. The other servant manager type is Servant Locator which is used to locate a servant for a single invocation and then forgets the object. This type of servant will not be retained for future use.

If the servant retention policy is "non-retain", the servant manager would have to be a servant locator, whose task is to locate a servant suitable only for a single invocation. A servant locator supplements the default servant mechanism in providing a pool of default servants; it is the flyweight factory according to the flyweight pattern. It can also be used for load balancing, as the example of a print service shows, in which the *print* method is directed to the printer with the shortest queue.

4.1.3 Auction Protocol Implementation over CORBA

In a distributed system development using CORBA, a servant manager is responsible to manage the bids. The implementation of the auction service over CORBA uses the benefits of POA [12, 11]. The auction house is responsible for creating bid objects. However, the content for the bid object, such as its shares, is generated by the bidder. In the case of an auction server, the servant activator is chosen so that whenever the bid object is first used, the servant manager will incarnate and activate a new servant. Further operations on the same object reference will use the already active servant. The auction server's create operation executes the create_reference() operation on the POA, which does not cause an activation to take place. It only creates a new object reference encapsulating information about the supported interface and a unique system-generated Object ID. This reference is returned to the client bidder. For example, the pointer for the auction server, Nilam_ptr, is returned when it is created.

```
Nilam_ptr Auction_impl::create()
{
  Nilam_ptr retref;
  CORBA::Object_var obj = newpoa-> create_reference (char'IDL:Nilam:1.0char');
  retref = Nilam::_narrow(obj);
  return retref;
}
```

When the client invokes an operation on the returned reference, the POA will first search the Active Object Map. If the desired object is not found, the servant manager will be referred to serve the request and to find an appropriate implementation for the request. The collaboration diagram of the object finding is shown in Fig. 4.

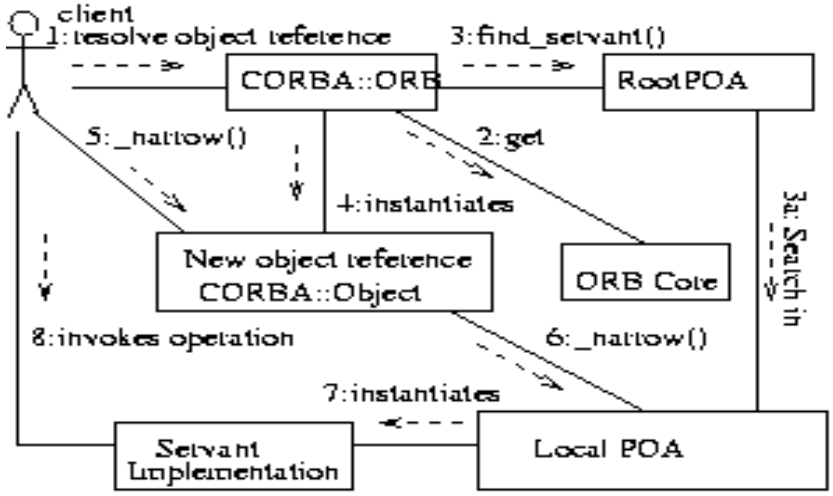

Fig. 4: Finding a Nilam object in CORBA implementation

The problem regarding the persistence of the auction server is important. A persistent object's lifetime is not bounded by the process that implemented it. If the object is not of persistent type, then due to some reason if the server is down and then restarted, and whenever the client invokes a service from the server, it will receive an exception that its object reference has become invalid. According to the CORBA specification of POA, an object is persistent if the servant that implements it is activated in POA that has PERSISTENT lifespan policy. Due to the persistent lifespan policy, the disruption of server service will not be noticed by the client object as long as the server is running whenever an invocation is performed.

In the case of the auction service, it requires to create persistent bids. When the server is down, it writes its states to a disk file and when the server is restarted, the states are read again. To accomplish this, a persistent POA is used to create the auction server object. A servant manager provides the necessary hooks to save by issuing *etherealising* to the auction server object, which writes the states to the disk and restores the states by *incarnating* a bid, which checks if an appropriate named file with the states exists. The following code snap shows how to use this technique.

```
CORBA::PolicyList pl;
pl.length(2);
pl[0] = poa->create_request_processing_policy (PortableServer::USE_SERVANT_MANAGER);
pl[1] = poa->create_lifespan_policy (PortableServer::PERSISTENT);
PortableServer::POA_var nilampoa =
poa->create_POA (char'Nilamschar', PortableServer::POAManager::_nil(), pl);
PortableServer::POAManager_var nilammgr = nilampoa->the_POAManager();
/* * Activate ServantManager */
NilamManager * am = new NilamManager;
PortableServer::ServantManager_var amref = am->_this();
nilampoa->set_servant_manager (amref);
```

By using different POAs to activate the auction server, the auction server can be made persistent. In the example, two different servants are used instead of using the same POA which requires to distinguish two objects while *etherealizing* or *incarnating*. At creation of the bid object (using *create* operation), the bid object is extended to activate itself with a specific Object ID which is used as the name for the *state file* on disk. A *shutdown* operation in the auction server interface is included to terminate the server process. This is accomplished by calling ORB's *shutdown* method. Invoking shutdown() on the ORB first of all causes the destruction of all object adapters. Destruction of the bid's POA next causes all active objects to be *etherealised* by invoking the servant manager. Consequently, the servant manager plays the major role to save and to restore the states. The auction house POA can be created as in the following code:

```
CORBA::PolicyList pl2; pl2.length(2);
pl2[0] = poa->create_lifespan_policy(PortableServer::PERSISTENT);
pl2[1] =poa->create_id_assignment_policy(PortableServer::USER_ID);
PortableServer::POA_var auctionpoa = poa->create_POA(char'Auctionchar', mgr, pl2);

/* * Create and activate an auction house */
Auction_impl * myaucthouse = new Auction_impl (nilampoa);
```

4.2 Naming Service

The CORBA services extend the core CORBA specification with a set of optional utilities that are useful for different applications. The CORBA naming service [3] is one of the simplest and the most useful utilities. Its role is to allow a name to be bound to an object and to allow that object to be found subsequently by resolving that name within the naming service.

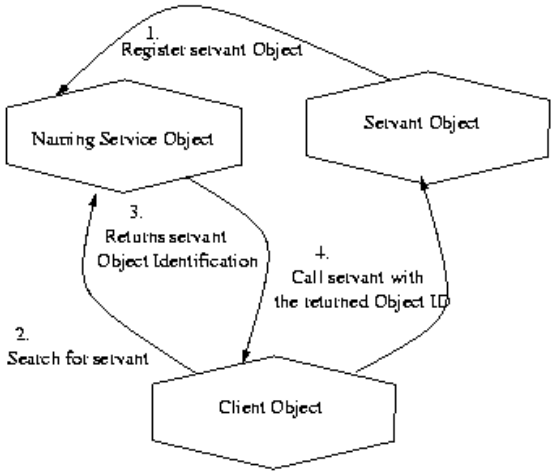

Fig. 5: Naming service invocation sequence

An auction server holds an object reference and registers it with the naming service, giving it a name that can be used by other components of the system subsequently to find the object. The CORBA object is the object which gives the reference of the remote object. The naming service invocation sequence is shown in Fig. 5.

One of the advantages of the naming service is that the names associated with objects are independent of any properties of the objects referred by them. In particular, a name is independent of an object's interface, server, or host name. In the case of a primitive *bind* operation, which is mainly vendor specific, for obtaining an object reference, it requires that the client knows the objects marker, server and host name. In contrast, finding an object using the naming service simply requires the caller to know the name that has been bound to the object. Successfully resolving a name within the naming service gives an object reference to the required object.

There are two ways in which an application can use the naming service. Firstly, the naming service can be used to name a significant number of objects in the system. Alternatively, some important objects in each service can be named, and these objects can act as entry points for the other objects. In the case of the auction service, the later is suitable for development. The auction house POA is registered with the naming service. The following code segment shows how a server registers its POA object with the naming service.

```
CORBA::Object_var nsobj = orb->resolve_initial_references(char'NameServicechar');
 assert (! CORBA::is_nil (nsobj));
  CosNaming::NamingContext_var nc = CosNaming::NamingContext::_narrow(nsobj);
 ...
 CosNaming::Name name;
 name.length(1);
 name[0].id=CORBA::string_dup (servername);
 name[0].kind=CORBA::string_dup(char'char');
  nc->bind(name, ref);
```

When the client invokes a POA object, the POA object in turns invokes the other servant objects. Now the client invokes a naming service to resolve the object reference for the servant object. When it gets the reference, it can directly invokes the methods of the object.

```
 CORBA::Object_var nobj = orb->resolve_initial_references(char'NameServicechar');
 assert (! CORBA::is_nil(nobj));
 CosNaming::NamingContext_var nc=CosNaming::NamingContext::_narrow(nobj);
 CosNaming::Name name;
 name.length (1);
 name[0].id = CORBA::string_dup (char'myNilamchar');
 name[0].kind = CORBA::string_dup (char'char');
 CORBA::Object_var obj;
 obj = nc->resolve(name);
 Auction_var auction = Auction::_narrow(obj);
```

The client bidder object creates a bid object in the auction server by calling a *create* to the auction house object. The auction house object returns a virtual object reference to the client object. Using this object reference, the client bidder object can directly communicate with the target object Nilam. The object management is done by POA manager nilammgr. The manager is responsible for the states of the objects, i.e. if the server has to shut down, the manager first calls *etherealise* to save the states of the object. When the server restarts, the manager calls *incarnate* to return the previous states of the objects. The underlying CORBA chosen for the implementation is Mico. The details of Mico is available in [10]. The asynchronous nature [13] of the bidding process for the sealed-bid auction will be discussed in the following section.

5.0 CONCURRENT PROCESSING AND SYNCHRONISATION

The implemented sealed-bid auction protocol uses both synchronous and asynchronous remote method calls. Fig. 6 illustrates the differences between synchronous and asynchronous processing of a cancelled order, for example. A synchronous process is a process, before processing the next process, has to wait for the acknowledgement of the responder. On the other hand, for an asynchronous process, the process continues without waiting for a response from the receiver of the message.

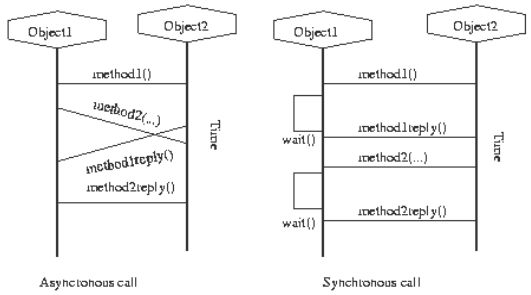

Fig. 6: Asynchronous process versus synchronous process

Synchronisation choices are crucial in the case of systems that support concurrent processing. In an auction, bidders can invoke two processes:

1. Submit a new bid.
2. Cancel an open bid.

The auction server can execute the following two processes simultaneously:

1. Route all open bids in the bid-table to the sharing program.
2. Process cancelled bids.

Therefore bidders can submit new bids, while the auction server routes open bids to the sharing program, which implements the sharing algorithm. If sharing is successful, the bid execution system updates the bid-table and notifies the corresponding bidder. These multiple processes use threads to execute concurrently. A new thread is invoked every time a bidder sends a request to the server. Threads require fewer system resources than computations. In a multi-processor workstation, multiple threads can operate simultaneously to take advantage of different processors. In a single processor machine, multiple threads can run in an interleaved manner so that different tasks run simultaneously. Thus the auction server can concurrently perform intensive computations for bid sharing, and at the same time, support interactive access. However, multiple threads are not protected; more than one thread can access the same data item. The most common way to implement concurrency control is to use exclusive locks. By locking the data, the application is in effect serialising access to the data. For example, when a bidder submits a bid, a new thread is launched at the auction server. The thread operation has three parts :

1. Assign the current BidID to the new bid.
2. Increment the BidID.
3. Add the bid to the bid-table.

Now suppose two bidders submit bids and the resulting threads interrupt each other:

1. Bidder A starts to submit a bid.
   Thread A executes Part 1 of the submitted bid.
2. Bidder B starts to submitted a bid.
   Thread B interrupts Thread A.
   Thread B executes Part 1 of the submitted bid.
1. Thread A interrupts Thread B.
2. Thread A executes Part 2 and Part 3 of the submitted bid.
3. Thread B finishes Part 2 and Part 3 of the submitted bid.

This scenario causes the two bids sent by bidders A and B to have the same BidID. We can solve this problem by adding the synchronised keyword to the *distribute share* method, distSHR(). This keyword serves as a mutually exclusive lock for the method, allowing only one thread to call the method. Upon completion of the method, the thread automatically releases the lock. Locks are useful if the portion of the data that must be serialised remains as small as possible. If unnecessary locks are applied, program performance becomes less efficient. For example, if a bidder cancels a bid right after it was routed to the sharing program, it is immediately deleted from the bid-table. The sharing program may then calculate and update shares, only to find that the bid has been deleted. To solve this problem, the bid-table is locked while matching is conducted, but this approach would freeze the bid-table constantly. Instead, asynchronous processing can be implemented. In a synchronous remote call, object A sends a

message to object B and waits for the feedback. Thus, the sending and receiving processes synchronise with every message. To continue, object A has to wait for the feedback from object B. Object B has to respond instantaneously to object A's request as well as to other remote calls. Otherwise, object A and other objects will be delayed while waiting for replies. CORBA implements asynchronous call by using *oneway* operation. If an IDL operation can be defined to be *oneway*, most implementations of CORBA will not block the caller of a *oneway* operation, but allow the caller to safely continue in parallel with the process of the request. A *oneway* operation must specify a *void* return type and cannot have *out* or *inout* parameters, and it also cannot have a *raises* clause [1]. With asynchronous communication, the server can schedule its operations more efficiently because it does not have to reply to each order immediately. Meanwhile, the client application does not have to wait for an immediate reply in order to conduct the next task. Asynchronous communication for cancelled bids is used because it does not need the bidding process to wait for the replies of the cancelled process. Using asynchronous communication, the cancelled requests are stored in a queue at the server side. After submitting the cancelled requests, the client application can proceed without waiting for the replies. The server side empties the cancelled queue each time before it restarts the sharing program.

The following shows the main algorithm of the cancelled process.

```
{
  store cancelled bidID in a cancelled queue;
  .
}
```

The main part of the sharing routine is as follows:

```
{
  ...
  if(sharing is completed and bidID found in cancelled queue)
  delete the bid from bid-table;
  remove cancelled bidID from cancelled queue;
  notify bidder;
 }
```

6.0 CONCLUSION

Our sealed-bid auction protocol uses verifiable secret and polynomial sharing algorithm. In this paper, we focus on the implementation issues of VSPS algorithm for the auction. We also focus on the implementation scenarios of the sealed-bid auction protocol over CORBA framework. The implementation techniques used for CORBA is portable object adapter (POA) whose specification is newly published by the OMG. The paper allows a reader to be familiar with the complex world of CORBA and its application. The total implementation is done in CORBA 2.3 implementation MICO-2.3. The platform used is Linux and implementation language used is C++ over IIOP. The multi-threads are implemented using POSIX compatible thread of Linux.

ACKNOWLEDGEMENT

The authors wish to thank Dr. Rosario Gennaro for his valuable suggestions in the implementation of VSPS scheme for auction service.

REFERENCES

[1] Sean Baker, *CORBA Distributed Objects.* Addison-Wesley, 1997.

[2] Rosario Gennaro, Michael O. Rabin, and Tal Rabin, "Simplified VSS and Fast-Track Multiparty Computations with Application to Threshold Cryptography", in *Seventeenth ACM Symposium on Principles of Distributed Computing, PODC'98*. ACM, 1998, pp. 101-111.

[3] *Interoperable Naming Service Preliminary Specification*, Technical Report, Object Management Group, October 1998.

[4] "The Common Object Request Broker: Architecture and Specification", *2.3 ed. Technical Report, Object Management Group*, June 1999.

[5] The LiDIA Group. *LiDIA - A Library for Computational Number Theory*. TH Darmstadt, Fachbereich Informatik, Institute fur Theoretische Informatik, Alexanderstr. 10, D-64283 Darmstadt, Germany, 1996.

[6] Michi Henning, "Binding, Migration and Scalability in CORBA". *Communications of the ACM*, Vol. 41, No. 10, October 1998.

[7] Michi Henning and Steve Vinoski, *Advanced CORBA Programming with C++*. Newblock Addison-Wesley, 1999.

[8] K. M. Yew, Mohammad Zahidur Rahman, and Sai Peck. Lee, "Specification of a Secure Fault Tolerant Pseudo-Anonymous Electronic Sealed-Bid Auction Protocol", in *MICC'99, IEEE Malaysia*, December 1999, pp. 218-223.

[9] Robert Orfali, *The Essential Distributed Objects Survival Guide.* John Wiley & Sons, 1996.

[10] Arno Puder and Kay Römer, *MICO - MICO is CORBA.* Morgan Kaufman Publishers, 1998.

[11] Douglas C. Schmidt and Steve Vinoski, "C++ Servant Classes for the POA". *SIGS*, Vol. 10 No. 6, June 1998.

[12] Douglas C. Schmidt and Steve Vinoski, "Using the Portable Object Adapter for Transient and Persistent CORBA Objects". *SIGS*, Vol. 10, No. 4, April 1998.

[13] Douglas C. Schmidt and Steve Vinoski, "Programming Asynchronous Method Invocations with CORBA Messaging". *SIGS*, Vol. 11 No. 2, February 1999.

[14] J. Siegel, *CORBA: Fundamentals and Programming*. John Wiley & Sons, New York, 1996.

BIOGRAPHY

Sai Peck Lee is currently an Associate Professor at Faculty of Computer Science & Information Technology, University of Malaya. She obtained her Master in Computer Science from University of Malaya in 1990, her D.E.A of Computer Science from University of Paris VI Pierre et Marie Curie in 1991 and her Ph.D. in Computer Science from University of Paris I Panthéon-Sorbonne in 1994. Her current research interests include Software Engineering, Object-oriented Methodologies, Software Reuse, E-Commerce, Information System and Database Engineering.

Zahidur Rahman, M. is currently an Associate Professor at Department of Electronics and Computer Science, Jahangirnagar University, Savar, Dhaka, Bangladesh. He obtained his B.Sc. Engineering in Electrical and Electronics from Bangladesh University of Engineering and Technology in 1986 and his M.Sc. Engineering in Computer Science and Engineering from the same institute in 1989. He obtained his Ph.D. degree in Computer Science and Information Technology from University of Malaya in 2001. His Ph.D.'s thesis work is on designing a secure protocol for electronic commerce transactions. His current research includes the development of a secure distributed computing environment using formal method for E-commerce.