A Scalable Architecture for Responsive Auction Services Over the Internet

A. Amoroso and F. Panzieri

Technical Report UBLCS-2003-09
June, 2003

Department of Computer Science
University of Bologna
Mura Anteo Zamboni 7
40127 Bologna (Italy)
The University of Bologna Department of Computer Science Research Technical Reports are available in gzipped PostScript format via anonymous FTP from the area ftp.cs.unibo.it:/pub/TR/UBLCS or via WWW at URL http://www.cs.unibo.it/. Plain-text abstracts organized by year are available in the directory ABSTRACTS. All local authors can be reached via e-mail at the address last-name@cs.unibo.it. Questions and comments should be addressed to tr-admin@cs.unibo.it.

Recent Titles from the UBLCS Technical Report Series

A Scalable Architecture for Responsive Auction Services Over the Internet

A. Amoroso\textsuperscript{2} and F. Panzieri\textsuperscript{2}

Technical Report UBLCS-2003-09
June, 2003

Abstract
In this Report we discuss the design and implementation of a distributed architecture we have developed in order to support "responsive" (i.e., timely and available) auction services over the Internet.

Specifically, our architecture can support auction services whose timing requirements fall within the range of a few seconds. In addition, this architecture can adapt timely to possible variations of the network conditions (e.g., congestion, load).

In order to provide the users with highly available auction services, our architecture allows one to implement those services by means of replicated servers over the Internet. Those servers periodically coordinate by exchanging messages, so as to maintain a mutually consistent view of the auction state.

The implementation of an auction service based on our architecture does not require any kind of external clock synchronization; rather, our architecture implements a time-aware service model that can tolerate both server and message failures, as well as network partitions.

\textsuperscript{2} Dipartimento di Scienze dell’Informazione, Università di Bologna, Mura Anteo Zamboni 7, 40127 Bologna, Italy.
1 Introduction

In recent years a number of auction services have been made available over the Internet (e.g., www.ebay.com, www.antiquorum.com, www.artnet.com, to name a few). A common feature of these services is the considerable amount of time they require to complete the auctioning process. Typically, a user of these services can submit a bid and, only after an amount of time that can range from hours to days, that user knows whether her/his bid has been accepted [38, 59, 40]. This feature is a consequence of the asynchronous nature of the IP based, best-effort communication service provided by the current Internet. Namely, this service allows applications neither to reserve bandwidth, nor to exercise control over the network resources (i.e., no admission control policy is implemented); thus, the communication delays over the Internet are unpredictable (hence the use of the term asynchronous above). Owing to the asynchronous nature of the IP communication service, timeliness requirements can hardly be met over the Internet [60, 48, 46, 47, 24].

In addition, current Internet-based auction services rely, in general, on centralized auction server architectures. These architectures exhibit a number of limitations, including the following. Firstly, a centralized architecture cannot deal adequately with issues of service availability and scalability. Typically, such an architecture can be vulnerable to server’s failures, if not equipped with sufficient redundancy; in addition, server’s overloading may occur, if an arbitrary large number of users concurrently access the service. The increasing number of customers of Internet-based auction services suggests that both these issues are crucial in the design of those services. In particular, as pointed out in [47], service availability is required as a frequently unavailable service may discourage users from using it, and result in a business loss for its provider. The service scalability is necessary as an auction service is expected to provide all its users with an equally satisfactory (and fair) service, regardless of the number of those users and their geographical location.

Secondly, an Internet-based auction service must be accessible to users that are distributed, at least in principle, on an international, possibly planetary, scale; thus, that service may have to deal with different national selling rules that pertain to individual countries. As pointed out in [24], within this scenario a centralized architecture may turn out to be inadequate, as a great deal of complexity may have be incorporated in the centralized auction server, in order to deal with those different selling rules.

Needless to say, such requirements as timeliness, availability and scalability, introduced above, are only a subset of those that are to be met in the design of an effective auction service; additional crucial requirements include, security, privacy, anonymity [30], and fairness [8], for example. However, for the purposes of our discussion, we shall focus on issues of responsiveness (i.e. availability and timeliness) and scalability.

Owing to the above observations, in this Report we propose an architecture for supporting auction services over the Internet that is based on replicating the service across a number of auction servers distributed over this network.

In essence, within our architecture, the auction servers cooperate as in a periodic real-time system, as proposed in [47]. Specifically, an auction is organized as a sequence of one or more rounds. Within each round the auction servers collect bids from the participants, asynchronously from each other, and evaluate the locally received best bid (defined later). Periodically, the servers synchronize in order to reach consensus on the received best bid in the current round, and to maintain consistently what we term the shared auction state. This shared state information includes the item on sale, the best bid received for that item, the identifier of the client that submitted that bid (further details on the shared auction state will be introduced later).

Following each synchronization phase, the servers resume the bid collection process, asynchronously from each other, until the next synchronization phase. The auction terminates during a synchronization phase in which specific auction termination conditions are met (see next Section).

Note that, in our architecture, the servers do not need access to any kind of global timing service, for the purposes of the periodic synchronization mentioned above, as it can be shown that the servers’s local physical clocks are sufficiently accurate to maintain the required synchroniza-
tion among them. Rather, a global timing service is required for initialization purposes, only. To
this end, we use the NTP timing service currently available over Internet [44].

In addition, we circumvent the unsolvable “consensus” problem [29] by assuming the fail
aware datagram service proposed in [28].

In this Report we focus our discussion on the servers synchronization protocol, as it is the
core of our architecture. However, for the sake of completeness, we wish to point out that, in
order to guarantee consistency of the shared auction state, the client-server interactions need to
be structured as atomic actions; i.e., it must be guaranteed that the bid submission operation
invoked by a client either terminates by correctly delivering the client bid to an auction server, or
it has no effect (in either cases, the invoker of the operation is notified of the termination of the
operation). A protocol such as that described in [41] can be used, so as to ensure atomicity of the
client server interactions.

In summary, the principal contribution of this Report is to describe a distributed architecture
for the implementation of responsive auction services which can scale (i.e., maintain its respon-
siveness property) effectively so as to accommodate an arbitrary number of clients. This architec-
ture is based on a simple synchronization protocol among distributed auction servers that carry
out their activity asynchronously from each other, and periodically coordinate, as in a periodic
real time system.

This Report is structured as follows. In the next Section we introduce the most common types
of real-life auctions. Section §3 discusses the principal design issues we have addressed. Section
§4 describes the architecture we propose, and discusses its properties. Section §5 summarizes a
prototype implementation of our architecture we have carried out. Section §6 examines related
work. Section §7 provides some concluding remarks. Finally, we include in the Appendix the
formal proofs of the theorems introduced in the discussion of the properties of our architecture.

2 Types of auction

The open-cry auction (also called “English Auction”), the sealed-bid auction, and the Dutch auction
represent the most common types of auctions in the real-life [40, 47, 59]. (Variations of these
three types of auctions are also possible, e.g. the “Vickrey” auction, the “Discriminative” and
“Non Discriminative” auctions; however, for the purposes of this Report, we shall only consider
the three types of auctions mentioned above).

Each of the above three auction types can be thought of as consisting of the following three
principal entities. The goods to be auctioned. The participants who are interested in bidding for
those goods. An auctioneer, who is responsible for providing the institutional setting of the auc-
tion, managing the auction progress, and terminating the auction by selecting the winning bid,
according to some predefined rule. In practice, goods at auction belong to vendors; however,
for the purposes of our discussion, we shall assume that the auctioneer represents the vendors.
Hence, auctioneer and vendors are considered as a single entity. An auction can be either one-
sided, or two-sided. In the former case, a single auctioneer offers goods to a community of par-
ticipants. In the latter case, any participant, in addition to placing bids for the goods at auction,
can he himself assume the role of auctioneer by offering goods to the other participants, as the
auction progresses.

A typical example of two-sided auction is the so-called double auction, used in the stock ex-
change market. In this Report, we consider one-sided auctions, only; thus, the architecture we
propose is not intended to support double-auctions. A distributed architecture that supports this
type of auction is described in [43].

The open-cry, sealed-bid, and dutch auctions can be classified according to the following four
principal characteristics:
1. the location where the auction is to be held;
2. the start-up rule, which dictates the initialization rules of the auction (e.g. the time and place
   where the auction is to be held, possible registration requirements, and so on);
3. the auction progress rule, which states both whether the items on sale by auction will be auctioned either in a single round or in multiple rounds, and the time extent of each round, and, finally,
4. the termination rule, which states the condition by which the received bids can be resolved, and the winning bid declared, thus terminating the auction.

In the following, we examine these three types of auctions in isolation, in view of the four characteristics itemized above.

2.1 Open-Cry
1. Location: the auctioneer and the participants gather in the same location at a pre-specified time;
2. Start-up: the auctioneer enables the auctioning process by setting an asking price for an item on sale, and requesting bids from the floor for that item;
3. Auction progress: periodically the auctioneer resets the asking price to the value of the highest received bid, and starts a new round of bids;
4. Termination: the auctioning of an item terminates when no more bids are submitted for that particular item within a round. The item is assigned to the bidder that submitted the highest bid.

Note that, in a real-life open-cry auction, both the number of rounds and the duration of each round, required to complete the auctioning of an item, cannot be predicted. In addition, each participant is aware of all the bids submitted by the other participants, as all the participants can see and hear each other; thus, each participant can decide her/his own bidding strategy, based on that information record. Moreover, all the participants share the same information on the items on sale by auction, and on the auction progress. Finally, they have the same perception of time, and experience the same communication delays.

2.2 Sealed-Bid
1. Location: there is no a location where the auctioneer and the participants gather at the same time. Rather, the participants are made aware of a location where bids should be submitted within a deadline;
2. Start-up: the auctioneer enables the bidding process by setting a deadline within which bids must be delivered at the above location;
3. Auction progress: a sealed-bid auction can be organized either as a single round, or as a multiple round auction. In a single round auction, each participant submits a bid; each submitted bid is kept secret until the deadline, set by the auctioneer, expires. Then, the auctioneer resolves the received bids, and the auction terminates (according to the termination rule below). In contrast, in a multiple round sealed-bid auction, the auctioneer evaluates the received bids at the end of each round. If none of these bids meets the termination condition, the auctioneer starts up a new round, and defines a new deadline (at the end of each round the received bids may or may not be made public);
4. Termination: the auctioneer resolves the received bids, and declares the best received bid (see below) as the winning bid.

Note that, in sealed-bid auctions, the “best” bid does not necessarily coincides with the “highest” bid received by the auctioneer. In some cases, the “best” bid may be the lowest received bid, as in the case of a contract for services, for example; in other cases, the “best” bid may be the closest bid to a secret target, as, for example, in the case of building contracts.

2.3 Dutch
1. Location: the auctioneer and the participants gather in the same location;
2. Start-up: the auctioneer sets a (possibly high) asking price for the item on sale by auction, and enables the submission of bids;
3. Auction progress: the auctioneer decrements the asking price periodically, until a buyer submits a bid for the item on sale that matches the current asking price for this item;
4. **Termination:** the auction terminates when either a bid is accepted as it matches the current asking prices, or the price of the item falls below a predefined threshold, and the item is withdrawn from the auction (possibly temporarily).

Note that the Dutch auction is typically used to sell perishable goods (e.g., fish, agricultural products, last minute flight tickets). In this type of auction, buyers can bid for product lots that are offered by different sellers. The sellers are represented by a sales manager (i.e., the auctioneer). Owing to the very nature of the items on sale by auction, the duration of each round is typically very short, e.g., a few seconds. As in the open-cry auctions, the participants are aware of the progress of the auction.

### 2.4 Real-life vs. Internet-based auctions

It can be observed that the three auction types introduced in the previous Subsection differ principally in their termination condition. However, these auction types share the following common characteristic: goods are sold in rounds, governed by a clock; specifically, in each auction type a round consists of the time interval within which a bidder can submit a bid for an item.

Although the time extent of the rounds in the various auctions types may differ notably from each other (e.g., a round in the dutch auction may last a few seconds; instead, in a sealed-bid auction, it may last hours or days), the common characteristic pointed out above suggests that an electronic auction system can be adequately modelled as a time-triggered real-time system [39] that periodically displays objects on sale, collect bids for those objects, and resolves them in order to declare a winning bid (if any).

This apparently simple model can be implemented by a distributed architecture that uses the Internet as its communication infrastructure, as discussed in Section §4. Further observations on real-life and Internet-based auctions are in order.

In Internet-based auctions the participants are subject to the same rules as in a real-life auction. Hence, in Internet-based auctions the participants must be provided with an abstraction that approximates real-life auctions as much as possible; thus, for example, the implementation of an Internet-based auction must maintain consistently the auction state (i.e.: goods on sale, current best bid, deadline to place a winning bid) that can be displayed to the participants. Note that in the Internet-based auctions, both the auctioneer and the participants may not be human beings; rather, they can be programs that implement the selling rules and the buying strategies, respectively, on behalf of humans [59].

In both traditional and Internet-based auctions, it is possible for the participants to inspect the items on sale by auction before the auction begins. In the Internet-based auctions, these items can be shown through pictures, movies, or some virtual reality objects. (Needless to say, in some cases the inspection of the items on sale by auction can be irrelevant, owing to the very nature of these items, as in the case of airplane tickets for specific destinations).

Internet-based auctions can be characterized further as follows. Firstly, in order to enable the largest possible number of participants to take part in an auction, the starting time of that auction is to be announced in ample advance. In an Internet-based auction, this announcement can be easily disseminated all over the planet. This announcement may well include the deadline by which bids are to be delivered to the auctioneer. This deadline may coincide with the end of the first auction round (e.g., in an open-cry auction), or with the end of the auction itself (e.g., in a single-round sealed bid auction).

Secondly, depending on the auction type, the auction announcement may include the asking price for each of the various items that will be sold by auction.

Thirdly, participants can join a real-life auction already in progress; in an Internet-based auction, this entails that those participants are to be made aware of the current auction state, in order to be enabled to take part to that auction.

Fourthly, each type of auction is characterized by both a specific time duration of the rounds in which the auction is structured, and a particular evolution of the selling prices of the items on sale by auction. For example, the time duration of a round in an open-cry Internet-based auction can be set to a few minutes (e.g., 2 to 3 minutes), and the price of the item increases as the auction progresses. In contrast, a round in a Dutch auction can last a few (e.g., 10 to 20) seconds, and
the item asking price tends to diminish as rounds progress. Finally, a round in a multiple round sealed-bid auction can last a few days, and the asking price of the item at auction may or may not vary depending on the specific auction policy.

According to [6], Internet-based auctions introduce some variation to the traditional auction models proposed in literature. For example, rather than focusing on a single item, an auction over the Internet can sell multiple identical units of an item, using a mechanism analogous to the Open-Cry. The termination rules of Internet auctions may differ from the traditional real-life ones. In some auctions over the Internet the auctioneer typically terminates the auction if the predefined closing time is passed and there were no new bids in the last few minutes; this behavior is usually called Yankee Auction. In addition, usually the Internet-based auctions allow the participants to put new bids with a minimal increment.

3 Design issues

The principal issues we have addressed in the design of our architecture include availability, timeliness, and scalability; these issues can be summarized as follows.

*Availability* In order to meet the Internet-based auctions availability requirement, mentioned earlier, we propose an architecture that consists of a number of replica servers, distributed across the Internet, that implement the auction service. Provided that these replica servers are maintained mutually consistent, service availability can be obtained by distributing the client requests across these servers. Moreover, in case a replica server fails to deliver its expected service (e.g. owing to a server crash or overload, or to a link failure, or network congestion) the load of that replica server can be allocated across its available replicas, at runtime, thus maintaining the auction service availability. Note that our architecture can tolerate failures such as server crashes, message loss, and network partitions, as described later; instead, issues of load distribution fall outside the scope of this Report (the interested reader can refer to a vast body of literature in this field, including, for example, [23, 14, 32, 12]).

*Timeliness* Meeting the timeliness requirement, in our architecture, means providing auction clients with a response time of the same order of magnitude as that of real-world auctions. To this end, we assume that each client can be bound to its “most convenient” replica server (e.g. the most lightly loaded auction server, or the replica server with the least congested path to the client). Binding clients to their relative most convenient replicas can be achieved by implementing such protocols as those described in [17, 18, 51] (these protocols fall outside the scope of this Report; hence, we shall not discuss them further).

*Scalability* Scale is a primary factor that can influence the design and implementation of a distributed system [54, 19]. In particular, mechanisms that work adequately in small distributed systems may fail to do so when deployed within the context of larger systems. For the purposes of our discussion, we term scalable a system that can provide its services, according to the performance and reliability specifications of those services, regardless of both the number of resources (i.e., servers, networks), and clients it accommodates.

Specifically, we shall assume that an auction house will make use of some Application Service Provider (ASP) that hosts and runs the auction application. Thus, the auction house will have negotiated and established some Service Level Agreement (SLA) [2] with the ASP so as to dictate QoS requirements such as bandwidth and latency, as well as service availability, service response time, and number of transactions per second. Based on that SLA, the ASP will be able to size appropriately the system architecture necessary to host the auction application (thus deciding, for example, the maximum number of replica servers that can be required in order to honor the SLA). Within this scenario, we claim that our architecture can scale effectively, as it can accommodate an anticipated (within the SLA) maximum number of both clients and replica servers, and yet maintain its responsiveness.
3.1 Infrastructure support

Our design approach is based on the Timed Asynchronous Distributed System Model (TADSM), proposed by Fetzer and Cristian [22]. The communication model we adopt is the fail-aware datagram service proposed by the same authors in [28].

In TADSM a message has a minimum and a maximum transmission delay, $\delta_{\text{min}}$ and $\delta_{\text{max}}$ respectively, and the processing speed of each process is bounded within a known interval.

In addition, the physical clocks have a known drift rate, and a process can read the physical clock of the computer where it is running. The clocks in a distributed system based on the TADSM may be not synchronized, and they are used to measure local time intervals.

Finally, in TADSM, a process may suffer two types of failures; namely, a crash failure and a performance failure. A crash failure occurs when a process works correctly until a certain instant, and then stops working. A performance failure occurs when, in order to accomplish a task, the process takes too much time with respect to a predefined timeout.

3.1.1 Fail-aware datagram service

A fail-aware datagram service provides its users with primitives for sending and receiving messages, and allows receivers to detect performance failures.

In the TADSM a message is uniquely identified by the sender, the receiver, the content, and the sending time. Specifically, in this model, a message can be classified as follows, according to its transmission delay $\delta$:

i) $\delta < \delta_{\text{min}}$: the message is early;
ii) $\delta_{\text{min}} \leq \delta \leq \delta_{\text{max}}$: the message is timely;
iii) $\delta > \delta_{\text{max}}$: the message is late;
iv) $\delta = \infty$: the message is lost.

Hence, if a message is early or late, it is said to suffer a performance failure; if it is lost, it suffers an omission failure.

The value of $\delta_{\text{max}}$ affects the performance of the system. A too small value of $\delta_{\text{max}}$ may cause a growth of the number of performance failures. A value of $\delta_{\text{max}}$ which is too large may cause a reduction of the quality of service of the system. The value of $\delta_{\text{min}}$ may be assessed by means of the physical characteristics of the network, or by means of measurements. It is legitimate to assume $\delta_{\text{min}} = 0$.

3.1.2 JGroup

In order to implement our architecture we require an interprocess communication mechanism that, based on the fail-aware datagram service mentioned above, provides us with group communication support over a wide area network. Specifically, we require that the group communications support provides us with a reliable multicast mechanism and deal effectively with possible network partitions.

Current off the shelf technology, such as CORBA, JBoss, and .NET, meet only partially these requirements. Hence, for the purposes of this work, we have decided to use JGroup, a prototype middleware layer that supports group communications in asynchronous, wide-area networks [4, 45]. JGroup embodies a number of sophisticated communication capabilities, including the following three, which are relevant to our design.

- **Group semantics**: the processes can link together to form a group. JGroup installs views, that correspond to the process’s local perception of the group’s current group membership.
- **Reliable multicast**: a process in the group can send a message to all the group members. In essence, the reliable multicast protocol guarantees that in the absence of failures, all group members receive the same set of multicast messages, and that any multicast message will be eventually delivered to all the group members. This requirement cannot be met if a network partition occurs; in this case, the reliable multicast guarantees that processes within the same partition will receive the same set of messages.
- **Partition awareness**: if failures occur that partition a group, then the processes in the group partition install identical views.
3.2 Failure model

We consider the following three classes of failures: process failures, message failures, and network partitions. Process failures are as defined in the TADSM introduced in Subsection §3.1.1. Message failures and network partitions are discussed in the following two Subsections.

We do not include arbitrary (i.e. byzantine) failures in our model as they fall outside the scope of our current investigation on service scalability and responsiveness. (Techniques for coping with arbitrary failures in Internet-based systems are a topic for our future research).

3.2.1 Message failures

Following [34], we assume that a communication link can suffer from crash and omission failures. Moreover, we assume that messages can be affected by performance failures, as defined in Subsection §3.1.1. We say that a link has crashed when that link stops transporting messages, and before it crashed it behaved correctly. If a faulty link intermittently omits to transport messages, we say that the link suffers omission faults. (In addition, a link may suffer performance failures, as introduced in the previous Subsection §3.1.1).

We do not include unidirectional omission faults in our model, i.e. it cannot occur that a process \( P_i \) can receive messages from process \( P_j \) and that \( P_j \) cannot receive messages from \( P_i \).

Crash and omission failures are managed by the JGroup communication layer. When JGroup cannot send a message through a communication link, even after a sequence of repeated attempts, it tries to send the message through alternative communication paths. If those attempts fail, JGroup defines the destination process as unreachable, recognizes a network partition (see below), and raises a partition exception to the application.

As mentioned earlier, in our system a message may suffer performance failure. As JGroup can only deal with omission, crash, and network partition failures, we need a mechanism to detect performance failures. To this end, a performance failure detector has been introduced in our architecture (see Section §4.4).

3.2.2 Network partitions

A group of servers can suffer communication failures that cause partitioning of that group. The servers in the same partition can communicate with each other, while they cannot communicate with the servers in other partitions, even if those are correctly running. This scenario may occur when the network is disconnected or it is heavily loaded and any process belonging to the same partition can exchange timely messages. Partitions can be temporary; hence, the auction system should be able to deal with both the partitioning of the servers group and the merging of partitions.

A partition is stable if any process within a partition can communicate timely with the other processes in that partition, while any communication with processes outside that partition results in an omission or performance failure.

4 Responsive auction architecture

The architecture we propose consists of a number of geographically distributed servers, interconnected via the Internet, that implement auction services such as those introduced earlier. An arbitrary number of clients can make use of these services by connecting to any of these servers, yet again via the Internet.

For the purposes of our current discussion, we assume that the date and time an auction starts can be advertised sufficiently in advance to enable clients to participate; however, our architecture allows clients to join an auction in progress at any time.

In addition, we assume that each client gets bound to the most responsive server, relative to the client itself. Specifically, the binding between a client and its most responsive server is carried out transparently to that client.

Techniques for transparently binding clients to servers fall outside the scope of this Report; hence, we shall not discuss them here. However, for the sake of completeness, we wish to point
out that, as part of a separate research activity, we are investigating particular techniques that allow both transparent and dynamic binding of clients to their most responsive server [17, 18, 32, 51].

Figure 1 illustrates an example of our architecture. In this figure, five auction servers are distributed in five geographically distant locations. Each client in Figure 1 is connected to its most responsive server.

The auction servers are structured as a group of peer processes that deploy the techniques described in [11, 21, 27, 28] in order to deal with possible network partitions within the group.

The solid arrows in Figure 1 represent the communications within the servers’ group. These communications are based on the reliable multicast protocol introduced earlier, extended as described in the following sub-sections. These extensions of the reliable multicast protocol guarantee that, in the absence of failures, messages exchanged within the servers’ group are delivered timely.

The communications between the clients and the auction service, represented by the grey arrows in Figure 1, are governed by the atomic interaction protocol mentioned earlier.

The servers’ group maintain the auction service abstraction by providing the clients with the same, correct view of the auction progress. To this end, the auction servers maintain consistently a shared auction state that includes the item on sale, and the current best bid for that item.

Note that the architecture we propose is fully decentralized. Specifically, as the servers are peer processes, the possibility that there be a single point of failure within the servers’ group is ruled out. In particular, a server crash, when detected, can be dealt with by (i) binding the clients of the crashed server to an alternative, active, and sufficiently responsive server within the group, and (ii) reconfiguring the servers’ group so as to exclude the crashed server from the group. In addition, our architecture does not include a specific centralized component, such as the auctioneer in a real-life auction, that masters the auction progress. Rather, in our auction system, clients can observe the auction progress through the shared auction state, maintained by the auction servers.

In order to maintain the auction state consistently, the servers periodically coordinate by exchanging messages, as depicted by the black bidirectional arrows in Figure 1. The periodic synchronization among servers is enabled by a server synchronization protocol, described later, based on the TADSM [22].

4.1 Auction life cycle

Our architecture operates as a periodic real-time system. Specifically, in the absence of failures, it operates as follows.
The auction process proceeds in consecutive \textit{rounds}, once the auction has started (the starting of the auction is described later). Rounds may not be of the same duration; rather, the duration of each round is agreed upon by the servers at the time the periodic synchronization among them occurs.

Each round of the auction is divided into two phases: a \textit{bid collection phase} and a \textit{servers synchronization phase}. During the bid collection phase the servers accept bids from the participants. At the expiration of the collection phase, every server enters the synchronization phase, selects the local best bid\footnote{As previously mentioned, the “best” bid does not coincide necessarily with the highest bid.}, and broadcasts it to the group of servers. When a server has received all the broadcast best bids, it computes the best offer for the current round, and elects the server that proposed it as the “leader” for the next round; finally, the server waits for a communication from the leader. Note that, owing to the reliable multicast communication service, all the servers receive the same set of broadcast messages; hence, all the servers elect the same leader. The leader server computes the duration of the next round, and broadcasts it to the servers in the group by transmitting them a so-called \textit{auction state message}. When a server receives the auction state message, it starts the next round, i.e. the collection phase. The leadership may change round by round, but there is only one leader at any one time.

Figure 2 illustrates the whole life cycle of an auction, as implemented by our architecture. This figure depicts the various phases of our auction life-cycle, including the start and termination phases.

It is worth observing that time synchronization over the Internet may be complex and costly, and may require a time-server external to the system. The time accuracy depends on the auction duration; in long duration auctions, such as a sealed-bid auction that may last several days, the time synchronization among the servers may not be strict, except for the start and termination of the auction. Within this scenario, we can assume that the asynchronous property of the Internet, and the non-predictability of both the servers schedule and the message latency, do not significantly affect the auction correctness as the message delays are orders of magnitude smaller than the auction duration.

Time synchronization problems may arise in multi-round auctions where the round duration can be of the order of magnitude of minutes. In this type of auctions, where the timing requirements are strict, a network error may jeopardize the whole auction. In our auction model, a synchronization on a global time of all the servers clocks is only required for a correct starting time of the auction. When the auction is in progress, the servers do not need to agree on a common global time, they simply need to agree on the duration of each round of the auction. As the expected drift rate for standard computer clocks is smaller than or equal to $10^{-6}$ [13, 22], we can assume that during a round any two server clocks are sufficiently \textit{precise} [7], for all practical purposes.
Finally, both clients and servers processes can tolerate possible message losses. Specifically, a server process is shielded from message losses as the JGroup communication support it uses installs a new view for that process when a message loss is suspected. In contrast, if a client process suspects the loss of one of its bid messages to an auction server (possibly after a number of retransmissions of that bid message), it submits that bid to a different server.

The various phases of the auction progress are discussed below, in isolation.

4.1.1 Start-up
We assume that the start-up time of an auction is advertised sufficiently in advance to allow the servers to set up the service group, and synchronize, prior to the start of that auction. Note that the startup time is the only situation where the clocks of our system need to be synchronized with an external clock service.

The mechanism that the servers use to synchronize the auction service group is similar to the one they use during the auction process. When the servers start their synchronization, they transmit to each other the best bid message containing a virtual bid representing the starting price of the goods to be sold. Since all the bids are equal, at start-up time, the servers need only to identify uniquely the leader according to a predefined criterion (e.g., the server with the smallest id in the system can be elected “leader”, at start-up time).

At the time an auction is due to start, participants to that auction are expected to be connected to their most convenient server. Thus, these participants can be notified of the item on sale, its current price, and a deadline by which they can place a bid. This latter information coincides with the duration of the collection phase of the first round, as agreed by the servers. During the first phase of the round, the servers collect the bids from the participants.

4.1.2 Progress
Each bid must be an atomic transaction to guarantee both the bidder and the server about the registration of the bid, and its arrival time. All the bids received by a server in a round are evaluated in that round as the deadline of the collection phase expires. If a bid is delivered after the deadline, it will be evaluated in the next round.

When the deadline of the collection phase expires, the synchronization phase among the servers begins. Figure 3 shows an example of a synchronization phase when \( N = 3 \). Every server \( S_i, i = 1, \ldots, N \), computes the best locally received bid, say \( b_i \), and multicasts it to all the members of the group; moreover, \( S_i \) records the sending time \( ST_i \) of the transmission of \( b_i \). When a server \( S_i \) receives the best bid from the server \( S_j \), it records the arrival time of the message, say \( AT_j \). When a server has received all the best bids from the other servers, it computes the best bid for the actual round, called \( bh^k \), where \( k \) is the sequential number of the round. Based on the information received, every server recognizes as leader the server which sent the best bid of the round. Since all the servers delivered the same set of messages, due to the properties of the reliable multicast protocol they use, all of them agree on both the best bid and the leader.

4.1.3 Termination
As introduced earlier, the auction termination rule depends on the type of the auction. In any case, to sell an item it is crucial that no failures occur during the best bid exchange phase. If a failure occurs, some bid, better than the one currently assumed to be the best bid, might be ignored.

In our system, a failure in the synchronization phase is detected and dealt with as a partition. The action to be performed inside each partition depends on specific auction policies. A simple policy may be to abort the auction and restart it later, or to suspend the auction until all the partitions merge.

An alternative policy may be to continue the auction in parallel in each partition, and compute the best bid, when the partitions merge. For fairness, the item on sale cannot be sold until all the partitions merge. It depends on the auction policies how long the auction can proceed in separate
partitions; the more it proceeds in parallel, the less all the participants share the same auction progress.

A further alternative approach can be based on dynamically binding the clients of a partitioned away server to one of the remaining servers of the group. This allows the users to continue the auction, regardless of the server to which he/she is connected. The same mechanism can be used to face server crashes that cannot be masked.

### 4.2 Leader election

When a server $S_l$ recognizes itself as the current leader, it computes the duration of the next bid collection phase, $\tau$. The leader computes $\tau$ in order to keep the duration of the current round as close as possible to a predefined value. The leader multicasts to all the servers in the group the auction state message including $\tau$, the sending time $SST$ of this message, and the arrival times of all the message it has received, $AT_j, j = 1, ..., N : j \neq l$. The auction state message contains all the $AT_j$'s as these are necessary to the servers in order to approximate the local duration of the next bid collection phase, by means of the mechanism presented in Subsection 4.3 below. (Note that, even if each server $S_l$ needs only the $AT_j$, the leader includes all the $AT_j$ into the auction state message in order to multicast a single message to all the servers in the group).

Figure 3 shows a space-time diagram illustrating an example of the message exchange among the servers, during the synchronization phase in a system with three servers. In that example the server $S_2$ becomes the leader, represented with the crown icon, and sends the auction state message to the other servers. The figure shows all the timestamps taken and exchanged during the synchronization phase. The large dots on the process lines represent the time recording of sending and receiving messages.

When a server $S_l$ receives the state message from the leader, it records the arrival time of this message, say $SAT_l$. Since the arrival time of the auction state message is peculiar to each server, as it depends on both the message delay and the server clock, we denote the $SAT_l$ with the server identifier $i$. Note that every server can neither measure the message delay $\delta_{b_i}$, experienced by the communication of the best bid $b_i$, nor the delay $\delta_{s_i}$, experienced by the auction state message that it received, as the send and receive times of these two messages are measured with different clocks. Instead, a server $S_l$ can measure the local duration of the synchronization phase, i.e. the time interval between $ST_l$ and $SAT_l$, with its own clock. The duration of...
the synchronization phase is the sum of the two message delays, δ_{b_i} and δ_{s_i}, and the computation time, L, needed by the leader to recognize itself as the leader and to compute τ. Note that L = min(SST - AT_i : i = 1, ..., N; i ≠ l) that and it is measured by means of the leader’s clock.

Finally, note that more than one server may receive a best bid of the same amount during the same round. In this case a simple rule can be used to elect the leader; e.g., the leader is the server with the smallest identifier.

4.3 Server Synchronization

While it is impossible for a server to measure the message delay experienced by the communications with the leader, during the actual synchronization phase, each server can approximate this delay by means of the mechanism proposed in [20]. A server cannot approximate separately the two delays δ_{b_i} and δ_{s_i}; however, it can compute their sum, and then approximate each one of them to the half of that sum. Hence the approximation δ_i computed by the server S_i of both δ_{b_i} and δ_{s_i}, is as follows:

$$\delta_i = \frac{\delta_{b_i} + \delta_{s_i}}{2} = \frac{(SST - AT_i) - (SST - AT_j)}{2}$$

The sum of the delays is measured using consistent timestamps of both the server and the leader clocks, and assuming the effects of the drift of those clocks to be negligible with respect to the synchronization phase duration. As an example, consider the server S_1 in Figure 3. The timestamps ST_1 and SAT_1 are generated by observing the S_1 server clock; then, the duration of the interval SAT_1 - ST_1 is computed on consistent clock values. The duration of the interval SST - AT_1 is obtained by observing the leader clock values. Thus, the difference between the two intervals is the sum of the messages delays δ_{b_1} and δ_{s_1}.

Using the approximated value of the message delay, every server can compute the local duration τ_i of the current collection phase, that should simultaneously terminate in all the servers. Therefore every server S_i computes τ_i = τ - δ_i, and sets the deadline γ_i^k of the collection phase to γ_i^k = SAT_i + τ_i. Due to the accuracy of the approximation, the termination of each collection phase at every server cannot be strictly synchronized. We call server synchronization skew, Γ^k, the maximum difference between all the γ_i^k in the k-th round; that is, Γ^k = max(|γ_i^k - γ_j^k| : i = 1, ..., N ; j = 1, ..., N). Since Theorem1 (see Appendix) shows that Γ^k ≤ (δ_{max} - δ_{min}) for any round k, the above mechanism allows the servers clock to be synchronized, within a predefined constant time interval, at any round.

As shown in Corollary 1 and Lemma 3, in the Appendix, we can characterize the responsiveness of our auction system with respect to the network performance; i.e., we can bound the “speed” of an auction based on the actual network latency. In particular, the minimal duration of each collection phase and that of each round can be expressed in terms of δ_{min} and δ_{max}. These observations are the basis for the local computation of the timeout values required for performance failure detection purposes, as shown in the next Section.

To conclude this Subsection, we wish to mention that we have carried out a preliminary experimental evaluation of the overheads entailed by our server synchronization protocol. Specifically, we were interested in measuring the duration of the server synchronization phase; i.e., the time required to the replica servers to exchange their best bid and elect the leader.

To this end, we experimented our architecture in the following two different scenarios:

- **1st scenario:** two replica servers, simulating an open-cry auction, were distributed over the Internet. Namely, one of these servers was located in our Department, and the other server was located in the Department of Computer Science and Engineering of the University of California at San Diego (UCSD).

- **2nd scenario:** four replica servers, yet again simulating an open-cry auction, were distributed over a local area network in our Department.

In all the experiments we have carried out, the simulation of the open-cry auction was tuned to terminate after 12 rounds, and the first round was used to start up the auction. The results that have emerged from our experiments in the 1st scenario are illustrated in Figure 4.
This Figure shows that the server synchronization cost is negligible, as it is always below 500ms, with the exception of the start up round.

Figure 5 shows the results of our experiments in the 2nd scenario. It can be observed that, in this case, the four replica servers distributed over our departmental LAN perform slightly worse than those distributed over the Internet (owing to the congestion of our LAN). However, even in this case, the overhead of our server synchronization protocol appears to be negligible.

4.4 Fault tolerance

In this Subsection we describe the timing properties of our architecture. Specifically, as these properties can be violated principally by performance failures (defined in Subsection 3.1), we discuss below issues of performance failure detection and treatment in our architecture. Additional failures, such as omission failures and network partitions, are managed by the JGroup communication layer; crash failures can be dealt with by deploying such well known techniques as primary-backup [9], state machine [55], coordinator-cohort [7], or message logging[1]. These techniques fall outside the scope of this Report; hence, we shall not discuss them here.

As we adopt the time aware model, it is important to define a time-line to recognize early and late messages or processes. We show below that any server can compute those deadlines based on its local knowledge of time, and by means of the existing communication patterns. In our analysis we consider only the messages exchanged among the servers, as the bid messages are asynchronous.

The servers synchronization phase at each server can be decomposed in two sequential time intervals. In the first interval, called best bid exchange interval, each server sends its local best bid to the other servers, and receives the (local) best bids from these servers. In the second interval, called leader message interval, each server receives the auction state message from the leader. These two intervals enable each server to detect possible early and late messages. Specifically, each
server can compute locally the time extent of each of these intervals by means of Lemmas 2 and 3 (in the Appendix).

Any performance failure leads to a group partition; in particular, a server performance failure leads to late messages. Any late message means that its sender is not in synchrony with the server group to which it belongs. Hence, the sender of a late message can be considered by the other members of the group as partitioned away.

The server group manages the group view derived from a partition caused by late messages; the JGroup communication layer manages the view of the group derived from partitions caused by communication crashes or omission. In case JGroup detects a group partition, it notifies the members of each detected partition, and installs new correct views in each of these partitions.

If a server $S_i$ detects a performance failure of a message from $S_j$, it can only suspect that $S_j$ is partitioned away, owing to the asymmetry of the performance failure (see §3.2.1). Typically, it may occur that the server $S_j$ receives in time messages from $S_i$, and it is unaware of the fact that its own messages are delivered late to $S_i$. In these circumstances, $S_j$ can assume that $S_i$ belongs to its group, while $S_i$ will assume that $S_j$ is partitioned away.

Asymmetric performance failures can be managed by the leader, which locally detects the performance failure, installs a new view, and multicasts the auction state message to all the members of the new group. The leader sends the auction state message based also on the performance faults that it has detected, i.e. the auction state message contains only the $AT_j$ of the member of the new group of servers. When a server receives the auction state message which does not contain the arrival time of its own broadcast, it knows that it has been partitioned away from the group described in the message, due to a performance fault. It may occur that the server who received the best bid, say the server $S_i$, was partitioned away in an asymmetric fashion. The server $S_i$ behaves as the leader, but the other servers will ignore its auction state message as they agree on a view of the group that does not include $S_i$. When $S_i$ eventually joins the group again, it will be able to exchange its bids.

When a server detects that it is partitioned away, it invokes the JGroup communication layer services to leave the group, and then to join it again. JGroup executes a join request by notifying the group members, and installing a new group view. In our architecture, the servers can install the new view during the bid collection phases, only, so as to avoid interferences with the synchronization phases. (Note that, in case a new view is to installed during a synchronization phase, that phase is to be re-executed after the new view has been installed.)

### 4.5 Scalability

Following [54, 7, 16, 57], a distributed system is scalable if it preserves its properties, regardless of the number of users using that system, and the number of resources it accommodates.

The principal property our architecture is expected to preserve is the auction service responsiveness, i.e. the service availability and timeliness.

With respect to service availability, our architecture can scale by deploying techniques such as those mentioned above in order to face server crashes.

The scalability of our architecture with respect to the service timeliness can be assessed by evaluating the overhead introduced by our Server Synchronization Protocol, as the number of users and replica servers grows.

To this end, we have carried out an analytical study of our Server Synchronization Protocol, based on a simple analytical model of an auction replica server.

This study shows that, in the absence of failures, our protocol can preserve the timeliness property of the auction service at the cost of some additional bandwidth.

As the size of the messages exchanged during the server synchronization protocol is rather small, this additional bandwidth cost is negligible; hence, we claim that our synchronization protocol scales adequately, with respect to the service timeliness.

The two principal metrics we use to assess the scalability of our protocol are the User Response Time, and the protocol bandwidth usage. This latter metric is expressed in terms of maximum number of messages that are exchanged in an auction round among bidders and replica
servers, in order to submit bids, and among the replica servers themselves, in order to synchronize.

In our model, we assume $N$ replica servers, each of which can serve at most $R$ requests per second (RPS). Serving a request includes the time to receive a request (i.e., either a bid from a client or a synchronization message from a peer replica server), the time to process that request, and the time to transmit a possible reply message to the sender of that request (e.g., an acknowledgement).

Further, we assume that all the servers have the same performance and the same number of users connected to them, submitting bids possibly concurrently (i.e., at the same time).

Note that none of the above assumptions reduces the generality of our model; rather, this model captures the worst case scenario (see below) in which replica servers may operate. In addition, current server technology allows us to assume that $R$ be equal to some thousands RPS [10]; hence, in practice, a few tens of replica servers will be sufficient to manage the great majority of an Internet-based auctions. Our model, and the analytical results we have derived from it, are introduced below.

### 4.5.1 User Response Time

We define the user response time (URT) as the time elapsed between a user (bidder) submitting a bid, and the normal termination of that bid submission, (i.e., no failure exceptions are raised and the user receives the acknowledgement message for its bid).

A replica server may receive bids during the synchronization phase (in addition to those it receives during the collection phase).

Bids received at a server during the synchronization phase are immediately acknowledged to their origin bidders, and considered as if submitted at the beginning of the next collection phase.

Within this scenario, in the absence of failures, the URT depends on:

- the delay of the bid message from the bidder to the server;
- the bid processing time at the server;
- the delay of the acknowledgement message from the server to the bidder.

The worst case URT occurs when both the bid message and its acknowledgement experience the maximum delay $\delta_{\text{max}}$, and a server has its maximum load, i.e. it receives all the bids from its clients during the synchronization phase, concurrently with all the best bid exchange messages from the other servers. Then, assuming that $n$ clients participate to the auction, the maximum URT can be calculated as:

$$\text{URT}_{\text{max}} = \delta_{\text{max}} + \left\lceil \frac{n}{N} \right\rceil + \delta_{\text{max}}$$

Figure 6 shows the worst case URT in a system where $\delta_{\text{max}} = 0.5\text{sec}$, and $R = 200$. This graph can be used to estimate the number of replica servers required to meet specific SLA requirements. Thus, for example, if an auctioneer wishes to provide its clients with a URT = 4sec, and expects no more than 2000 clients, he will need at least 3 servers that serve R requests per seconds; in contrast, if that auctioneer expects at most 7500 users, he will need at least 12 such servers.

Hence, it is possible to size the auction system with respect to the expected maximum number of users. In case the actual number of users is less than the expected maximum, the exceeding computational power can be used to offer them enhanced services, as suggested in [3].

Note that Figure 6 shows that the URT grows linearly with respect to the number of users; moreover, adding replica servers to the system reduces the URT.

Finally, we can assess the URT speedup that can be obtained from our architecture, when compared and contrasted with a single server architecture. Figure 7 shows this speedup when adding new replica servers to the system; specifically, each curve in this figure illustrates the ratio between the URT provided by a single server and that provided by the number of replica servers corresponding to the curve. Thus, for example, an auction service provided by 12 servers to 10000 users can be ten times faster, approximately, than the same service provided by a single server. (Note that in some cases the speedup is close to the theoretical maximum, owing to the poor performance of the single server case).
Figure 6. User Response Time in the worst case

Figure 7. Speedup with respect to a single server case.
4.5.2 Bandwidth usage

The scalability of our system with respect to bandwidth usage can be assessed, at a first approximation, by estimating the number of messages that are exchanged in a single auction round (i.e., a bid collection and the relative server synchronization phases), as the total number of rounds that compose an actual auction is strictly dependent on that auction progress.

Assuming that \( u \) users are concurrently connected to the auction service, and \( N \) replica servers implement that service, we indicate with \( \mu(u, N) \) the number of messages exchanged in an auction round.

The values of \( \mu(u, N) \) depends on the following three factors:

- **Server synchronization messages**: the \( N \) replica servers of our system exchange \( N^2 - 1 \) synchronization messages in each round, as each replica transmits \( N - 1 \) messages to the \( N - 1 \) peer replica servers, other than itself. In addition, one of these servers will identify itself as the leader, eventually, and will transmit \( N - 1 \) messages to its peer servers (see Section §4.3). Thus, the total number of synchronization messages, exchanged during the synchronization phase, is \( N(N - 1) + N - 1 = N^2 - 1 \);
- **Number of bid messages**: in general, we can assume that only a percentage \( P \) (\( P \in [0, 1] \)) of the users connected to the auction service submit a bid within a single round. As each submitted bid is acknowledged by a single message, in the absence of failures, there will be \( 2Pu \) messages exchanged among clients and replica servers for bid submission and acknowledgement purposes;
- **Users update messages**: according to our protocol, at the end of each round all the users connected to the auction service are notified of the current best bid; this requires that \( u \) messages be transmitted by the replica servers to the connected users.

In summary, the number of messages exchanged in an auction round can be expressed by the following function:

\[
\mu(u, N) = N^2 - 1 + 2Pu + u = N^2 - 1 + u(2P + 1)
\]

We wish to point out that, in a real Internet-based auction, the number of users of that auction is expected to be orders of magnitude greater than the number of replica servers deployed to implement that auction. Hence, the value returned by the above function is largely dominated by the term related to \( u \), rather than that related to \( N \). In particular, it is worth observing that, for very large number of users (of the order of the thousands) and very small number of servers (of the order of the units) \( \mu(u, N) \) is in practice a linearly proportional function of \( u \).

However, the function above does not capture sufficiently accurately the bandwidth requirements of our architecture as it does not consider message timing issues; rather, it only expresses the total number of messages sent across the network in an auction round. In contrast, our architecture exhibits its worst case bandwidth requirements when both bid and synchronization messages are concurrently in transit over the network, during the server synchronization phase; i.e., when the network is required to accommodate all the messages that can be transmitted simultaneously over the connections to the replica servers. Note that these messages do not include both the acknowledgements to the bids, and the messages from the leader, mentioned above.

Thus, in the worst case, the maximum bandwidth \( MB \) required by each replica server, in terms of number of messages, consists of

- \( N - 1 \) synchronization messages that the replica receives, and \( N - 1 \) synchronization messages the replica transmits, i.e. \( 2(N - 1) \) messages, and
- \( \frac{Pu}{2} \) bid messages (assuming, as we did before, each replica receive bids from a percentage \( P \) of its users, only).

Hence, \( MB \) can be expressed as:

\[
MB = 2(N - 1) + \frac{Pu}{N}
\]

In order to estimate the total bandwidth (\( TMB \)) requirements of our architecture, in the worst case, it is worth observing that only the 50% of the synchronization messages each replica sends
and receives are to be considered (by not doing so, the same message would be considered twice). Owing to this observation, \( TMB \) can be expressed as:

\[
TMB = N [(N - 1) + \frac{Pu}{N}] = N^2 - N + Pu
\]

As an example, Figure 8 illustrates the maximum bandwidth requested by our architecture, in the worst case, for four different configurations of a replicated auction system. Namely, in this Figure, we consider an auction system implemented by 2, 3, 4, and 5 replica servers, respectively. We assume that each server has a processing capacity \( R = 200 \) RPS, and that all the users submit bids during the server synchronization phase (i.e., \( P = 1 \)).

It can be observed that, by augmenting the number of servers, the required bandwidth decreases. This follows from the fact that the increased number of synchronization messages, entailed by an increase in the number of servers, is compensated by the reduction in the number of bids transmitted to each server (under the assumption that each server is connected to the same number of users). Thus, as pointed out previously, for very large numbers of users and small numbers of servers, the bandwidth requirements of our architecture are dominated by the users bid messages.

In summary, we claim that our auction architecture scales adequately with respect to bandwidth usage, for all practical purposes, and that the bandwidth required by our server synchronization protocol is negligible, compared to that required by the bid message exchange.

## 5 Implementation

This Section summarizes a prototype implementation of our system we have developed using the Java programming language. This language offers platform independence; hence, it is possible to test the system using a wide variety of hardware resources. Even though Java has not been developed for soft real-time distributed programming, in the distributed auction system we have developed the performance bottleneck is the network, and the latency of the communication layer prevails against the overhead of the local programs; thus, Java has proved sufficiently adequate for our purposes. Moreover, JGroup is implemented in Java as a collection of Java classes.

For the sake of conciseness, we describe our implementation using a Java pseudocode, whose syntax is similar to that of the Java programming language. In addition, the pseudocode below...
does not include the management of asymmetric performances failures, as performed by the
leader, and discussed in the previous Subsection §4.3. Moreover, we assume that:
- the communication channels are bidirectional, i.e. if the server $S_i$ can send messages to the
  server $S_j$, then $S_i$ can send messages to $S_j$. The delays on both the directions are compar-
  able (this assumption excludes the possibility that asymmetric performance failures occur);
- the object implementing a server have access to a local physical clock;
- the operating system in each server module provides its local objects with a primitive called
  setAlarm(t), that causes the operating system to send a wakeUp signal, at time $t$, to the object
  that invoked that primitive;
- each process cannot set more than one alarm at any time, and a setAlarm(t) invocation
  overwrites the value of any previous alarm (note that the setAlarm(t) command, as it is
  presented, is a variation of the one proposed by Fetzer and Cristian in [22]);
- invocations to object methods, or to JGroup methods, are non blocking; i.e., the caller of
  the method continues its computation in parallel with the execution of the called method,
  without waiting for the call to complete and return;
- the objects react to the events they receive, and the method triggered by a received event
  is indicated in the pseudocode by the keyword method followed by the name of the event.
  Methods may take parameters, and may explicitly invoke other methods.

In our implementation, a server is structured in two principal objects that share a number
of variables. This structuring allows us to separate the time management from the actions per-
formed by a server. The first object, called timer, manages the timing of the server and its op-
erations. The second object, called communicator, handles the communications with the other
servers, and with the clients.

5.1 Modelling the underlying communication system
The JGroup communication system provides the following primitive operations:
- JGroup.multicast(msg, group): this method sends the message msg to the group of pro-
  cesses identified by the parameter group;
- JGroup.getView(): this function returns the set of processes belonging to the current view
  of the communication group.

Moreover, the communication layer may asynchronously send messages to the objects of the
system to notify them of possible changes in the communication group. These messages are:
partition and merge. The object timer manages these notifications by means of two methods with
the corresponding names, i.e. timer.partition and timer.merge.

5.2 Shared variables
The variables that are shared between the components of the auction system are shown in Table
1. The initial value of the variables is set by the startup procedure.

It may be worth observing that some of these variables, such as those referring the best bids,
can be initialized to default values (e.g., the initial selling price of an item); in contrast, the initial-
ization value of other variables, such as the view set, depends on specific configuration param-
eters (e.g., the Internet addresses of the replica servers). The type State is the set of the following
values {BIDS_COLLECTION, STEP1, STEP2}. BIDS_COLLECTION is self-explicative; STEP1
identifies the best bid exchange interval, STEP2 the leader election interval.

5.3 Timer object
Table 2 shows the pseudocode of the timer object methods.

The wakeUp method defines the timing of the auction. This method periodically starts the
collection phase, and sets the alarm to the deadline of the phase; similarly it sets alarms for the
deadlines of each one of the two intervals of the synchronization phase. If the timer object does
not receive a wakeUp event before the actual deadline, indicating the completion of the current
interval, a performance error occurred.
float bestBidLocal = worst possible bid;
float bestBidLocalMasked = bestBidLocal;
float bestBid = bestBidLocal;
float receivedBestBid = bestBidLocal;
State state = BIDS_COLLECTION;
ArrayOfFloat AT[view] = 0;
Time ds1 = 0;
Time ds2 = 0;
Time τ = initial value;
Time γ = 0;
Time δmax = initial value;
Time δmin = initial value;
SetOfServers view = initial value;
SetOfParticipants clients = 0;

Table 1. Shared variables, and their initial value

method wakeUp (event e)
{ if (e == endGamma)
  { state = STEP1;
     ds1 = γ + 2 * δmax - δmin;
     communicator.sendBestBid();
     setAlarm(ds1);
  } else if (e == endSt1)
  { state = STEP2;
     ds2 = γ + 3 * δmax - δmin;
     setAlarm(ds2);
  } else if (e == endSt2)
  { state = BIDS_COLLECTION;
     AT = 0;
     setAlarm(γ);
     communicator.updateBestBid();
  } else performanceError();
}

function newView()
{ V = Ø;
  foreach s ∈ view do
    if (AT[s] != 0) V = V ∪ {s};
  return V;
}

method performanceError()
{ if (state == STEP1)
  { communicator.kill(receiveBestBid);
    view = newView();
  } else
  { communicator.kill(receiveState);
    view = view - bestBid.sender;
  }
  AT = 0;
  γ = getTime();
  wakeUp(endGamma);
}

method partition()
{ view = JGroup.getView();
  AT = 0;
  γ = getTime();
  wakeUp(endGamma);
}

method merge()
{ if (state != BIDS_COLLECTION)
  wait until state = BIDS_COLLECTION;
  view = JGroup.getView();
}

Table 2. Timer object
As mentioned before, any performance error is detected by the auction system; in contrast, view changes, due either to a server group partition or to a group merge, are detected by the JGroup communication layer, and reported to the timer object.

The reception of messages does not cause any acknowledgment; hence, a message sender cannot know if a message it has transmitted has been delivered. Moreover, we assume that the underlying multicast service is reliable. A server detects a performance error when it does not receive the messages it is waiting for within their relative deadlines (as discussed in the previous Subsection 3.4). The kind of partition a server can detect depends on the current state of that server. If a server is exchanging the best bid (first interval), it can identify the servers that belong to its own partition. In contrast, if a server is waiting for the state message from the leader (second interval), it can only know if the leader belongs to its partition. In case a server detects that the leader does not belong to its own partition, that server starts again the synchronization phase to define the new members of its own group.

The communicator.kill method, invoked by performanceError, is a method that stops and resets all the activities of the receiving object; i.e. it aborts all the current communicator’s running methods and sets the object into a consistent “neutral” state.

JGroup does not deal with performance failures; rather it deals with group partitions and merges. When JGroup detects one of these cases, it calls the partition and merge methods of the timer object. A network partition can be detected only during the synchronization phase, and it needs to be managed in the current phase. A merge of partitions can be detected any time, and it is managed during the first collection phase, after its detection.

Figure 9 shows the state diagram of the phases of a server, and the events notified to the communicator object in any transition. The transition from the BIDS_COLLECTION phase to the STEP1 of the synchronization phase, due to a wakeUp(endγ), causes the notification of the event sendBestBid() to the communicator object. The transition from the STEP1 to the STEP2 of the synchronization phase depends on the reception of the wakeUp(endSt1), i.e. it depends on the signal that indicates that the phase 1 ended correctly. The transition from the state STEP2 to the state BIDS_COLLECTION, due to the reception of the wakeUp(endSt2), causes the notification of the event updateBestBid() to the communicator object.

5.4 Communicator object

The Table 3 shows the pseudocode of the communicator object, that takes care of the communication of the server. This object manages the synchronization messages with the other servers, and the bids it receives from the participants. Note that the JGroup.multicast(m, view) multicasts the
method participantBid (b)
{ if (betterBid(b,bestBidLocal) 
    AND (state == BIDS_COLLECTION)) 
    { bestBidLocal = b; 
    } 
    else if (betterBid(b,bestBidLocalMasked) 
    AND (state != BIDS_COLLECTION)) 
    { bestBidLocalMasked = b; 
    }
}

method sendBestBid()
{ ST = getTime();
  JGroup.multicast(bestBidLocal, view);
}

method receiveBestBid(bb)
{ AT[bb.sender] = getTime();
  if (betterBid(bb.value, bestBid))
    { bestBid = bb.value; 
    } 
  receivedBestBid++;
  if (receivedBestBid == view.size())
    { receivedBestBid = 0;
      if (bestBid.sender == this) sendState();
    } 
    else timer.wakeUp(endSt1);
}

method receiveState(stateMsg)
{ SAT = getTime();
  delta = ((SAT - ST) - 
    (stateMsg.SST - stateMsg.AT[this]))/2;
  \gamma = SAT + stateMsg.tau - delta;
  JGroup.multicast(bestBid, clients);
  timer.wakeUp(endSt2);
}

method sendState()
{ stateMsg.STT = getTime();
  stateMsg.tau = \tau;
  stateMsg.AT = AT;
  JGroup.multicast(stateMsg, view);
  timer.wakeUp(endSt2);
}

method updateBestBid()
{ if (betterBid(bestBidLocalMasked, bestBidLocal))
    { bestBidLocal = bestBidLocalMasked;
    }
}

Table 3. Communicator object

message m to all the processes in the set view, including its sender.

A participant may make a bid any time; if a server receives a bid when it is in the synchronization phase, it considers the bid as it will be received in the next collect phase. The two methods participantBid, and updateBestBid, manage both the reception of a bid and its possible postponement. Note that in the following pseudocode of the communicator object, the bids are managed according to the rules of the auction that are implemented in a function called betterBid. The function betterBid(x,y) returns the boolean value true if the bid x is better with respect to the bid y, otherwise the function returns the boolean value false.

The best bid exchange interval is implemented by the two methods sendBestBid and receiveBestBid. The former method is invoked by the timer.wakeUp method, while the latter is triggered by the reception of the best bid exchange messages from the other servers. The leader message interval of the synchronization phase is implemented by the two methods receiveState and sendState. The former method may be invoked at the end of the best bid exchange interval, the latter reacts to the reception of the leader state message. The termination of either the best bid exchange or the leader message intervals causes that the method currently in execution notify the timer.wakeUp method.

6 Related work

According to [5], the first popular auction service was eBay, established in september 1995. The fast growing popularity of eBay pointed out that the main advantage of on-line auctions was the broad base of possible clients they may reach [60]. However, on-line auctions showed some
shortcoming, such as the possibility of frauds, and the problem of maintaining the anonymity of the parties.

In the literature, few papers propose distributed architectures for auction systems. An early distributed e-commerce system was *Enchère* [8] that, in 1986, implemented a prototype of an agricultural marketing system consisting of a loose network of autonomous workstations communicating via message exchange. The auction model was the Dutch auction, due to the perishability of the goods on sale. Enchère supports disjoints groups of sellers and buyers connected via a network; auctions in each group can proceed in parallel to the ones in the other groups.

An interesting auction process model is described in [49]. The global market is subdivided in several markets, and the seller starts an auction of its item in the local market. If the seller does not receive bids better than the asking price, it starts new parallel auctions in selected remote markets. The seller continues the starting of new auctions until she/he receives a satisfactory bid; in the case of multiple winning bids, coming from different markets, the seller applies a conflict resolving technique to assign the item. In this model several auctions can be performed in parallel in order to sell the same item, allowing the system to be naturally scalable with respect to both the number of clients and servers. Moreover, a crash of a server during the evolvement of an auction does not affect the evolution of the whole auction; rather, it simply results in the casting away of the local market, relative to the crashed server. The principal shortcomings of this model are that, firstly, it lacks the idea of a global single auction; hence, an auction starts with a local portion of the total possible bidders, and gradually increases them by reaching new markets. Secondly, this model does not guarantee that an item be sold at the best possible price, as the final selling price for an item is the one that meets the seller expectancy. Thirdly, this model is suitable for English auctions, only. In Dutch and sealed bid auctions the timing is an important constraint, which contrasts with the idea of gradual additions of remote auctions.

In [24] the authors propose a distributed hierarchical architecture to implement English auctions, that is based on a tree of servers that periodically communicate by means of message exchange. At the opposite of the previous approach, this system starts the auction in the global market, and shrinks it excluding the local markets that do not place bids. In the tree hierarchy, the servers periodically exchange messages solely with their parent and children. The messages exchanged between servers contain the local history of the bids that a server received; thus, that server can present the global history of the auction's bids to the connected participants. As the system needs a consistent global ordering on the bids history, each single bid is propagated throughout the whole system. The classical organization of overlapping communication groups bounds any reliable multicast to all the servers as a sequence of reliable multicasts to parent-children groups, thus making the whole system inadequate to support Dutch auctions. The system does not tolerate permanent network partitions, and deals with server crashes with local passive replication.

Several works, such as [30, 35], deal with sealed-bid auctions. Those are long standing auctions; the main focus of the systems that implement them is on the security, validity and secrecy of the bids, rather than responsiveness.

It is argued in [36] that, in the near future, the on-line auctions market will be dominated by agents that will bid on behalf of human users. This scenario might lead to a new form of price definition completely different from human experience. Several papers propose agents as an emerging technology to implement auctions over the Internet; e.g., [15, 46, 52]. The principal idea of agent based auctions is that an agent, acting on behalf of a user, may search the Internet for the required goods, and buy them to the "best" price, as defined by that user. The agents auction follows the usual rules; the agents may implement sophisticated bidding policies in order to get a good at the best possible price. In [56, 33] the authors describe a competition among agents in order to test both a feasible framework for agent based trade, and trading strategies. In these papers, the competition is principally focussed on strategy issues, and the competition framework is centralized.

A completely different approach to e-commerce is discussed in [62] where the parties of a bargain are allowed to negotiate by means of an Internet application, called CBSS, that provides them with several interaction services, such as videoconferencing, whiteboard, document shar-
ing. This system is aimed to replacing a face-to-face negotiation between parties.

As mentioned in [26], electronic marketplaces introduce notable technical challenges, and stimulate new forms of trading. Interesting analyses of the dynamics of prices of goods in electronic marketplaces, and a comparison with the traditional marketplaces, can be found in [31, 53, 61].

The “real-time” aspect of a distributed architecture for auctions over the Internet represents a challenging problem due to the best effort nature of the communication network [25, 59]. Some authors propose a centralized real-time protocol for a client-server auction architecture based on Java applets [48]. This solution requires strict clock synchronization between the server and the applets running on the clients computers, a fair multicast from the server to the applets, and timely processing and delivery. The system performs periodic updates of the state of the auction, during which the client cannot put bids. The architecture proposed in [48] is poorly scalable, owing to the strict centralization of the server. In the already cited paper [43], the authors discuss a system to support the stock exchange market. The main focus of that system is on the total ordering of the bids, trust and responsiveness. The authors propose a twofold hierarchical architecture in order to obtain both acceptable performances and scalability. This architecture has different goals than ours, and the solutions it deploys are not well suited in our context. Finally, [42] investigates issues of design of small-scale auction applications, based on wireless, ad-hoc networks.

7 Concluding remarks

We have described a distributed architecture that can provide support for short term, on-line, one-sided auctions. This architecture is fair with respect to the participants, fault tolerant, and allows auctions to progress even in case of system partitions.

Our architecture scales well with respect to the number of participants. As an example, consider an increment of an order of magnitude in the number of clients. Typically, in a centralized auction system this increment is suffered by the single auction server; in contrast, in our architecture, it can be shared among \( N \) auction servers. In principle, our architecture may appear not to scale properly with respect to the number of servers, as the server synchronization phase requires \( N + 1 \) broadcast messages (i.e., \( O(N^2) \) messages). However, in practice, this is not a severe limitation, as the number of servers is several orders of magnitude smaller than the number of clients.

As in other proposals, our system subdivides the global market in several local ones; the novelty of our approach is in the definition of the locality, which is related to the Internet performance, rather than to physical distances. Typically, in our architecture, a client can be bound to the most responsive server, regardless of its physical location.

As shown in Subsection 4.3, the responsiveness of our system is tightly bound to the message transmission delays of the underlying Internet, and the overhead of the synchronization phase is of the same order of magnitude of these delays, i.e. few fractions of a second. Moreover, the participants to an auction do not perceive interruptions during the synchronization phase, as the servers continue to accept bids even in this phase.

Our architecture is appropriate for implementing english, dutch, and sealed-bid auctions; however, in this last type of auctions, it may introduce unnecessarily strict timing constraints, and excessive resource demands.

We have adopted an optimistic fault tolerance policy that allows our architecture to support both server and communication link failures, and to correctly manage network partitions. Finally, our architecture is capable of tolerating permanent network partitions of the auction server group by dynamically binding the clients of a partitioned away server to the surviving servers of the group.
8 Acknowledgment

We wish to thank our colleagues at UCSD, who made available the resources required to carry out the experimental evaluation of our architecture, described in this Report. In addition, we wish to thank our colleagues Prof. Lorenzo Alvisi (University of Texas-Austin) and Prof. Marco Roccetti (University of Bologna), for their comments on an earlier version of this Report.

References


Appendix

In this appendix we discuss analytically the principal timing properties of our architecture. For the sake of completeness, we wish to mention that firstly the message delays $\delta_n$ and $\delta_g$, introduced in Subsections 4.2 and 4.3, may change round by round; hence, they should be denoted by a round index $k$. However, in order to simplify our notation below, we have omitted the index $k$, where not strictly necessary. Secondly, we assume that the arrival time of a message at a server is the time at which that message is delivered to that server.

UBLCS-2003-09 28
Lemma 1 The difference between the delay $\delta_s$ and its approximation $\delta_i$ is bounded, specifically:

$$|\delta_i - \delta_s| \leq \frac{\delta_{\text{max}} - \delta_{\text{min}}}{2}$$

Proof By definition, $\delta_i$ is the middle point of the closed interval $[\delta_{\text{min}}, \delta_{\text{max}}]$, and this interval cannot be larger than $(\delta_{\text{max}} - \delta_{\text{min}})$. This proofs the assertion $\Box$.

Figure 10 depicts a possible scenario of the synchronization phase at the end of the first round. The dots on the process lines represent the end of the collection phases, labelled as $\gamma^1_i$, $i = 1, 2, 3$, where the subscript refers to the server identifier, and the superscript refers to the round. In this scenario all the $\gamma^1_i$ are simultaneous, owing to the clock synchronization in the start up phase. In this example $\delta_{\text{max}} = 7\delta_{\text{min}}$, and the delay of any message is either $\delta_{\text{max}}$ or $\delta_{\text{min}}$. According to the Lemma 1, in this scenario both $S_2$ and $S_3$ make the worst possible estimation of $\delta_s$; thus, $\gamma^2_2$ is over-approximated by $3\delta_{\text{min}}$ and $\gamma^2_3$ is under-approximated by $3\delta_{\text{min}}$. The Figure 10 shows a skew $\Gamma_2 = \gamma^2_2 - \gamma^2_3 = 6\delta_{\text{min}}$.

Since the duration of the collection phase $\tau_i$ must be bigger than zero, we have the following:

Corollary 1 The duration of any collection phase must be $\tau \geq \frac{\delta_{\text{max}} - \delta_{\text{min}}}{2}$.

Proof By its definition, we have that $\tau_i = \tau - \delta_i \geq 0$. The proof derives from Lemma 1. $\Box$

This corollary allows one to assess the smallest duration of any round. Not surprisingly this value depends on the network communication delays, i.e. it depends on $\delta_{\text{min}}$ and $\delta_{\text{max}}$.

As the effect of clock drifts is negligible with respect to the order of magnitude of the system timing, and the local clocks are used to measure time intervals, we obtain the following result:

Theorem 1 $\Gamma^k$ is bounded by a constant value at any round $k$ of the auction, specifically:

$$\Gamma^k \leq (\delta_{\text{max}} - \delta_{\text{min}}); \forall k$$

Proof Any server $S_i$ computes

$$\gamma^k_i = SAT_i + \tau_i$$

that can be rewritten, using the definitions of $SAT_i$ and of $\tau_i$, as

$$\gamma^k_i = \gamma^k_i + \delta_s - \delta_i$$
Figure 11. Summary of message timeliness properties computed by a server.

From Lemma 1 it follows that:

$$\gamma_i^k - \frac{\delta_{\text{max}} - \delta_{\text{min}}}{2} \leq \gamma_i^k \leq \gamma_i^k + \frac{\delta_{\text{max}} - \delta_{\text{min}}}{2}$$

Then the difference $\gamma_i^k - \gamma_j^k$ lies in the closed interval $[\gamma_i^k - \frac{\delta_{\text{max}} - \delta_{\text{min}}}{2}, \gamma_i^k + \frac{\delta_{\text{max}} - \delta_{\text{min}}}{2}]$ whose amplitude is $\delta_{\text{max}} - \delta_{\text{min}}$. This proof the theorem. □

We consider as the starting point of the following timing analysis the $\gamma_i^k$ of the $i$-th server. For the sake of simplicity we omit both the superscript $k$ and the subscript $i$ because any server $S_k$ has the same time-line scheme, and this repeats for any round $k$.

Each server can locally compute the time limit of both the best bid exchange and the leader message intervals; let those intervals be $[\alpha_1, \omega_1]$ and $[\alpha_2, \omega_2]$, respectively. The values of the time periods can be locally computed as shown by the lemmas below. Note that a server could receive best bid exchange messages even before its $\alpha_1$. Further, we can say that the best bid exchange messages and the auction state message are coordination messages between the servers.

Figure 11 shows the message time-lines that each server can locally compute in order to detect a performance failure. Any synchronization message that is delivered to the server $S_k$ before $\alpha_1$ is an early message; in contrast, if it is delivered to the server after $\omega_1$ is a late message. Similarly, an auction state message that is delivered before $\alpha_2$ is early, and, if delivered after $\omega_2$, is late. Note that setting $\delta_{\text{min}} = 0$ may lead to early messages due to the communication pattern.

**Lemma 2** The beginning of the best bid exchange interval is $\alpha_1 = \gamma - (\Gamma_{\text{max}} + \delta_{\text{min}})$. Moreover, the beginning of the leader message interval is $\alpha_2 = \gamma + 2\delta_{\text{min}}$.

**Proof** It may occur that the server $S_k$ receives a synchronization message, experiencing a delay $\delta_{\text{min}}$, coming from a server $S_j$, which is as early as $\Gamma_{\text{max}}$ with respect to its local clock; then, the first part of the lemma is true. Since the leader needs to receive all the best bid messages, the state message can arrive to a server at least $2\delta_{\text{min}}$ after that the server has multicast its best bid. □

**Lemma 3** The upper bound of the best bid exchange interval is $\omega_1 = \gamma + \Gamma_{\text{max}} + \delta_{\text{max}}$. Moreover, the upper bound of the leader message interval is $\omega_2 = \gamma + \Gamma_{\text{max}} + 2\delta_{\text{max}} + L$.

**Proof** It follows from the observation that the values of $\omega_1$ and $\omega_2$ must allow any correct message to arrive timely even if it experiences the maximum possible delay, and it comes from a server that is as late as $\Gamma_{\text{max}}$ with respect to the local clock. □
In Figure 12 the server $S_3$ exhibits the longest possible duration of the synchronization phase. The Figure shows an example of the two intervals as measured by the server $S_3$. In this Figure, the rectangle on the time line of server $S_3$ represents the deadline $\omega_{13}$ of the best bid exchange interval, and the triangle represents the deadline $\omega_{23}$ of the leader message interval. In the figure, the server $S_2$ receives the messages $b_3$ exactly at $\alpha_{12}$, while the server $S_3$ receives the auction state message from the leader $S_1$ exactly at $\omega_{23}$. 

Figure 12. Example of local computation of the two intervals of the server synchronization phase.