

Control Architecture for a Flexible Internet Auction Server

Peter R. Wurman Michael P. Wellman
William E. Walsh Kevin A. O'Malley

University of Michigan
Artificial Intelligence Laboratory
{ pwurman, wellman, wew, omalley } @umich.edu

March 4, 1999

Abstract

The flexibility to support both high activity and low activity auctions is required by any system that allows bidding by both humans and software agents. We present the control architecture of the Michigan Internet AuctionBot, and discuss some of the system engineering issues that arose in its design.

1 Introduction

The Michigan Internet AuctionBot (<http://auction.eecs.umich.edu>) is a highly configurable auction server built to support research on electronic commerce and multiagent negotiation [3]. The first generation architecture was simple and robust, designed to conduct a large number of simultaneous auctions with asynchronous and relatively sporadic bidder interactions. Once we introduced an automated trading facility (i.e., a software agent interface), we found that the frequent interactions demanded by automated traders could result in bottlenecks. We have redesigned the core AuctionBot control architecture to address this issue and improve overall performance. Our new control architecture meets the original goals of configurability and maintainability, while improving the scalability of the server to large numbers of simultaneous auctions with varying degrees of activity.

The pattern of interaction of the bidders with an auction depends on the number of bidders and the frequency with which they bid. Clearly,

software agents can analyze information and respond to changing market conditions much more quickly than their human counterparts. Auctions that host software agents, or many human agents, can expect to get bids very frequently. On the other hand, auctions that host a smaller number of human agents, or that operate over a longer time period can expect bids much less frequently, and may have extended periods of dormancy.

In AuctionBot architecture nomenclature, we say an auction is *open* if it is admitting bids, and it is *active* if it currently embodied in an auctioneer process. In the original AuctionBot control architecture, an auction remained open only for the duration of an individual auction event. The primary improvement in the new architecture is support for auctioneer processes that persist across events, yielding significant performance improvements for high-activity auctions.

In describing the control architecture, we focus on coordination of system components while handling the following auction events:

- **Create auction:** The AuctionBot system allows any registered user to initiate an auction. The auction's rules are specified by parameters [3].
- **Bid:** The AuctionBot system allows each user to have only one bid active at a time—a new bid replaces any previous bid the agent may have had. This is not a restriction because the AuctionBot permits very flexible bid representations.
- **Withdraw:** A bidder can withdraw its bid, subject to the auction rules.
- **Expire bid:** When the auction rules permit it, a bidder can specify a time at which its bid expires.
- **Clear:** Clearing is the act of matching buyers and sellers and determining the terms (i.e., price and quantity) of their exchanges. The AuctionBot is designed to support a variety of clearing policies, and has an extensive set of parameters to control the number and timing of clear events.
- **Quote:** Quotes provide intermediate information (e.g., current prices) to the agents. A set of parameters control the number and timing of price quotes.

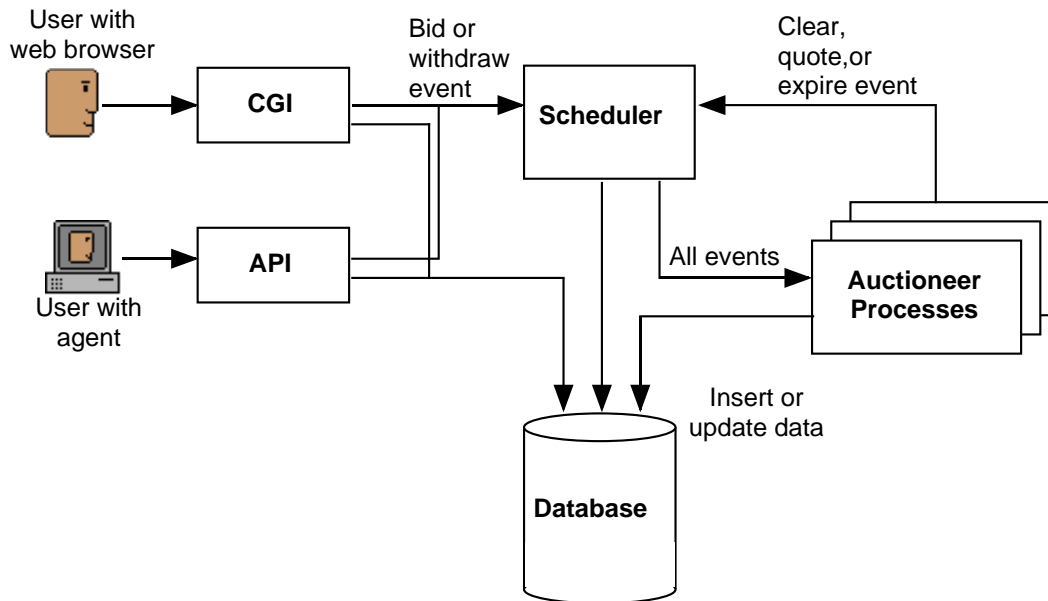


Figure 1: AuctionBot architecture.

2 Architecture Overview

The basic AuctionBot architecture is illustrated in Figure 1. It consists of four classes of components: a scheduler, a database, the interface modules, and the auctioneer programs.

The *interface modules* provide access for both humans (via a web server and CGI programs) and software agents (via the API—Agent Programming Interface). Both interfaces command the full range of AuctionBot functionality. However, this paper is concerned exclusively with the interface features that initiate auction events, as described in Section 3.1.

The *scheduler* is discussed in detail in Section 3.2. Its primary task is to sequence auction events. It also manages system resources by controlling the set of active auctioneers, and dynamically swapping them in and out based on the upcoming events.

The *auctioneer* programs, discussed in Section 3.3, implement the auction’s rules and pricing policies. Each auctioneer program can be instantiated in multiple concurrent processes, each one managing a different auction.¹

¹Perhaps a more efficient design would have each auctioneer be a multi-threaded program capable of managing several auctions at a given time. This design would save on the

From	To	Message	Parameter
interface	scheduler	BID	bid ID, auction ID
		WITHDRAW	user ID, auction ID
scheduler	auctioneer	BID	bid ID, auction ID, time t
		WITHDRAW	user ID, auction ID, time t
		EXPIRE	bid ID, auction ID, time t
		CLEAR	auction ID, time t
		QUOTE	auction ID, time t
		LOAD	auction ID, time t
		DEACTIVATE	
auctioneer	scheduler	EXPIRE	bid ID, auction ID, time t
		CLEAR	auction ID, time t
		QUOTE	auction ID, time t
		TERMINATED	process ID

Table 1: Event messages in the AuctionBot system.

The database serves as a central repository of information, and mediates access by all of the other modules. The database stores the details of each auction, bid, and transaction. Thus, if the scheduler is brought offline for maintenance or due to a system failure, the interface can continue to provide status information and receive bids. When the scheduler is brought back online, it extracts events from the database and processes them all in the correct order. This design provides a high level of data assurance.

The interface objects, the scheduler, and the auctioneer processes communicate via UNIX sockets. Table 1 shows the messages that are passed between the system components. The time parameter, t , is the time at which the event is scheduled to occur. In the following sections, we discuss the actions that each component takes when it sends and receives these messages.

Auction descriptions are stored in the database as a set of parameters and state variables (see [3] for more details). For the purposes of this paper, we are concerned with only the five auction fields shown in Table 2 and the subset of a bid record shown in Table 3. Note that, in the bid record, t_s is the official submission time—the time fields for bid reception and admission, overhead of initiating processes, but would be more complex than the current design and more fragile—one failed auction can prevent an entire class of auctions from progressing.

Field	Contents
auction_state	open or closed
next_quote_time	t_q
earliest_clear_time	t_e
next_intermediate_clear_time	t_i
final_clear_time	t_f

Table 2: Important fields in the auction record.

t_r and t_a , respectively, are for internal bookkeeping purposes.

3 System Components

3.1 Interface Modules

Three interface features initiate auction events:

- **Create Auction:** When a user initiates a new auction, the interface module first creates a new auction record in the database, then posts the auction’s first **CLEAR** and, when appropriate, **QUOTE** events to the scheduler. Note that the auction’s first clear time is bounded by its `earliest_clear_time`.
- **Bid:** When a user places a new bid, the bid submission interface module creates a new bid record in the database with a state of **received**. Then, the interface module sends the scheduler a **BID** event.
- **Withdraw:** When a user withdraws, the interface changes the `withdrawal_state` field in each of the user’s bids not already in a terminal state to **preliminary-withdraw**. It then sends a single **WITHDRAW** event to the scheduler.

The interface modules do not consider it an error if their communication with the scheduler fails. The system is designed to be robust even when the scheduler is offline—when it restarts, the scheduler will reconstitute the events from the database.

An aspect of the interface not explicitly diagrammed in Figure 1 are the messages passed from the AuctionBot to the user. Users can choose to be notified of the outcomes of auction events, including the admittance of bids, price quotes, and clears. Both human users and software agents can choose to be notified via e-mail or more directly through TCP/IP.

Field	Contents
bid state	see Figure 2
time_received	t_r
time_submitted	t_s
time_admitted	t_a
withdrawal_state	no-request, preliminary-withdraw, or withdraw-requested
time_withdraw_requested	t_w
expiration_time	t_x
time_closed	t_c

Table 3: Important fields in the bid record.

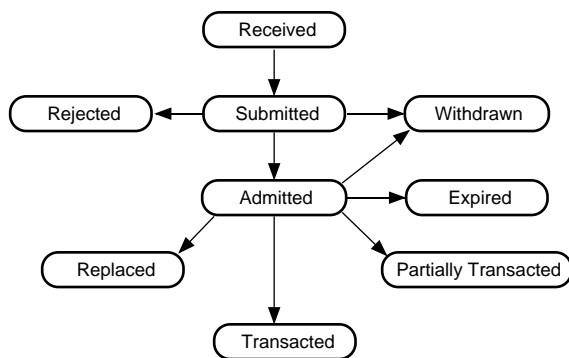


Figure 2: The state transitions of bids.

3.2 The Scheduler

The scheduler provides three services: it sequences events, manages the set of active auctioneers, and serves as the official timestamper of interface generated events.

The AuctionBot scheduler is a multi-threaded application with a shared data structure—the *event queue*—which is protected by mutex and condition variables. The main thread is the *event loop*, which continually checks the event queue and removes the top item at the appropriate time. The listener thread accepts new connections from auctioneers or interface objects and creates connection threads to receive the messages.

When it is brought online, the scheduler populates its event queue by extracting relevant event information from the database. An auction record contains the time of the auction's next scheduled clear and quote events. A bid record includes the bid's state and timestamps associated with its state transitions. This information is sufficient to completely rebuild the event queue.

Once the scheduler is online, it starts accepting connections. When a connection thread receives a **CLEAR**, **QUOTE**, or **EXPIRE** message, it simply puts the event into the queue. Incoming **BID** or **WITHDRAW** events, however, require that the scheduler interact with the database before putting the event in the queue. When a **BID** event is received, the scheduler changes the bid's status in the database to **submitted** and puts the current time in the `time_submitted` field. When a **WITHDRAW** event is received, the scheduler retrieves all bids in the auction that belong to the user and have a **preliminary-withdraw** state. For each bid retrieved, it updates the withdraw state to **withdraw-requested** and puts the current time into the `time_withdraw_requested` field. For all messages, the connection thread locks the event queue from the time the message is received until its insertion into the queue is complete.

The scheduler maintains a table of currently active auctions, called the *process table*. An entry in this table contains, among other things, the auction ID, the process ID, the socket ID, and a deactivation flag (initially set to false). When the event loop pops an event from the queue, it checks whether the auction is listed in the process table. If so, the scheduler sends the event directly to the appropriate auction process over a socket. If the auction is not active, the scheduler must find, or make, an open slot in the process table.

To clear a slot in the table the scheduler deactivates a running auctioneer in a two-step process. First the scheduler sends the auctioneer process

a **DEACTIVATE** message and sets its deactivation flag in the process table. When the auctioneer receives the message (after completing tasks already in its queue), it replies with a **TERMINATED** message. When the scheduler receives a **TERMINATED** message, it puts it on the front of the event queue.

However, the scheduler’s actions after sending the initial **DEACTIVATE** message are a little more complicated. First the scheduler waits briefly to see whether the process terminates immediately. If so, it clears the slot in the process table and instantiates the new auction. If, after this brief wait, the auctioneer process has not terminated (say, because it is in the middle of a complex auction calculation), the scheduler puts the event back on the event queue and continues with its event loop. In the next iteration of its event loop the top of the event queue will either be a **TERMINATED** message (which will allow the auction to clear a slot in the process table), or the original event (which will prompt the scheduler to select another auctioneer to deactivate). This continues until either all of the auctioneers have their **deactivating** flag set, or one of them has responded with a **TERMINATED** message.

This progressive deactivation of auctions can result in auctions in the table that have their deactivation flag set even after room has been cleared for the new auction. If the scheduler’s next event belongs to an auction that is deactivating, the scheduler must wait until the deactivation is completed (or it detects that the auctioneer process terminated).² This temporary recession is necessary to ensure that an auction is not instantiated in a second auctioneer before the first auctioneer has finished its computations and released the data.

Once a clear slot in the process table is found, the scheduler invokes an auctioneer and sends it a **LOAD** message, which tells the auctioneer which data set to load from the database. When an auction is being instantiated in a slot that is currently filled by the same auctioneer program, some of the overhead of process initiation can be eliminated by not deactivating the auctioneer and, instead, issuing it a new **LOAD** message.

3.3 Auctioneer Programs

Auctioneers in the new design are persistent—they maintain internal state between events as long as the process is kept running. This allows us to implement efficient, incremental auction algorithms such as the 4-heap algorithm [2], and to avoid a large number of database accesses.

²An improvement to the system would allow the scheduler to put the blocked event back on the queue and take another off.

Like the scheduler, the auctioneer consists of two threads: one listens for event messages, and the second runs the event loop. The auctioneer, however, receives events only from the scheduler, and only when they are past due.

There are three levels to the object hierarchy used to build auctioneer event loops. All auctioneers derive from a base class that provides support for managing bids efficiently by keeping pointers to bids in several internal data structures. The base class provides mechanisms that allow bids to be quickly located by owner ID (for withdrawal or replacement), by their notification status, and by their expiration time.

The second level provides support for particular bidding languages. The current auctioneers allow bids as a list of price-quantity points, and we have designed languages for continuous goods and for combinatorial auctions for discrete goods. The primary function of the language level is to parse bid strings into machine manipulable representations.

Classes at the third level implement specific clearing algorithms. Figure 3 diagrams the data structures at each level for an auctioneer that uses the 4-heap algorithm [2] to implement the family of auctions based on the k -double auctions [1].³

3.3.1 Auctioneer Event Handling

When the auctioneer receives a **LOAD** message, it retrieves the auction description from the database and all bids that have an **admitted** state. Bids are loaded in order of submission and hashed by owner ID. If the bid has a fixed expiration time, the bid is inserted onto the expiration queue.⁴ If the bidder requested to be notified of price quotes and/or clears, the bid is added to the appropriate notification set(s).

When a bid is loaded that has not been verified, either in the process of executing the **LOAD** command or in response to an explicit **BID** event, the auctioneer must verify that the bid satisfies the bid admission criteria. In addition to the syntactic constraints imposed by the bidding language, the auction may have rules that further restrict bid expressions, or require bids to beat the current price quote or the user's previous bid. If the bid fails these tests it is immediately transitioned to a **rejected** state and the user is sent a notification.

³The k parameter specifies the clearing price as a ratio between the upper (M th) and lower ($(M + 1)$ st) equilibrium prices.

⁴If the bid expired before the event time, $t_x < t$, the auctioneer expires the bid without loading it into the internal data structures.

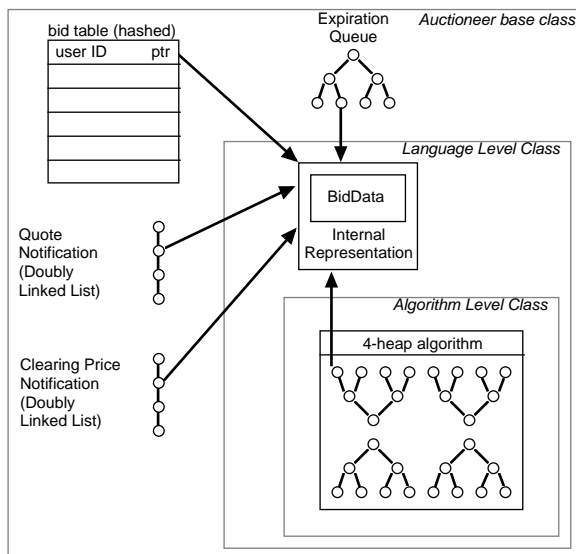


Figure 3: The internal structure of an auctioneer.

If the bid passes these tests, its admittance is recorded in the database by changing its state to **valid** and entering its admittance time. Before the new bid can be added to the supporting data structures, the auctioneer must remove any previous bid the user had in the auction. To do so, it changes the old bid's state to **replaced**, records the closing time, and removes it from any internal data structures.

The auctioneer takes two additional steps when it processes a **BID** message with a fixed expiration time, t_x .⁵ First, a reference to the bid is stored in the expiration queue, and, second, an **EXPIRE** event is sent to the scheduler. This is necessary to ensure that, should the auctioneer be deactivated before t_x , it is reactivated to handle the expiration. Without this explicit message, the auctioneer would not expire the bid until some other event forced its reactivation, and, until that time, the user interface would show the bid in an (incorrect) active state.

When the auctioneer receives an **EXPIRE** event, it removes the bid from the internal data structures, changes the bid's state to **expired**, and records the closing time. Similarly, a **WITHDRAW** event prompts the auctioneer to remove the bid and update its database record to reflect its new state (**withdrawn**), and closing time.

⁵The AuctionBot allows bids to expire at a fixed time, at the next clear, or when they are partially filled.

The actions of an auctioneer when it receives a `QUOTE` or `CLEAR` event depend on the algorithm being employed. However, there is one step common to all auctioneers. Both the `CLEAR` and `QUOTE` events can produce notification messages that need to be sent to the bidders. The auctioneer avoids examining all of the bids by using the notification lists.

A clear requires the added step of forming transactions between agents. Transactions can have one of two effects on a bid. If the bid transacts fully it is transitioned to `transacted`. If the bid is not completely satisfied by the transaction, it remains active in an adjusted form. In this case, the auctioneer duplicates the bid's database record and changes the state of the original bid to `partially-transacted`. The bid expressions of the copy in memory and the duplicate record are adjusted in a manner dependent on the bidding language. The local copy of the user's bid is updated to refer to the new database record.

Finally, some bids expire when the auction clears even if they do not participate in the transaction. The auctioneer needs to find these and close them.

The last step in the processing of a `CLEAR`, `QUOTE`, or `BID` event is to calculate the time of the next clear and quote events and, if they have changed, post the new events to the scheduler.

3.3.2 Asynchrony Issues

In this section, we address some subtle issues that we swept under the rug in the previous section.

First, there is the matter of bogus clear and quote events in the scheduler's event queue. An auction with a periodic clearing mode will generate a new `CLEAR` message only when it clears. Thus, the auction will have exactly one outstanding `CLEAR` event. However, an auction that clears after a period of inactivity will recalculate its next clearing time with each new bid. This can leave spurious `CLEAR` events in the scheduler's event queue. Rather than try to track and eliminate these bogus events, we catch them when they return to the auctioneer. An `AuctionBot` auctioneer ensures that it clears at the correct time by keeping a local record of its next clear time. When a `CLEAR` is received, if $t \neq t_i$ and $t \neq t_f$, then the event is ignored. Similarly, `QUOTE` events are executed only if $t = t_q$.⁶

⁶This scheme has the undesirable effect that auctions can be activated whose sole activity is to ignore bogus messages. Nevertheless, we concluded that measures to avoid this situation would not be worthwhile.

The fact that the auctioneer and scheduler communicate asynchronously introduces other problems. Consider a situation in which the auctioneer, at time t , calculates its next intermediate clear time to be t_i . A bid is received by the scheduler at t' , where $t' > t_i$, however the auctioneer's message to clear at t_i is delayed and doesn't reach the scheduler until after t' . The **CLEAR** and **BID** messages could be sent to the auctioneer out of order. For this reason, when the auctioneer receives the **BID** message at t' it checks whether $t_i < t'$ (or $t_f < t'$), and, if so, executes a clear before verifying the new bid.

A similar problem can occur with the posting of **EXPIRE** events. Consider a situation in which the auctioneer receives a **BID** message at time t with a fixed expiration time t_x . A **CLEAR** is scheduled for t' , where $t' > t_x$, but the auctioneer's **EXPIRE** message to the scheduler is not received until after t' . The bid should expire before the clear is executed, but the events will be sent to the auctioneer in the reverse order. The expiration queue is used to disarm this situation. When the auctioneer receives a **CLEAR** or **QUOTE** message, it checks the expiration queue and expires any bids with $t_x < t'$.

4 Conclusion

We have presented many of the details related to our design of a scalable Internet auction server that can accommodate the activity levels of both human users and software agents. The primary feature in the control architecture is its facility to run auctioneer processes that persist between auction events, without imposing a limit on the number of auctions that can be open at one time. This allows for more efficient auctioneer programs and a significant reduction in the number of database accesses.

Acknowledgments

This work was supported in part by grants from NSF, AFOSR, and DARPA, and IBM. Many University of Michigan students have contributed, including Stewart Blacklock, Brian Joh, Yee-Wah Lee, Jonathan Mayer, Tracy Mullen, Ryan Papa, Chris Wong, and Itai Zohar. We are grateful to them, as well as to all of the students who have participated in AuctionBot experiments. Jeff Kopmanis deserves credit for keeping the AuctionBot running smoothly.

References

- [1] Mark A. Satterthwaite and Steven R. Williams. Bilateral trade with the sealed bid k -double auction: Existence and efficiency. *Journal of Economic Theory*, 48:107–33, 1989.
- [2] Peter R. Wurman, William E. Walsh, and Michael P. Wellman. Flexible double auctions for electronic commerce: Theory and implementation. *Decision Support Systems*, 24:17–27, 1998.
- [3] Peter R. Wurman, Michael P. Wellman, and William E. Walsh. The Michigan Internet AuctionBot: A configurable auction server for human and software agents. In *Second International Conference on Autonomous Agents*, pages 301–8, Minneapolis, 1998.