A Dependable Distributed Auction System: Architecture and an Implementation Framework

Paul Ezhilchelvan and Graham Morgan

Department of Computing Science, University of Newcastle

Newcastle upon Tyne NE1 7RU, UK

Abstract

The work presented here develops a distributed systems architecture and propose an implementation framework for conducting dependable Internet based on-line auctions, meeting the requirements of scalability and service integrity. Current auction services essentially rely on a central auction server. Given the increasing popularity and usage of electronic auctions, such a centralised approach is fundamentally restrictive with respect to scalability. Further, different national markets have different monetary regulations and may employ different procedures for payment settlements. Catering for local market autonomy means that decentralisation is an essential and practical requirement. With these design goals in mind, the paper develops an approach that permits an auction service to be mapped on to globally distributed auction servers. It then proposes a framework for a fault-tolerant implementation of the architecture. Fault-tolerance is achieved through matured technologies: replication management and group paradigm.

Keywords and Phrases: Auctions, distributed servers, multicast groups, passive and active replication, reliable multicast, membership service, synchronous and asynchronous networks.

1. Introduction

The Internet and World Wide Web have emerged as a valuable networked information source that is increasingly being used for commerce. We focus on a particular domain of electronic commerce: the Internet based auctions, which are rapidly diversifying into various products. Most notable auction companies are eBay [http://www.ebay.com] for a wide range of products, CNET [auctions.cnet.com] for electronic goods, Priceline [www.priceline.com] for air-line tickets, and E*Trade [www.etrade.com] for financial products; eBay has recently moved into real estate markets. These auction services essentially rely on a central auction server. As the market grows, such a centralised approach, we believe, will prove to be very restrictive as too many users can overload the server, making the whole auction process less responsive than the sellers and buyers would prefer. We require an auction service to be *scaleable*, i.e., capable of providing its end users with satisfactory Quality of Service (QoS), regardless of the number of those users and their geographical distance. Further, as the auction market becomes global, the centralised approach cannot effectively cater for variations in national/regional market regulations and procedures: different markets may employ their own rules, monetary regulations, payment procedures, etc. These autonomy considerations make decentralisation in system design not just a desirable option but also an absolute practical requirement.

We therefore investigate ways of enabling widely distributed, arbitrarily large number of auction servers that cooperate in conducting an auction. Each auction server serves a local market and is a part of the global system. Allowing a user to bid at any one of the servers is our principal way of achieving scalability. We propose an architecture that hierarchically structures the auction servers to minimise inter-server communication and also preserves *fairness*: every participant bidder, no matter which server he/she places his/her bid to, should have an equally fair chance for submitting a successful bid. For simplicity, we consider that there is one seller, one item to be auctioned and many bidders, so fairness will be concerned only with the bidding process (generalisation is clearly possible, but not considered here). Achieving fairness of auctions conducted over the Internet using a single auction server is a challenging problem as it is [1,2], since differing message transmission delays experienced by bidders can clearly compromise an auction's fairness. Achieving fairness of an auction conducted over a group of auction servers is even harder, but must be done in order to obtain scalability.

Having presented the system architecture, we propose an implementation framework taking node failures into account. We use replication management techniques to build a reliable server that preserves the single-server abstraction to the users (traders). We then resort to group management techniques to facilitate the (replicated) servers to exchange messages and cooperate with each other. The proposed implementation assumes two basic services: *reliable multicast* and *group membership*, which many of the existing group management middleware systems [3-9] can readily provide. By exploiting the fact that a server is internally replicated (over a synchronous network), we circumvent the unsolveable problem [10] of accurate failure detection in an asynchronous network (e.g. the Internet) which the servers would use to communicate with each other. The paper is organised as follows. Next section describes various types of auctions and the design approaches taken by existing auction systems. Section 3 presents our system architecture, section 4 the implementation framework where fault-tolerance issues are discussed. Section 5 concludes the paper.

2. Background and Related Work

2.1. Types of Auctions

One of the most commonly known auctions is an *open-cry or English auction*. In a traditional (non-electronic) setting, the auctioneer initiates the auction by quoting an initial price which is usually the minimum price acceptable to the seller and is also known as the seller's *ask*. Bidders, assembled in the same room as the auctioneer, are to signal to the auctioneer to express their acceptance of the quoted price. As soon as the auctioneer sees a bidder signalling, he cries out the quoted price pointing at that bidder; this is meant to announce that the bidder has placed his bid for the quoted amount and his bid has been accepted; the quote now becomes the highest bid accepted. The auctioneer then increases his quote by some $\alpha > 0$. If the auctioneer sees another bidder signalling, the latter is assumed to be placing a bid for the latest quote (= current highest bid + α). He cries out the new highest bid accepted, pointing at the second bidder, and then increases his quote by α again. This continues until bidders signal no more for the latest quote from the auctioneer. At this point in time, the auctioneer may revise his latest quote by adding a smaller α to the highest accepted bid. If a few revisions of the latest quote also do not attract any bidder, the auctioneer conveys his intention to terminate the auction by repeating the highest accepted bid three times. (Any signalling from a bidder during this repetition would mean that the bidder is placing his bid for the latest quote, and the auction would continue.) If no signalling is detected during these repeated cries, the hammer is banged on the table to signal the end of the auction. The item is sold to the highest bidder who pays the highest bid amount. Note that if there is no bid even for the ask (the initial quote), the auction is abandoned.

Dutch auctions are generally used for selling items that tend to perish rapidly (e.g., flowers, fruits, etc.). Here the auctioneer starts with a quote which is usually more than the expected price for the item. If there is no bid, the quote is decreased by a fixed amount at fixed interval, until a bid is received or the quote falls below the seller-specified minimum amount. If there is a bid, the bidder takes the item for the bid amount. In *sealed-bid auctions,* a bid placed is not made known to other bidders. Once the deadline for accepting bids is gone past, the bids received are inspected. In *the sealed-bid first-price auction*, the highest bidder takes the item for the amount he specified; in *the sealed-bid second-price auction*, the highest bidder takes the item for the amount specified by the second highest bidder. The former can be shown to be equivalent to Dutch auctions, and the latter to English auctions [11].

In the auctions described so far, the seller remains passive (except in stating the minimum price). This asymmetry is removed in *double auctions* which admit aspects of bargaining. The seminal, *k*-double auction model of [12] involves a single buyer and a single seller who respectively submit a *bid* b and an *ask* a; if b exceeds a, then a trade is consummated at the price $k\mathsf{b} + (1-k)\mathsf{a}$, where $0 \leq k \leq 1$. Details on double auctions can be seen in [13, 14]. In what follows, we will focus on English auctions for selling a single item.

2.2. State of Art on Internet Auctions

Centralised Server Approach

A central server displays the item to be sold, announcing the deadline for placing bids. Bidders can access the server for bid placements and it is their responsibility to ensure that their bids get to the server before the deadline. Once the deadline is past, the item is sold to the highest bidder (as in English auctions). Some well-known central server systems are eBay and AuctionBot [15], the latter having been developed as a research system to explore auction mechanisms. The server may be accessed by long- or short-distance bidders via Internet, or by mobile systems, or by agents executing on the server itself (see figure 1(a)). The access is usually authorised through brokers whose main role is to ensure that the server can trust the bidders. (For example, zipRealty acts as the broker for eBay's real-estate business[http://www.ziprealty.com/]) The brokers typically verify the bidders' ability to settle the account and distribute authentication keys. The server itself executes tasks that provide various functionalities and can be configured to implement various auction-related policies. The functionalities are mainly: (i) verifying the authenticity and timeliness of incoming bids, (ii) displaying the information about the auction details, such as the bidding deadline, the highest bid placed etc., and (iii) the core functionality of processing the bids. We refer the reader to [15**,** 16] for details on these server tasks and the functionalities they are designed to provide.

With the increasing popularity and usage of Internet auctions, the centralised approach cannot clearly scale. Further, as [16] observes, it cannot effectively deal with the *autonomy* of *local markets*: different national markets may employ their own rules, monetary regulations, payment procedures, etc. Given that the auction system would be used by bidders from different parts of the world with diverse market procedures and customs, it must be designed to cater for variations in market mechanisms. These autonomy considerations make decentralisation in system design not just a desirable option but an absolute practical requirement. Enchere [17], an early auction system built with emphasis on data integrity, supports a *serverless* model whereby bidders and traders interact directly and decide the auction outcome. (Alternatively, it can be viewed as a centralised system where the central server is the stateless communication subsystem.) We believe such an approach is suited only to small scale auctions confined to a single market.

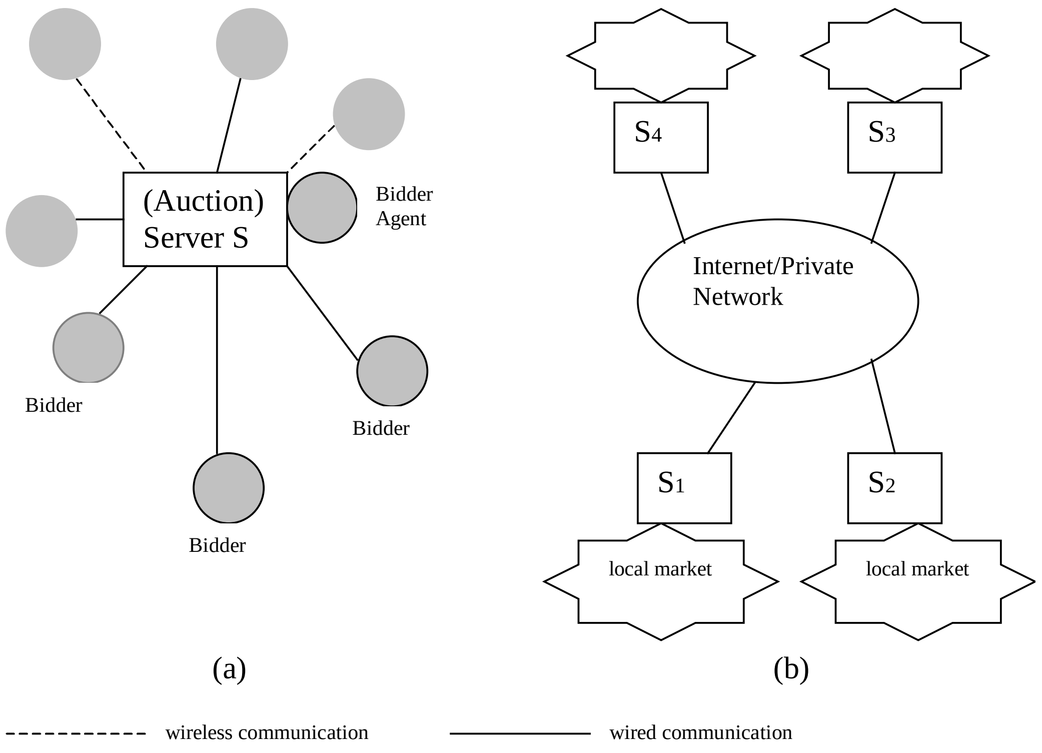

Figure 1. (a) Central Server S. (b) Interconnected Local Market Servers

Interconnected Servers

In this approach taken by [16], there is a server for a given local market, which exercises policies best suited to local conditions and market mechanisms, e.g. setting bidding deadlines by taking into account of the reliability and the available bandwidth of the local portion of the Internet, formulating payment rules that are feasible and/or popular with the local traders, etc. Local market servers are connected to each other by the Internet or by a high-bandwidth private network (see figure 1(b)), with a communication infrastructure that enables them to conduct B2B (business-to-business) transactions. Thus, a collection of interconnected local markets constitutes the global market.

The following auction process model is pursued: consider a seller contacting his local market, say S_1, for an item to be auctioned. S_1 initiates an auction (of the specified type) within its local market. If S_1 receives no bid more than the ask, it contacts a subset of other local markets, say S_2 and S_3, regarding X. (S_1 would have policy defined for choosing remote markets.) Thus, S_1, when necessary, acts as a broker in the global market (on behalf of its local trader). S_2 and S_3

interact with their respective local markets over S_1's request. It is possible that both S_2 and S_3 come up with a bidder for X. To ensure that X is sold (as a single item as advertised) either to S_2 or to S_3, S_1 follows a technique that is similar to resolving lock conflicts distributed databases. By thus letting a local server go global only on a need-to-go basis, scalability and autonomy concerns are addressed. In another model, known as the *explicit multicast* model (also supported in [16]), the seller of X simultaneously and explicitly requests, right at the start, not just the local server S_1 but also a subset of remote markets of his choice, to conduct auctions on X in their respective local markets, with S_1 playing an active role in ensuring that X is sold in at most one market. We compare this approach with ours in the next section where we describe our system.

3. Distributed Auction System

3.1. Overview

For ease of exposition, we shall assume in the rest of the paper that a bid received by a server is valid, timely, and authentic, and hence is accepted. This enables us to concentrate on a server's core task of processing the bids and selecting one (the highest) bidder for the (single) item to be auctioned in a global market. We would also assume that a bid accepted by a server cannot be withdrawn. Further, we admit no server or communication failures which are discussed in section 4. We regard the distributed auction system to be made up of many bid servers connected to each other via the Internet or a privately-owned, high-bandwidth network. Each server serves a local market as in figure 1(b). A seller will request the local bid server for auctioning an item. Once the request is accepted, the local server informs all other servers (by default) in the system to initiate an auction on the item. Every server in the system then displays the details of the item to be sold, announces the start of a new auction to its respective local market, and accepts bids. Periodically, it multicasts the bids it has received so far to every other server in the system. These multicast messages are called the *episode* messages, as their contents are to be aggregated by each server to form the history of bids accepted (so far) in the global system. The episode messages generated by a given server obey the following rule: every local bid accepted is referred to in one of the episode messages, and no two episode messages refer to the same bid. This is necessary to ensure that the global history constructed by each server represents any given bid exactly once. To seek a better offer for the seller, out-bidding is encouraged: each server displays the global bid history it computes, to the local bidders - thus making them aware of the global bidding trend. Further, if a remote bid is known to be the same as or higher than the highest local bid, the server extends the deadline for accepting new bids from the local market. When no new higher bid is placed in any of the servers until the expiry of the locally-set deadlines, the auction terminates.

Note that, in our approach, the default market is the global one involving all servers; whereas, in the interconnected severs approach, the market starts with the local one, and expands to other local markets when there is no demand in the local market. However, implementing a distributed global auction system is a challenging task, and requires, from systems and networks point of view, the following problems to be solved.

Message Exchange: Imagine the system being comprised of tens of distributed bid servers. Requiring each server to multicast its episode messages periodically to the rest of the system, is not a scaleable way to build the system, even if one takes into account of the advances in IP-multicast technology that uses programmable (multicast-aware) routers. So, a sensible structuring of the system is needed. For reasons of scalability, such a structuring should not particularly require a server to know, or multicast messages to, all other servers in the system.

Termination detection: Knowing that the global auction has/has not terminated is equivalent to solving termination detection problem in a distributed setting – which is not a trivial task though algorithms do exist [18, 19] but are typically message expensive. Hence, the system structuring we decide on, must assist solving this problem in economical ways.

Market Shrinkage: Imagine that, in a particular local market, there is no bidding at all right from the start of the auction, or bidding ceases early during the auction process; it is better for that server to reduce its processing load by leaving the global system regarding that auction process, so that only the interested market servers continue to communicate among themselves. So, any technology we use to implement the system must be capable of supporting dynamically changing market groups.

Addressing the third issue, completes the differences between our approach and the interconnected severs approach: the same objective is realised starting from diametrically opposite points. We start off with a default global market, provide support for shrinking market base if there is no demand. In the other approach, auction starts off with the local market and support is provided for market base to expand when necessary.

3.2. System Structure

We structure the system of servers into a tree, rooted on a single server. Fig. 2(a) shows eleven servers arranged in a tree, with the root being server S_{11} with which the seller is assumed to be registered. Recall that servers can directly communicate with each other as shown in fig. 2(b) and this tree structure is a logical one imposed in an attempt to make the inter-server communication scaleable and the termination detection efficient; also, that each server caters for a local market and has its own (local) bidders registered directly with it.

Adhering to the conventional terminology, the root server is regarded to be at the top-most level of the tree. A server is termed the *parent* of all those servers that are directly connected to it and are one level below; the lower level servers are termed the *child* servers of the parent. (In the tree of figure 2(a), S_9 is a parent for S_7 and S_3, and is a child of S_{11}.) A server (such as S_1) that has no child is called a *leaf* server. We do not require the tree to be a balanced one (though such a tree would improve the communication efficiency) nor a binary one as shown in the figure. What we do require is that the root server be connected to every other server either directly or via a sequence parent of servers, and that every non-root server have only one parent.

Based on the tree structure, servers are partitioned (not disjointly) into *multicast groups*: a group consists of one parent and all its children. Within a multicast group, servers know each other's identifier and periodically multicast the episode messages. Referring to the tree in figure 2, the eleven servers will be divided into five multicast groups: $\{S_{11}, S_9, S_{10}\}$, $\{S_9, S_7, S_3\}$, $\{S_7, S_1, S_2\}$, $\{S_{10}, S_8, S_6\}$, and $\{S_8, S_4, S_5\}$. Every server is in at least one group and a parent server, except the root, is present in two groups. For a parent server (such as S_7), the group that contains its children is called its down-tree group and denoted as G_d; e.g., G_d of S_7 is $\{S_7, S_1, S_2\}$. For a non-root server, the group that contains its parent is called its up-tree group and is denoted as G_u; e.g., G_u of S_1 is $\{S_7, S_1, S_2\}$.

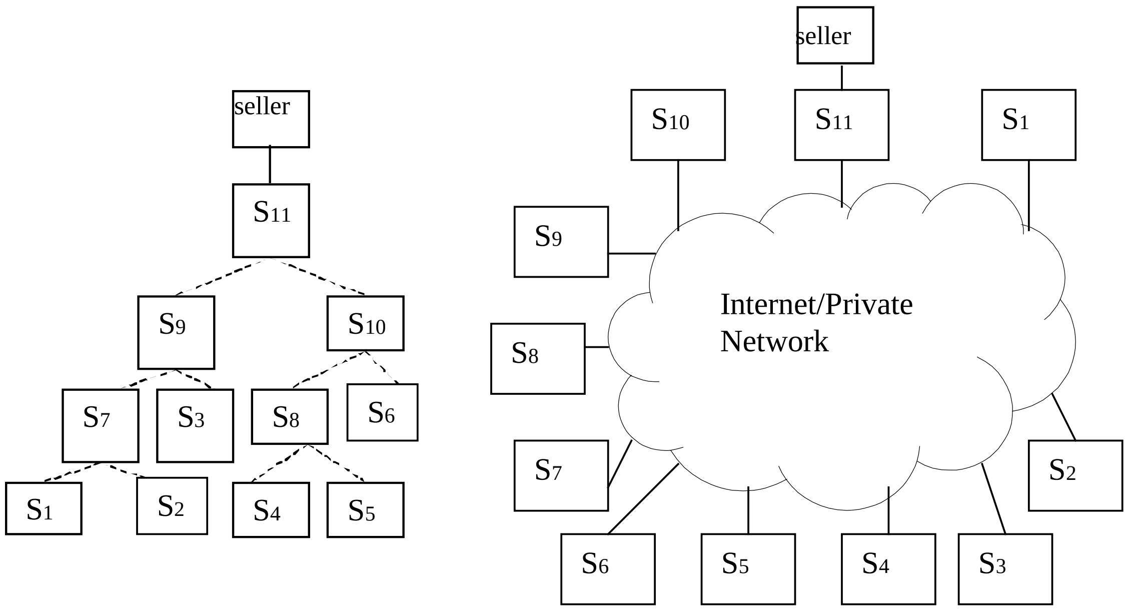

Figure 2. (a) Logical Tree of Servers. (b) Distributed System of Servers.

Partitioning the servers into multicast groups based on a tree structure, facilitates dissemination of episode messages in the following recursive manner. A non-root parent server periodically aggregates its own episode message with messages received from its children during the past period, and multicasts the aggregated episode message in its G_u. Thus, in its up-tree group, it represents the bids received by every server of the sub-tree rooted on itself. The downward propagation of episode messages also work in the same way but in the downward direction: a non-root parent server periodically aggregates its own episode message with the messages received from other members of its G_u during the past period, and multicasts the aggregated episode message in its G_d. The root server periodically multicasts only its own episode message in its G_d; similarly, a leaf server periodically multicasts only its own episode message in its G_u. The next subsection describes how episode messages are aggregated, particularly ensuring that any given bid (placed anywhere in the global system) is represented exactly once in the global history computed by every server.

Seeking tree-based structuring for reasons of scalability is frequently done in the literature. For example, the concept of IP-multicasting for a large number of receivers, first presented in [20], assumes that the IP-enabled routers are arranged in a tree (with the router attached to the message sender forming the root). Well-known scaleable transport protocols [21,22,23] use this tree structure to guarantee end-to-end reliability requirements. The analysis of [24] also favours that servers in a large scale setting be arranged in a tree for message efficiency. Assuming a tree structure, however, requires addressing the task of the tree-formation.

We assume that the server that a seller contacts will act as the root server (as indicated in fig 2). Given that the root is fixed, any of the appropriate tree forming algorithms readily found in IP-multicast literature can be used to construct a tree, if one is not already formed. We briefly focus on the policy issues that define the scope of the ‘global’ market. Though we assume that the seller’s local server includes all other servers into the system before forming the tree, in practice the server would exercise judgement in the selection of other local markets. This would depend on the expected demand in the remote markets for the item to be auctioned. For example, if the item is a second-hand bicycle and the local market is London, no remote market may be included; however, if that bicycle is the one which (late) Princess Diana used, then its market potential becomes truly global and at least all Western and Pacific markets certainly need to be included. The commission the seller is willing to pay will also influence the scope of the global market initially defined. Further, if a tree already exists for the type of item to be auctioned, then the local market may pass on the seller’s request to the root of the tree to include the item in the next auction. We here note that selecting a set of markets for defining the auction market is similar to the explicit multicast model supported in [16]; also that we permit the number of selected markets to be unrestricted. In what follows, we assume that the tree has been set-up, containing many local markets.

3.3. System Architecture

Information Board for Bidders

A server displays the following to the bidders of its local market: global bid history (GH), the ask price (ask) which is the minimum price the seller is willing to accept, the increment (I) which indicates the minimum amount by which a new bid should exceed the current highest bid, and the local time T_{clo} when the accepting of new bids will close. GH indicates the server’s (current) knowledge of the number of bids placed in the global system for different bid amounts. It may appear in the format shown in fig 3(a) which indicates that, for example, bid amounts of \$100 and \$150 are currently known to have been placed by 2 and 6 bidders, respectively. GH does not reveal the identity of bidders, only their cardinality for a given amount. The bidders counted in GH may have placed their bids at any one of the servers in the system. The number shown against a given amount may change with time as the server learns more on the global state. Since we assume a bid accepted cannot be withdrawn, it will never decrease.

Ask = \$100. I = \$10; T_{clo} = 1615 hours.

GH:

\$100	\$110	\$135	\$150
2	3	1	6

Figure 3. A server’s Display.

When the server starts the auction, it sets T_{clo} to be the current (local) time + Δ, where Δ is decided by the server depending on the worst case communication delays with the local bidders and the (local) time of the day (taking into account of the normal working hours, public holidays etc.). Whenever it accepts a new bid, it stores the bid details (the bid amount, the bidder id, bidder’s address etc.) in a local database, updates GH to reflect the new local bid, and extends T_{clo} by an appropriate Δ. Note that if the previous highest bidder reads the display before the closing time that was displayed when s/he placed the then highest bid, s/he would realise that new higher bid has been placed; extending T_{clo} by appropriate Δ allows the previous highest bidder to place a new bid higher than the current highest. Whenever the server receives an episode message, it updates GH. If it observes that the received episode message indicates a bid that is the same as or higher than the highest local bid it has accepted, then it extends T_{clo} by Δ (even if T_{clo} has gone past), and informs the local highest bidders of the change in the GH displayed and the new value for T_{clo}. Though extending deadlines and informing active bidders, result in a better price for the seller, they place a burden on the current highest bidder who intends to increase his bid in case a higher bid is placed locally or elsewhere. Such a bidder needs to be in a semi-active state, waiting to hear from the server. In practice, this burden can be alleviated by the server instructing the bidder to inspect its display at specified intervals, and computing Δ based on this interval. I is decided locally by the server which can change it (based on global bidding trend) when T_{clo} gets extended.

Episode Messages

We next describe how episode messages are constructed and aggregated. We will first consider a leaf server which need not aggregate episode messages but simply has to disseminate its own. When it accepts a new bid, it updates an *episode frequency table* (EFT) which has the same structure as GH: maintaining the bid frequency *only* for local bids that are accepted but not yet made known to the rest of the system. Before sending an episode message, it abstracts the contents of EFT into variable E which has three components: `E.base`, `E.bidders`, and `E.offset`; the last two components of E are integer and amount lists respectively, both indexed by k, $k \geq 0$. `E.bidders[k]` indicates the number of bids placed (according to EFT) for amount = `E.base + E.offset[k]`. After the computation of E, EFT is set to an empty table. Figure 4 gives an example EFT and the corresponding E.

EFT:

\$110	\$135	\$150
2	1	3

`E.base` = \$110;

`E.bidders` = [2, 1, 3]

`E.offset` = [0, 25, 40]

Figure 4. Episode Frequency Table (EFT) and the corresponding E..

If the EFT is empty, the E computed is denoted as $\perp$. We will model the process of computing E from EFT as an execution of function Φ: $E = \Phi(EFT)$. We define the operator + between variables of E: (E_1+E_2).bidders and (E_1+E_2).offset would represent all the bid frequencies represented by both E_1 and E_2, with (E_1+E_2).base = minimum{E_1.base, E_2.base}. Periodically, the leaf server computes E and, if $E \neq \perp$ it multicasts to every other server in its G_u the message `Episode(own-id, seq-no, E)` which has three parts: its own (unique) identifier `own-id`, the message sequence number `seq-no`, and the computed E. We assume that multicasts are delivered in the source-fifo (first-in-first-out) order at the destinations; so, the episode messages from a given server are delivered in the increasing order of their sequence numbers.

A non-leaf server (except the root) is in two groups, so it maintains two EFTs: EFT_u and EFT_d which maintain the bid frequency for accepted local bids which are not yet made known in G_u and G_d, respectively. (When a local bid is accepted, it is represented in both EFTs.) The server also has to aggregate the contents of the episode messages it receives. To ensure that no episode message is aggregated more than once, it maintains two private variables lr_i (last_received) and ld_i (last_disseminated) for every other server S_i in G_u and G_d: $\forall S_i \in G_u \cup G_d - \{own\text{-}id\}$. lr_i indicates the sequence number of the latest episode message received from S_i; ld_i the sequence number of S_i's latest episode message that the local server has aggregated and disseminated. Note that $ld_i \leq lr_i$ for any S_i. Finally, we use E_i^j to denote the E component of an episode message received from server S_i with seq-no= j.

When a non-root server wishes to multicast an episode message in G_u, it first computes $E_u = \Phi(EFT_u$; then, for every $S_i \in G_d - \{own\text{-}id\}$, it computes $E_u = E_u + E_i^{ldi+1} + E_i^{ldi+2} ... + E_i^{lri}$ if $ld_i < lr_i$, and sets $ld_i = lr_i$. Similarly, a non-leaf server computes:

$E_d = \Phi(EFT_d) + (\forall S_i \in G_u - \{own\text{-}id\}: E_i^{ldi+1} + E_i^{ldi+2} ... + E_i^{lri})$. The pseudo code for multicasting in G_u and G_d are given below.

```
periodically do
  compute $E_d$;
  if $E_d \neq \perp$ then multicast Episode(own-id, seq-no, $E_d$)
                                          in ($G_d$ - own-id);
  compute $E_u$;
  if $E_u \neq \perp$ then multicast Episode(own-id, seq-no, $E_u$)
                                          in ($G_u$ - own-id);
      // for termination detection
  if ($E_u = \perp$ and TC1 and TC2) then
      unicast to parent($G_u$)
                 Terminated(highest_bid, highest_bidders);
od
TC1: current_time $\geq T_{clo}$;
TC2: $\forall S_i \in G_d - \{own\text{-}id\}$: received_message(Terminated(highest_bid,
highest_bidders));
```

The above algorithm becomes generic for every server, if we let multicasting in an empty group be a no-operation. Provided that every leaf-server has only one G_u and the root has only one G_d, and that every non-root parent server S has $G_u \cap G_d = \{S\}$, we have the following: every parent server disseminates *at most once* (i) all bids accepted in the sub-tree rooted on itself, in its G_u; and, (ii) all bids accepted in the rest of the tree and by itself, in its G_d. A proof of this can be seen in [25]. Now, what needs to be shown is that the auction process does not terminate (prematurely) when a server has a non-empty EFT or its Episode message is in transit. Showing this will transform the above guarantee from *at-most-once* into *exactly-once*.

Auction Termination

When a leaf server finds $E_u = \perp$, it checks whether the current time is less than T_{clo}. If not, the terminating condition TC1 becomes true and the leaf server concludes that its GH has been displayed long enough and has attracted no further bid from the local market. (TC2 is trivially true for a leaf server as its G_d is a single-ton set containing itself.) So, it sends to the parent server of its G_u the Terminated message containing highest_bid and highest_bidders which respectively indicate the highest bid and the number of bidders who had placed that bid as per the displayed GH. (They are respectively the top and bottom entries of the rightmost column of GH in figure 3.) A non-leaf server additionally has to verify `TC2`: whether it had received a `Terminated` message from all its children for the pair {`highest_bid`, `highest_bidders`} which are the highest bid and the number of bidders who had placed that bid as per *its* GH. When the root server has both `TC1` and `TC2` to be true, it decides that bidding has ceased globally and the auction process has terminated.

Observe that the `Terminated` message is similar to the *marker* message used in [26] for determining a consistent global state of a distributed computation; also that a non-root server may send more than one `Terminated` message; if so, only the last one will have {`highest_bid`, `highest_bidders`} ≡ {$\$B_{max}$, N}, where $\$B_{max}$ is the highest bid amount placed by N bidders when bid placement ceases globally. To see how termination detection works ([25] has rigorous proofs), suppose that the root server decides on termination, with its pair {`highest_bid`, `highest_bidders`} being {$300, 5}. This means that none of its local bidders is willing to put a new bid of $300 or more within the deadline it had set (`TC1`); further, every one of its children (subtrees represented by its children aggregate to the entire tree minus itself) has also reported the same (`TC2`). So, by induction, every server had autonomously concluded and reported that it has no new local bidder who is willing to put a bid of $300 or more within the deadline it set.

Referring to 2(a), we describe how the root (S_{11}) decides the final winner, given that it had detected termination for {$300, 5}. Suppose that one of the six highest bidders is a local one and three are reported by S_9 and two by S_{10}. It conducts a weighted draw between itself (weight 1), S_9 (weight 3), and S_{10} (weight 2). (It typically chooses a random number in the range 1 .. 6, and decides the winner to be its local bidder, S_9, or S_{10}, if the random number is 1, in the range 2 .. 4, or in the range 5 .. 6, respectively.) It then announces the winner and the losers. If S_9 is the winner, it conducts a weighted draw, and this process of selection continues down the tree recursively. Full details can be seen in [25].

4. Reliability Issues

4.1. Network Fault Model

The distributed auction system described above has two subsystems: servers and the communication network that interconnects them. A server can fail, usually in various ways, and must be built reliable using internal redundancy so that the auction service remains available. Using well-known redundancy management techniques, reliable servers can be built. When the network is not owned or maintained by the e-Auction service provider, this “must-be(-built)-reliable” approach does not work for the network, especially in the case of the Internet. So we first establish the weakest failure model the network must satisfy. The Internet generally provides a reliable communication (in the sense that what is sent is received, perhaps after a few retries) provided networks do not partition. So, the network assumption needs to be:

NA1: provided that servers S_i and S_j are correct, a message sent by one to the other is eventually delivered (*asynchronous network*).

Meeting this assumption requires that communication path between any two servers, if broken, be eventually restored. NA1 enables the server communication be reliable but *not* synchronous: a bound on how long messages can take to reach the destination cannot be known with certainty. It should be noted that an eventual restoration of inter-server communication is essential for ensuring both liveness *and* fairness of the auction process. Consider, for example, a bidder B_1 registered with server S_1 that is permanently disconnected from the rest of the system. Other servers cannot terminate the auction without getting the `Terminated` message from S_1; otherwise, it is possible that the item is unfairly sold to another bidder who had placed a smaller bid than what B_1 placed (with S_1).

4.2. Handling Processor Faults

A processor can fail in many ways, and there are two extreme fault models.

Byzantine Model: A faulty processor can fail in arbitrary ways.

Crash Model: A faulty processor fails only by stopping to function (crashing).

In what follows, we would assume the latter fault type, since the abstraction of crash failures can be implemented on top of a system of processor replicas subject to Byzantine faults, by running appropriate software protocols [27]. The following assumptions are usually made in implementing such an abstraction.

NA2: The network (typically a LAN) that interconnects processor replicas ensures that, provided that two replicas are correct, a message sent by one to the other is delivered within some known bound (*synchronous network*).

A1: when two correct process replicas perform a given task with the same initial state, the final states they reach and any outputs they produce are identical.

A1 is essential for process replication and holds true in the context of bid processing; NA2 permits less than one half of the replicas to be faulty. (Without it, only less than a third can be faulty [28]).

4.3. An Implementation Framework

We would adopt passive replication strategy to build reliable servers as it would enable a replicated server S_i to provide fast responses in the absence of faults. Figure 5 shows the internal structure of S_i. IS_i is the interface processor (front end) between n, n > 1, processor replicas and the traders of S_i's local market, and it is assumed reliable for the time-being. Further, NA2 is assumed to hold between IS_i and the processor replicas.

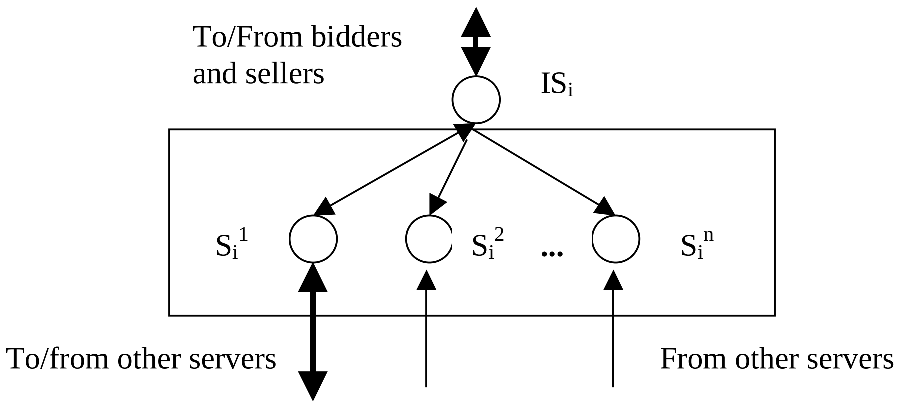

Figure 5. Internal Structure of Server S_i.

In passive replication, only the highest ranked replica, called the *primary* (S_i^1 in figure 5), processes, and responds to the requests; for every received request, it multicasts to other replicas the state changes effected and any response produced due to processing of the request. If ever the primary crashes, the highest ranked among the non-crashed replicas becomes the new primary and continues with the processing of incoming requests. The sever can provide services despite at most (n-1) replica crashes.

An implementation of passive replication is done using the following services within S_i. A *reliable fifo multicast service* (RM_i) which ensures that if the primary crashes during a multicast, either all functioning replicas or none of them receives that multicast, and a *group membership service* (GM_i) which promptly informs the functioning replicas of replica crashes and the order in which these crashes must be viewed with respect to message delivery order. (This property of synchronising crash notifications with message delivery order is known as *view or virtual synchrony* [3].) These services facilitate prompt selection of new primary after the existing one crashed, and guarantee that the survivors are in agreement on the last multicast the old primary made before it crashed so that the transfer of the processing role from the old to the new primary remains correct. The specification and protocols for RM_i can be found in [29], and for GM_i the specification in [3-9] and protocols (that use NA2) in [30-32].

Note that with passive replication, while every replica may receive the inputs, only the primary sends the server output to IS_i and to other servers. Next, we describe how the (passively replicated) servers exchange episode messages. For simplicity we will consider a single multicast group G = {S_7, S_1, S_2}(see figure 2), and assume that each server S_i, i = 1, 2, or 7, is internally duplicated (n = 2) and S_i^1 is the primary of S_i. (With n =2, at most one replica can crash within each S_i.) G can be configured to be G = {S_7^1, S_1^1, S_2^1}, containing only the server primaries. Note that the members of G communicate with each other using an asynchronous network where only NA1 (not NA2) holds. Suppose that S_7^1 crashes and an autonomous handling of this crash involves S_7^2 detecting the crash of S_7^1 (through GM_7 operating within S_7) and joining G. S_1^1 and S_2^1 (the surviving members of G) should not be entrusted with failure detection, as accurate failure detection is impossible over an aynchronous network [10]. Join operations are usually costly and time-consuming; so, we construct G containing not just the primaries but also the secondaries.

The composition of G is shown in figure 6. We assume a *reliable fifo multicast service* (RM_G) and a *group membership service* (GM_G) within G. Using RM_G, (only) primaries would multicast episode messages which are received by every member of G. Note that RM_G and GM_G must be implemented with NA1 alone. Many groupware systems e.g. [3-9], can provide these services just with NA1. However, they use *failure suspectors* to handle crashes which must be switched off and membership changes be effected by *failure notification* multicast (in G) by a S_i^2 when primary crash is detected through GM_i. Observe that S_i^2 can reliably detect the crash of S_i^1 using the (local) GM_i that is built with assumption NA2. Further, (the view synchrony property of) GM_G will ensure that S_i^2 is in agreement with other members of G over the last episode message that S_i^1 had multicast in G before it crashed. Therefore, no episode message of S_i will be left unsent in G when S_i^2 promotes itself to the primary of S_i.

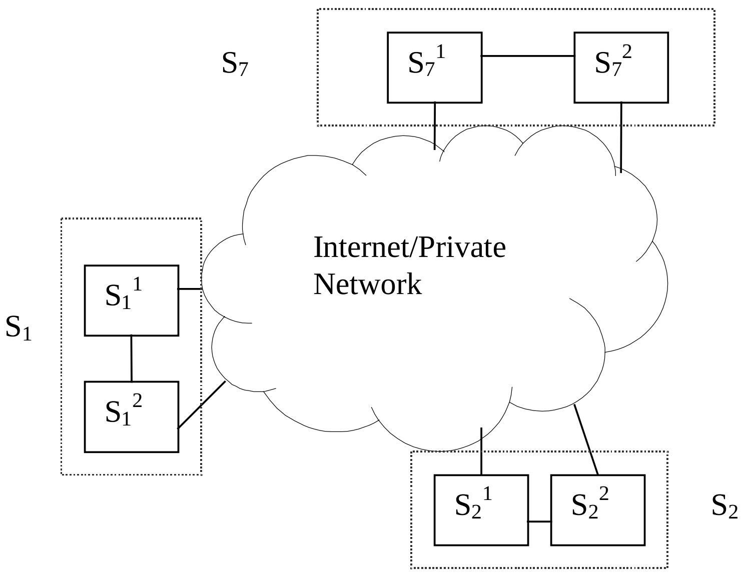

Figure 6. Replicated Processors S_1, S_2, and S_7 forming a Group

Market Shrinkage

Suppose that S_1 wishes to leave G due to lack of interest in its local market. This leave request can be easily handled by GM_G by treating it as 'announced crashes' of both S_1^1 and S_1^2. Suppose that S_7 also wants to leave G sometime after S_1 had left. It cannot leave G until S_2 joins its G_u = {S_7, S_3 S_9} (see figure 2(a)); otherwise S_2 will be left with no parent. As all the cited groupware systems support joining of new members in such a way that the existing members view the joining identically with respect to the messages they delivered in the old and new configurations; hence, S_2 joining {S_7, S_3 S_9} can be achieved in a manner consistent with the on-going multicasts within {S_7, S_3 S_9}.

Reliable Web Interface for Servers

Figure 7 depicts the approach we have taken to incorporate a web interface for a server (whose internal replication is not shown). The *auction service* supported by S_i can be implemented as a collection of CORBA services; traders (bidders, brokers, and sellers) would access these services (subject to well-defined access control policies) via serverlets running in association with a web server. This approach allows auction services to be implemented on a (market) server different to that of the web server. Access to auction services is via forms on web pages. As all popular web browsers may accommodate the use of forms, there is no need to enhance the trader side with additional functionality to enable the use of auction services. We ensure that the web server is a stateless machine and all state information is held within S. However, the failure of IS_i can make the auction services unavailable. So, we replicate the web server as shown in figure 8.

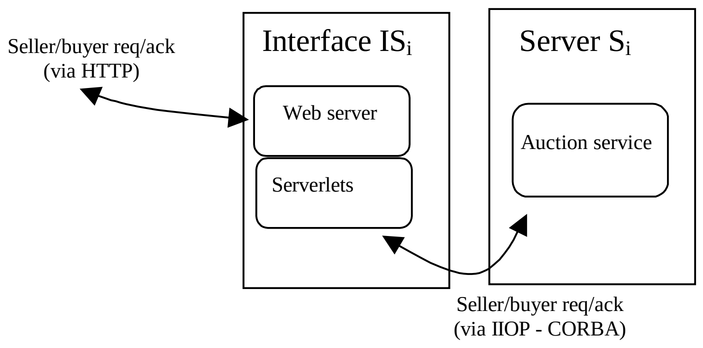

Figure 7. Web server as a server interface

There are two sets of web server/serverlets running on distinct machines IS_i^1 and IS_i^2, and at most one of them can crash. The router is assumed reliable (single point of failure) and uses a mechanism (such as round robin DNS) to determine which (functioning) web server to direct an incoming trader message to. The serverlets of each IS_i^j then reliably multicast the message to both server replicas.

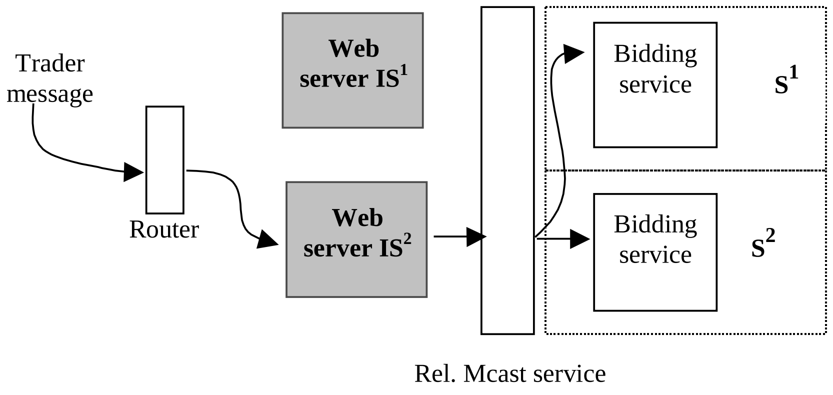

Figure 8. Increasing the Reliability of a Server Interface

5. Concluding Remarks

In this paper we have developed a hierarchic auction architecture to enable an auction to be conducted over a set of arbitrarily distributed auction servers, in the presence of processor failures. Allowing a user to bid at any one of the servers is our principal way of achieving scalability, as the total load is shared amongst many servers. In this respect the architecture presented here is very attractive for conducting auctions on a global scale, as it enables a federation of *auction houses* to cooperate. In the physical, non-electronic auctions, it is the auction house to which buyers and sellers must go in order to conduct an auction. The auction house is responsible for setting up and guaranteeing various contracts that are used to create and manage the auction. For example, certifying that the seller is authorised (and has) to sell the item, ensuring that bidders have sufficient credit limits (and have not been previously barred from bidding), and guaranteeing specific quality of service contracts. The auction house paradigm transfers relatively easily from the physical to the electronic world, with the local auction server representing an auction house, and the entire tree (fig. 2(b)) representing a federated auction house.

The implementation framework addresses two important issues: (i) building a reliable/available server through process replication, and (ii) enabling a group of such servers to communicate in a flexible manner that allows a server to leave the system if it receives no active bidding in its local market. The latter is achieved through the use of group management techniques which are well established both in theory and practice. By exploiting the fact that a server is internally replicated over a synchronous network, we circumvent the unsolveable problem [10] of accurate failure detection in an asynchronous network (e.g. the Internet) which the servers use to communicate with each other. Currently, we are implementing our architecture, for a system of 3 long-distance servers to start with. (They are situated in Pisa, Newcastle, and London.) Our NewTop group management system [6] (built in Java) provides the basic services which the framework proposed here assumes. We also intend to enhance the system design for multiple sellers, and other types of auctions including double auctions.

References

[1] M.P. Wellman and P.R. Wurman, "Real time issues for Internet auctions", IEEE Workshop on dependable and real time e-commerce systems (DARE-98), Denver, June 1998.

[2] C.S. Peng et al, "The design of an Internet based real time auction system", IEEE Workshop on dependable and real time e-commerce systems (DARE-98), Denver, June 1998.

[3] K. Birman , "The process group approach to reliable computing", CACM , 36, 12, pp. 37-53, December 1993.

[4] Amir, Y., et al, "Transis: A Communication Sub-system for High Availability", Digest of Papers, FTCS-22, Boston, July 1992, pp. 76-84.

[5] D. Dolev and D. Malki, "The Transis approach to high availability cluster communication", CACM, 39 (4), April 1996, pp. 64-70.

[6] P D Ezhilchelvan, R Macedo and S K Shrivastava, "NewTop: a fault-tolerant group communication protocol", 15th IEEE Intl. Conf. on Distributed Computing Systems, Vancouver, May 1995, pp. 296-306.

[7] L.E. Moser, P.M. Melliar-Smith et al, "Totem: a Fault-tolerant multicast group communication system", CACM, 39 (4), April 1996, pp. 54-63.

[8] P. Felber, R. Guerraoui and A. Schiper, "The implementation of a CORBA object group service", Theory and Practice of Object Systems, 4(2), 1998, pp. 93-105.

[9] M. Hayden, "The Ensemble system", PhD thesis, Dept. of Computer Science, Cornell University, 1998.

[10] M.J. Fischer, N.A. Lynch, and M.S. Paterson, "Impossibility of Distributed Consensus with one faulty Process," Journal of the ACM, Vol. 32, No. 2, pp. 374-382, April 1985.

[11] P. Klemperer, "Auction theory: a guide to the literature", Journal of Economic Surveys, 13(3), July 1999, pp. 227-286.

[12] K. Chatterjee and W Samuelson, 'Bargaining under Incomplete Information', *Operations Research*, Vol 31, 1983, pp. 835-51.

[13] P.R. Wurman, W.P. Walsh and M.P. Wellman, "Flexible double auctions for electronic commerce: theory and implementation", Decision Support Systems, 24, 1998, pp. 17-27.

[14] D. Friedman and J Rust, "The Double Auction Market: Institutions, Theories and Evidence", Addison-Wesley, 1993. [5].

[15] P. Wurman, M. Wellman and W Walsh, "The Michigan Internet AuctionBot: a configurable Server for Human and Software Agents", In Second ACM International Conference on Autonomous Agents, pp. 301-308.

[16] B Rachlevsky-Reich, I Ben-Shaul, N T Chan, A Lo, and T Poggio, "GEM: A Global Electronic Market System" Information Systems Vol. 24, No. 6, pp. 495-518, 1999.

[17] J-P Banatre, M Banatre, G Lapalme, F Ployette, "The Design and Building of Enchere, A Distributed Electronic Marketing System", CACM Vol. 29(1), 1986, pp. 19-29.

[18] J Brzezinsky, J-M Helary, and M Raynal, "Termination Detection in a very General Distributed Computing Model" Proc. 13th IEEE International Conf. on Distributed Computing Systems, Pittsburgh, USA, May 1993.

[19] F Mattern, "Algorithms for Distributed Termination Detection", *Distributed Computing*, Vol 2 (3), 1987, pp. 161-175.

[20] S Deering, "Multicast Routing in a Datagram Internetwork", Ph D Thesis, Stanford University, 1991.

[21] R. Yavatkar, J Griffoen, and M Sudan, 'A Reliable Dissemination for Interactive Collaborative Applications', ACM Multimedia, 1995.

[22] S Paul, K Sabnami, J Lin, and S Bhattacharya, 'Reliable Multicast Transport Protocol (RMTP)', IEEE Journal on Selected Areas in Communications, 15 (3), April 1997, pp. 407-21

[23] A M P Barcellos and P D Ezhilchelvan, "An End-to-End Reliable Multicast Protocol using Polling for Scaleability", IEEE INFOCOM'98, San Francisco, April 98, pp. 1180-87.

[24] Andersson, A; Ygge, F. "Managing large scale computational markets" 31st Hawaiian International Conf on System Sciences, Vol VII, Software Technology Track, pp 4-13, IEEE Computer Society, 1998.

[25] P D Ezhilchelvan, S K Shrivastava, and M C Little, "A Model and Architecture for Conducting Hierarchically Structured Auctions", May 2000.
Available at: www.cs.ncl.ac.uk/people/paul.ezhilchelvan/home.formal/unpublished/hierarchicFP.ps

[26] K M Chandy and L Lamport, "Distributed snapshots: Determining Global States of Distributed Systems", ACM Transactions on Computing Systems, Vol . 3(1), 63-75, 1985.

[27] S. K. Shrivastava, P. D. Ezhilchelvan, N. A. Speirs, S. Tao, and A. Tully, "Principle Features of the Voltan Family of Reliable System Architectures for Distributed Systems," IEEE Transactions on Computers, Vol. 41(5), pp. 542-549, May 1992.

[28] M Castro and B Liskov, "Practical Byzantine Fault Tolerance", Third ACM Symposium on Operating Systems DEsign and Implementation (OSDI) Feb 99, pp. 173-186.

[29] V Hadzilacos and S Toueg, "Fault-Tolerant Broadcasts and Related Problems", in *Distributed Systems*, (Ed.) S Mullender, Addison-Wesley, 1993.

[30] H Kopetz, G Grunsteidl, and J Reisinger, "Fault Tolerant Membership aervice in a Distributed Real-Time System", International Conference on Dependable Computing for Critical Applications (DCCA89), Santa Barbara, Aug 1989, pp. 167-174.

[31] P.D. Ezhilchelvan, and R. de Lemos, "A Robust Group Membership Algorithm for Distributed Real-time systems", Proceedings of the 11th Real-Time Systems Symposium, Florida, December 1990, PP. 173-179.

[32] F. Cristian, "Reaching Agreement on Processor Group Membership in Synchronous Distributed systems", *Distributed Computing*, 4(5), pp. 175-187, April 1991.