Measuring the Cost of Scalability and Reliability for Internet-based, server-centered applications

Paul Ezhilchelvan, Mohammad-Reza R.Khayyambashi, Doug Palmer and Graham Morgan
Department of Computing Science, University of Newcastle
Newcastle upon Tyne NE1 7RU, UK
{Paul.Ezhilchelvan, M.R.Khayyambashi, D.J.Palmer, Graham.Morgan}@ncl.ac.uk

Abstract

With large numbers of geographically dispersed clients, a centralized approach to Internet-based application development is not scalable and also not dependable. This paper presents a decentralized approach to dependable Internet based application development, consisting of a logical structuring of collaborating sub-systems of geographically-apart replicated servers. Two implementations of an Internet auction, one using a centralized approach and the other using our decentralized approach, are described. To evaluate the scalability of the two approaches, a number of experiments are performed on these implementations and the results presented here.

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

1. Introduction

We are concerned with a particular class of Internet-based, server-centered applications whose user domains are typically large, geographically distributed, and perhaps expanding. Examples of such applications are on-line auctions, Internet gaming, etc. On-line auctions are continually expanding into diverse products ranging from second-hand goods to airline tickets and financial products. The well-known Internet auction provider, eBay [http://www.ebay.com] has recently entered into the real-estate markets. The size and the nature of the user domain becomes obvious when we observe that eBay runs upto 2 million auctions at any given time, and its systems typically interact simultaneously with millions of Internet based customers from all over the world. Internet games not only are becoming increasingly popular but are such that the more the number of players participating in a game, the more interesting the game becomes for every player. So, in Internet gaming, systems are required to deal with a large number of users whose requests (for example to move or shoot an object) must be processed in an ordered manner and the effect displayed in a timely manner.

The applications with large and geographically dispersed client bases are currently supported in a centralised manner: client requests are sent (over the Internet) to systems located in a central place for processing. This centralised approach has serious scalability problems. A customer (an auction bidder or a game player) who is close to the central server can have faster server access than a remote client, and thus may have an unfair advantage over the latter. Further, as the number of simultaneously arriving client-requests increases, the server load increases – resulting in performance degradation. An unreplicated (central) server also constitutes a single point of failure. It has been recently reported that the eBay's (central) server suffered an outage for 22 hours [11].

The aim of this paper is to explore a decentralised approach that would admit scalability while enhancing client fairness and system availability and responsiveness.

In our decentralised approach, the system consists of many, geographically-apart subsystems; each subsystem provides services to those users who are in its geographical domain, while frequently coordinating its activities with other subsystems so that the services it provides are also correct and consistent at the system level. Observe that subsystems themselves can be replicas of the old centralised system that was once found adequate for a limited number of users. Figure 1(a) depicts the essence of our approach. It shows the system to be made up of 11 subsytems which interact over the Internet or a privately owned network for fast message exchange. Each S_i has a local client base which is the set of clients who choose to avail the global services through S_i. There can be many factors (e.g. currency regulations, service fee) that may influence a client to choose a particular server, and we would

assume that the primary ones include geographical proximity and fast server access.

Consider an activity A which can be an auction for a particular item or an instance of an internet game. If all participants of A are clients of just one server, say S_1, then it is as if S_1 is acting as the centralised server, except that there is an enhanced client fairness and system responsiveness. Suppose that the participants of A are distributed among the servers S_1, S_5 and S_9. Now, the state variables that define the progress of A must be maintained consistently by these three servers. Thus, the cost of our decentralised approach is influenced by the impact of this communication between servers on the overall service latency for client requests. The objective of this paper are two fold. First, we assess the impact of server communications in the latency for processing client requests. Second, we assess the cost of replicating a server. Towards these objectives, we have implemented a two-server distributed auction system and compared its performance with a centralised auction system.

The paper is organized as follows. Next section describes our distributed auction system. Section 3 describes its implementation, in which two servers are used: one in Bolognia (Italy) and another in Newcastle (England), connected by the Internet. Section 4 presents results from experiments we carried out using the distributed and centralised auction systems. The centralised system has only the Newcastle server which processes requests from both England and Italy. Section 5 concludes the paper.

2. A Distributed Architecture

2.1 Overview

For ease of exposition, we shall assume in the rest of the paper that a client request received by a server is always a valid one that needs to be processed. This enables us to concentrate on a server's core task of processing the requests. We also assume that a client request accepted by a server cannot be withdrawn. Further, we admit no server or communication failures, which are discussed in section 2.3. We regard the distributed system to be made up of many servers connected to each other via the Internet or a privately-owned, high-bandwidth network. Each server serves a local set of clients as in figure 1(a). A client will send their requests to its local server for processing. Periodically, a server multicasts the requests it has received so far to every other server in the system. These multicast messages are called *episode* messages, as their contents are used by each server to form the *history* of client requests accepted (so far) in the global system. The episode messages generated by a given server obey the following rule: every local client request accepted is referred to in one of the episode messages, and no two episode messages refer to the same client request. This is necessary to ensure that the global history constructed by each server represents any given client request exactly once.

Implementing a distributed auction system is a challenging task, and requires, from systems and networks point of view, the following problems to be solved:

1. *Message Exchange*: Imagine the system being comprised of tens of distributed 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.
2. *System Shrinkage*: Imagine that, in a particular local server, there is no need to collaborate with other servers to satisfy client requests; it is better for that server to reduce its processing load by leaving the global system, so that only the interested servers communicate among themselves. So, any technology we use to implement the system must be capable of supporting dynamically changing groups.

Addressing the second issue, completes the differences between our approach and the interconnected severs approach (described in [2]): the same objective is realised starting from diametrically opposite points. We start off with a default global environment, provide support for shrinking server base if there is no demand. In the other approach, environment starts off with the local server and support is provided for server base to expand when necessary.

2.2 System Structure

We structure the system of servers into a tree, rooted on a single server. Fig. 1(b) shows eleven servers arranged in a tree, with the root being server S_{11}. Recall that servers can directly communicate with each other as shown in fig. 1(a) and this tree structure is a logical one

imposed in an attempt to make the inter-server communication scaleable; also, that each server caters for a local set of clients and has its own (local) clients 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 1(b), 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 of parent of servers, and that every non-root server has 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 1, 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}.

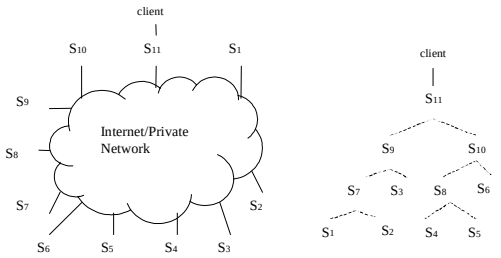


Figure 1. (a) Distributed System of Servers. (b) Logical Tree of Servers.

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. Recall that the formation and aggregation of episode messages are done in such a way that any given client request (sent to any server 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 [12] 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.

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' system. Though we assume that all servers are included, by default, in tree formation, in practice, judgement would be exercised in the selection of servers to form the global system and hence the tree. This would depend on the expected demand in the market base associated with a particular server. We here note that selecting servers to form the 'global' system is similar to the *explicit multicast* model supported in [2]; also that we permit any number of servers (resp. local markets) to be included in the global system (resp. auction market).

2.3 Reliability Issues

2.3.1 Network Fault Model

The distributed 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 reliably using internal redundancy so that a service remains available. Using well-known redundancy management techniques, reliable servers can be built. When the network is not

owned or maintained by the 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:

1. **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 to be reliable but *not* synchronous: a bound on how long messages can take to reach the destination cannot be known with certainty.

2.3.2 Handling Processor Faults

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

1. *Byzantine Model*: A faulty processor can fail in arbitrary ways.
2. *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 [13]. The following assumptions are usually made in implementing such an abstraction.

1. **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*).
2. **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; NA2 permits less than one half of the replicas to be faulty. (Without it, only less than a third can be faulty [14]).

2.3.3 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 S_i's clients, and it is assumed reliable[1]. Further, NA2 is assumed to hold among IS_i and the processor replicas.

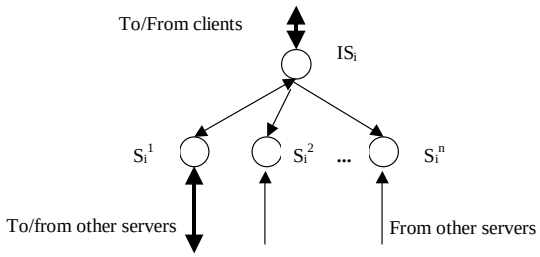


Figure 2. Internal Structure of Server S_i.

In passive replication, only the highest ranked replica, called the *primary* (S_i^1 in figure 2), 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

[1] The router is assumed reliable (single point of failure) and uses a mechanism (such as round robin DNS) to determine which (functioning) server to direct an incoming message to.

processing role from the old to the new primary remains correct. The specification and protocols for RM_i can be found in [15], and for GM_i the specification in [3-9] and protocols (that use NA2) in [16-18].

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 3. 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 groupbased 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.

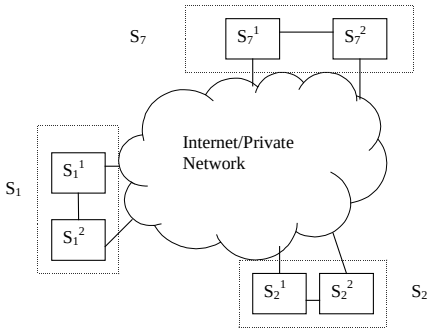


Figure 3. Replicated Processors S_1, S_2, and S_7 forming a Group

2.3.4 Server Group Shrinkage

Suppose that S_1 wishes to leave G due to lack of interest in its local group. 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 groupbased 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}.

3.Implementation of a Distributed Auction System

3.1 Overview

The auction system is structured as a number of market places that may be geographically separated over the Internet. A single server and a number of clients represent each market place; servers are structured hierarchically (one root server and two child servers), with clients placing bids and/or advertising items for sale via their local server. Servers are passively replicated, locally, to improve reliability within a single market place. Two replicas are used for replication purposes. Each server consists of four basic types of component: auction, reliability, distribution, group communication. Each component is implemented as a CORBA object.

3.2 Auction Object

The auction object provides a number of services:

- *Seller* – Allows the registration of seller details within the Bidding service. Contact details of a seller to enable buyers to trade with sellers are a minimum requirement and are managed by the seller service.
- *Buyer* – Allows the registration of buyer details within the auction service. As with the seller service, the primary purpose of the buyer service is to provide sellers with the relevant details of a buyer to enable trading between buyer and seller.
- *Product* – A product is made available to buyers by sellers via the product service. All the information required by buyers to enable an auction of a product is supplied by the product service (e.g., selling price, product description).
- *Trader* – Manages the bidding process. Buyers may place bids and buyers/sellers may enquire about the bidding status of products (if the auction in use allows this). The trader service also enforces the appropriate style of auction (e.g., English, Dutch).

Figure 4 gives a logical description of how these services collaborate to provide the bidding service.

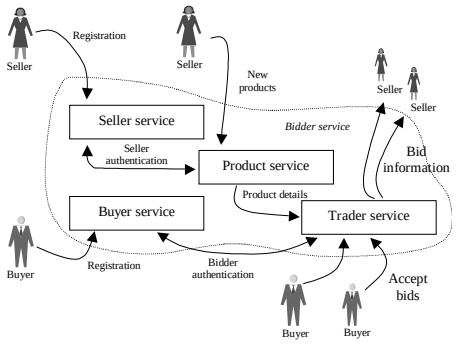


Figure 4. Logical view of auction object services.

3.3 Reliability Object

Reliability object makes the server tolerant to a single processor crash. Each auction object replica (two in our system) is provided with a reliability object. A reliability object provides two basic functions: accepts requests from clients, implements passive replication policy. Figure 5 describes the handling of a client request; M1 is the initial client request. On receiving M1, R1 multicasts this request (M2) to other reliability objects (two in the diagram) and forwards M1 to its own auction object (M3). On receiving M2, R2 forwards M2 (M4) to A2. As this is a passive replication scheme, only replies from A1 are returned to the client (via R1). Replies from A2 are ignored.

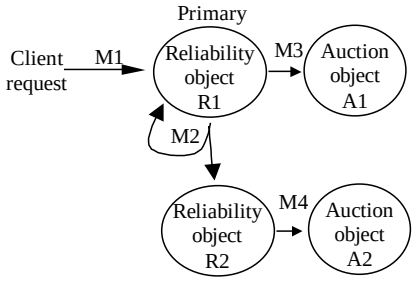


Figure 5. Reliability objects in a market place

The reliability object may be co-located in the same addressable space as an auction object or may be in a different addressable space, possibly on another machine in the network. In the latter case, the reliability object and its associated auction object may fail independently of each other. In this scenario, when an auction object fails the reliability object instantiates a new auction object, gaining state for the new replica from existing replicas (via other reliability objects). When a reliability object fails, another reliability object instantiates a new reliability object and associate this new object with the existing auction object. Furthermore, any client requests that have arrived since failure must be forwarded to the new reliability object, which in turn forwards them to its auction object.

3.4 Distribution Object

Each auction object is assigned a distribution object. The distribution object is responsible for imposing the hierarchical structure of the global auction system and managing the distribution of episode messages throughout this structure.

When auction objects are replicated, the distribution object is placed between clients and a reliability object (see figure 6). The distribution object accepts client requests (M1) and forwards them to its local reliability object (M2). The system administrator may determine the frequency a distribution object may multicast episode messages (M3). Episode messages

received by a distribution object (apart from its own) are disassembled into the original client request, each request is forwarded to the local reliability object (as M2).

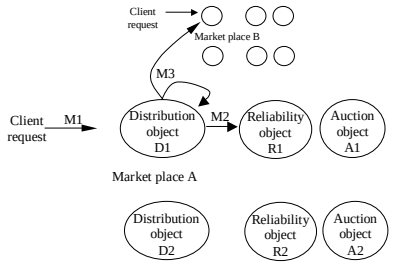


Figure 6. The role of distribution objects in the market place.

The distribution object of the passive replica (D2) does not receive any client requests nor does it send or receive episode messages. When the primary replica fails D2 assumes responsibilities of D1.

3.5 Group Communication Object

The group communication requirements of the reliability and distribution objects are satisfied by the Newtop service (a CORBA service) [24]. The Newtop service is a distributed service and achieves distribution with the aid of the Newtop Service Object (NSO). Each group member (reliability object or distribution object) is allocated an NSO. Group related communications required by a member are handled by its NSO.

The Newtop service consists of three services implemented by corresponding objects within the NSOs: membership; invocation/multicast; and group management. The management service provides members with create, delete and leave group operations. The invocation/multicast service provides three group invocation operations (wait for responses from all, from majority, from one and an asynchronous, no wait invocation). The membership service maintains the membership information and ensures that this information is mutually consistent at each member. This is achieved with the help of a failure suspector that initiates membership agreement as soon as a member is suspected to have failed.

4. Performance Evaluation

To demonstrate the effectiveness of our decentralized approach, we present performance figures related to a restricted implementation of the hierarchical architecture; only a single server group is implemented, releasing the need for a root node. We experiment with centralized server and distributed two-server systems (both replicated and non-replicated versions considered). We measure the time it takes for a bid to be registered at all auction objects in the system from the moment a bid is sent by a client. These measurements should not be treated as 'absolute' figures, but rather as an aid to compare the effectiveness of our decentralized architecture over a centralized architecture. For a fair comparison, all experiments were conducted overnight during which load fluctuations over the Internet were small. Four different types of experiment were carried out:

1. Centralised server non-replicated auction object
2. Distributed servers (two) non-replicated auction objects
3. Centralised server and replicated auction objects
4. Distributed servers (two) and replicated auction objects

Clients were configured to issue bids as frequently as possible; as soon as a reply is received another bid is issued. Client numbers were increased gradually from 100 to 200 in increments of 10. At each of these increments, registering 100 bids for each client in all auction objects is timed, and the average is taken.

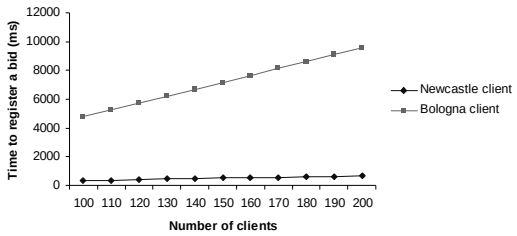
Communications between clients and servers were enabled via the Internet. Pentium Linux machines were used as hosts for clients and servers. Replicated servers (when used) where located on different machines on the same LAN. All objects of a single server were compiled into the same addressable space (e.g., D1, R1, A1 in figure 6). The implementation language used was Java 1.1 and the ORB used was ORBacus 4.0b3 [19]. Clients and servers were located at Newcastle (England) and Bologna (Italy). Clients were always equally distributed between England and Italy. (That is, when we say the number of clients is 100, it is 50 in England and 50 in Italy.) In the single server cases, only Newcastle server is operational which is accessed by both Italian and English clients.

4.1 Centralised Server, Non-replicated Auction Object

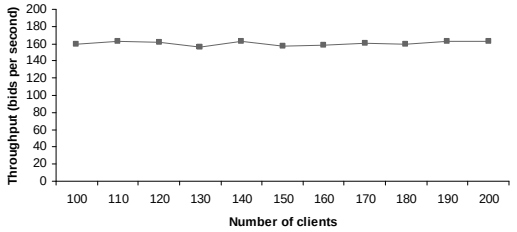
To enable comparative analysis of the performance figures, the CORBA RPC time of a client in Italy communicating with a server (without distribution or reliability objects) in Newcastle was 94 ms (and approximately the same for client in Newcastle and

server in Bologna), the equivalent CORBA RPC between a single client and a local server (e.g., communicating over the same LAN) was approximately 6-7 ms for both Newcastle and Bologna.

Graph 1 - Registering a bid - Single non-replicated server (in Newcastle)



Graph 2 - Throughput - SIngle non-replicated server (in Newcastle)

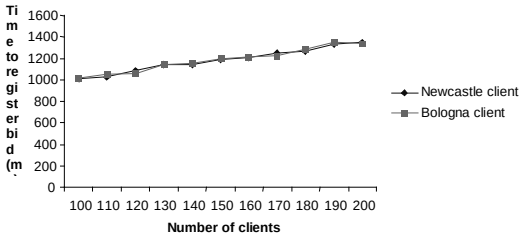


The first observation to be made is the time for a bid sent from a local client (Newcastle) to register in the auction object is far lower than the time taken to register a bid sent from a distant client (Bologna). This is expected, as the latency between the server and the distant clients is approximately 15 times larger than the latency between the server and local clients. Both, local and distant clients take longer to place bids when client numbers are increasing (shown by the upward slope in graph 1); doubling the number of clients doubles the time taken for bids to be registered. This indicates that the server may be overloaded, and this assumption appears to be confirmed by the slightly decreasing slope in graph 2 (throughput).

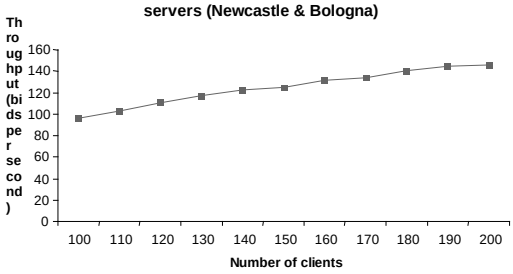
4.2 Distributed Servers, Non-replicated Auction object

Graphs 3 and 4 present the measurements when two servers are present, each serving local clients (Newcastle & Bologna). The time taken to register a bid at both servers is approximately 3 times slower than a client registering a bid at its local server (comparing graph 3 with graph 1). However, in this scenario there are no distant clients. Therefore, no client suffers the extremes of poor performance witnessed when only a single server is present (distant clients 15 times slower than local clients) as shown in graph 1.

Graph 3 - Registering a bid - Distributed servers (Newcastle & Bologna)



Graph 4 - Throughput - Distributed non-replicated servers (Newcastle & Bologna)



Another observation of interest is the degree of the slope of graph 5 compared to that of graph 1. Graph 1 shows a much steeper slope (for both Newcastle and Bologna clients) compared to graph 3. When two servers are present, increasing the number of clients does not have the same adverse effect on time taken to register a bid as when only a single server is present. In graph 6, the slope is still increasing when client numbers are increased, indicating that the server is not yet overloaded (as is the case when only a single server exists – shown in graph 2).

4.3 Replicated Auction Objects (Centralised and Distributed Approaches)

Tables 1 and 2 present the measurements for all experiments. These measurements indicate that the cost of passive replication is low. Replacing a non-replicated server with a passively replicated server increases the time taken to register a bid by approximately 1 – 5% for local clients and less than 1% for distant clients. This can be explained by the manner in which messages are processed by the primary. With reference to figure 5, the multicast to the replica group (M2) of the original bid (M1) is accomplished in parallel with the forwarding of M1 to the auction object (M3). Thus, the overhead of passive replication is the processing of M1 by the

reliability object and the time taken for M3 to be received and processed by the auction object. As the reliability and auction objects are compiled into the same addressable space, the cost of sending M3 is very low.

	Number of clients	100	110	120	130	140	150	160	170	180	190	200
Non-replicated	**Newcastle client**	336	362	397	445	460	512	541	565	603	622	655
	Bologna client	4776	5261	5734	6225	6685	7173	7635	8126	8610	9073	9556
Replicated	**Newcastle client**	341	377	413	443	471	504	566	581	619	664	689
	Bologna client	4778	5263	5740	6230	6693	7172	7666	8127	8626	9085	9593

Table 1. Performance of replicated and non-replicated auction objects for centralized approach

	Number of clients	100	110	120	130	140	150	160	170	180	190	200
Non-replicated	**Newcastle client**	1006	1027	1088	1143	1144	1188	1208	1254	1266	1328	1349
	Bologna client	1015	1051	1061	1138	1146	1194	1211	1223	1285	1346	1336
Replicated	**Newcastle client**	1027	1059	1102	1155	1168	1230	1219	1261	1314	1349	1365
	Bologna client	1049	1099	1121	1161	1209	1238	1250	1305	1340	1398	1427

Table 2. Performance of replicated and non-replicated auction objects for distributed approach

5. Conclusion

In this paper we have described a hierarchic architecture to enable Internet-based applications to satisfy the quality of service and reliability requirements of a large number of geographically dispersed clients. We have demonstrated the effectiveness of our architecture by implementing, and gaining performance measurements from, a distributed auction system.

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 its participation in the group is no longer required. The latter is achieved through the use of group management techniques that 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 unsolvable problem [10] of accurate failure detection in an asynchronous network (e.g. the Internet), which the servers use to communicate with each other.

The performance measurements presented here indicate that our solution achieves scalability, as the total load is shared amongst many servers. Furthermore, by presenting clients with geographically local servers, the overall quality of service provided by Internet-based applications to large numbers of geographically dispersed clients is improved.

6. References

[1]www.time.com/time/daily/0,2960,34249,00.html.

[2] 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.

[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] 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.

[13] 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.

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

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

[16] 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.

[17] 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.

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

[19] http://www.orbacus.com/products/orbacus.html

[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] G. Morgan, S.K. Shrivastava, P.D. Ezhilchelvan and M.C. Little, "Design and Implementation of a CORBA Fault-tolerant Object Group Service", Distributed Applications and Interoperable Systems, Ed. Lea Kutvonen, Hartmut Konig, Martti Tienari, Kluwer Academic Publishers, 1999, ISBN 0-7923-8527-6, pp. 361-374.