CONSTRUCTING DEPENDABLE WEB SERVICES

S K Shrivastava

Rapporteur: Dr Rogério de Lemos

Constructing Dependable Web Services

D. Ingham[1], F. Panzieri[2] and S.K. Shrivastava[1]

[1] *Department of Computing Science*
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK

[2] *Dipartimento di Scienze dell'Informazione*
Universita' di Bologna,
Mura Anteo Zamboni 7, 40127 Bologna, Italy.

Abstract

This paper discusses the issues involved in supporting high-volume, highly-reliable, Web services. Such services pose a number of diverse technical challenges. The paper discusses how recent research ideas from distributed computing can be deployed at the various levels of the architecture to yield an overall solution.

Key words: fault-tolerance, reliability, transactions, process groups.

1. Introduction

The majority of today's Web sites offer read-only access to relatively small amounts of infrequently-changing information. Also, since the load experienced by these sites is usually small, services can generally be hosted as a background task on a general purpose workstation. Such services are generally not overly concerned about the levels of quality of service presented to their users. Conversely, there exists a much smaller number of extremely popular sites that experience very high loads and, in order to maintain their popularity and reputation, tend to be concerned about the quality of service experienced by their users.

The quality of service (QoS) as perceived by the users of a Web service is dependent on a number of factors. Perhaps the most important of these relate to performance and reliability. Users expect services that are continuously available and appear responsive to their requests. A service that is frequently unavailable may have the effect of tarnishing the reputation of the service provider or result in loss of opportunity. Furthermore, from the user's perspective, a service that exhibits poor responsiveness is virtually equivalent to an unavailable service. QoS also encompasses the quality of the information provided, a specific instance being the integrity of hypertext linking between resources.

The users of sites that offer more advanced services, such as electronic shops, personalised newspapers, customer-support systems etc., have additional QoS requirements. The content provided by such services tends to be dynamically generated in response to some read/write interaction between the user and the service. From the user's perspective, it is desirable that the generated content is consistent. Examples of undesirable behaviour include forgetting that the user does not like frames or losing items

from a user's shopping basket. The issue here is data integrity; a service must ensure consistency in the face of concurrent access and occasional system failure. More severe problems can be envisaged for services that involve complex back-office processing. It would not be acceptable, for example, for a component failure within a merchant's service to cause a customer to be billed for a product that was not delivered.

This paper discusses the issues involved in supporting high-volume, highly-reliable, Web services. Such services pose a number of diverse technical challenges. The paper discusses how recent research ideas from distributed computing can be deployed at the various levels of the architecture to yield an overall solution.

2. Problem Understanding

Service providers are looking to computer vendors to provide low-cost, scalable fault-tolerant solutions. The prime requirement is to minimise reliance on specialist equipment and techniques for delivering core services. Indeed, an ideal solution would make use of 'standard' middleware services (e.g., CORBA services for persistence, transactions etc.). Research results on distributed objects and software implemented fault-tolerance techniques hold the promise of providing such solutions. However, the task of constructing such solutions using general-purpose, low cost components, such as commodity UNIX servers, middleware services etc. is extremely challenging.

The central problem is that any software implemented distributed fault-tolerance technique consumes resources (a combination of network bandwidth, processing power and disk storage) that otherwise would be available for normal use. For example, object replication introduces extra messages between replicas (required for replica synchronisation) and message logging introduces either extra messages or disk writes (or both). This frequently makes a fault-tolerant solution unacceptably sluggish (unresponsive) compared to its non-fault-tolerant version. This is particularly so for the case of Web sites: popular Web sites are heavily loaded with client requests, and the last thing that one wants to do is to increase the message traffic. Thus software implemented distributed fault-tolerance techniques must be applied with care. It is therefore important to understand the constraints under which solutions to dependable Web services need to be developed.

Fig. 1 shows a typical non-redundant system, where clients have low bandwidth paths to the Web server. The service will be unavailable to a given client if the server is down, or there is an internet routing problem that prevents the client from contacting the server. The service will not be responsive to a given client if the route is congested or the server is overloaded.

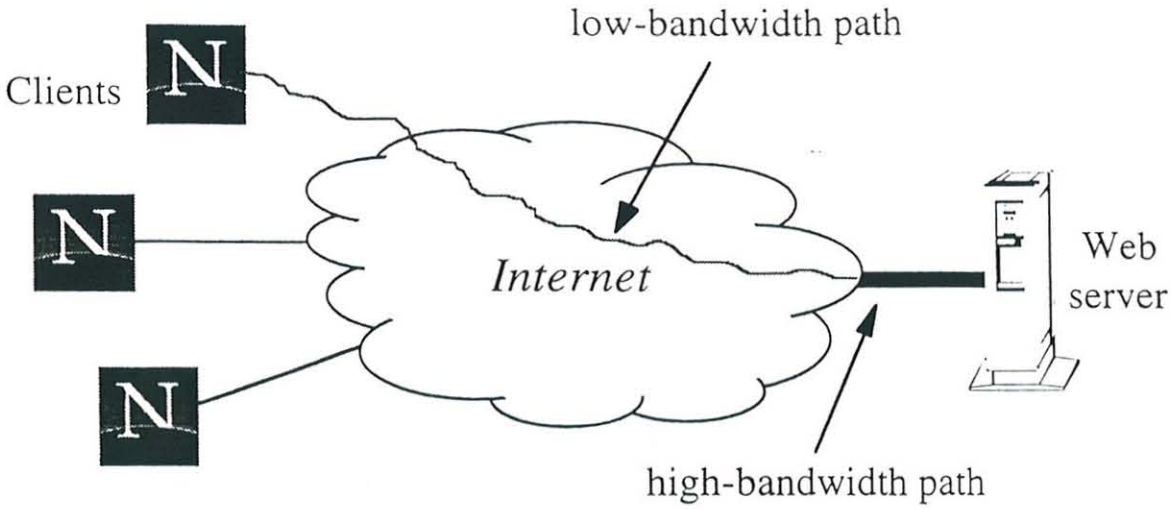

Fig. 1: A non-redundant system

How can the service be made responsive and available? We will assume that message routing and bandwidth allocation within the Internet itself is not entirely under our control, so a practical way of handling unavailability and the unresponsive problems would be to introduce redundancy, namely by replicating the server at distinct sites and ensuring that a client (somehow) gets bound to the 'nearest' lightly loaded server (see fig. 2).

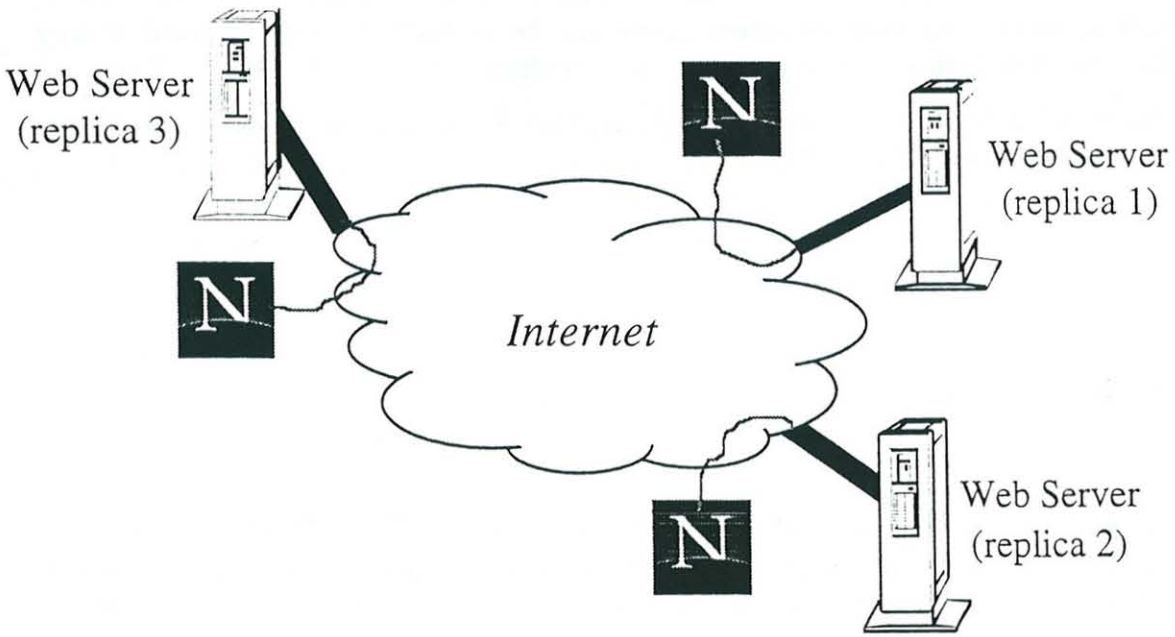

Fig. 2: Redundant system

The success of the above solution will depend on how well we succeed in achieving the following two goals:

(i) Load sharing/distribution: Dynamically binding the client to the 'right' Web site replica. Where 'right' web site choice would be based on: the need to distribute the

may be remote from the invoker by using remote procedure calls (RPCs). All operation invocations may be controlled by the use of transactions which have the well known properties of (i) *serialisability*, (ii) *failure atomicity*, and (iii) *permanence of effect*. Atomic transactions can be nested.

Serialisability ensures that concurrent invocations on shared objects are free from interference (i.e., any concurrent execution can be shown to be equivalent to some serial order of execution). Some form of concurrency control policy, such as that enforced by two-phase locking, is required to ensure the serialisability property of transactions. *Failure atomicity* ensures that a computation will either be terminated normally (*committed*), producing the intended results (and intended state changes to the objects involved) or *aborted* producing no results and no state changes to the objects. This atomicity property may be obtained by the appropriate use of backward error recovery, which can be invoked whenever a failure occurs that cannot be masked. Typical failures causing a computation to be aborted include node crashes and communication failures such as the continued loss of messages. It is reasonable to assume that once a top-level transaction terminates normally, the results produced are not destroyed by subsequent node crashes. This is ensured by the third property, *permanence of effect*, which requires that any *committed* state changes (i.e., new states of objects modified in the transaction) are recorded on stable (crash-proof) storage. A commit protocol is required during the termination of a transaction to ensure that either all the objects updated within the transaction have their new states recorded on stable storage (committed), or, if the transaction aborts, no updates get recorded.

It is assumed that, in the absence of failures and concurrency, the invocation of an operation produces consistent (class specific) state changes to the object. Transactions then ensure that only consistent state changes to objects take place despite concurrent access and any failures.

The above 'object and atomic action model' provides a natural framework for designing fault-tolerant systems with persistent objects [2]. In this model, a persistent object not in use is normally held in a *passive* state with its state residing in an object store or object database and *activated* on demand (i.e., when an invocation is made) by loading its state and methods from the object store to the volatile store, and associating a server process for receiving RPC invocations. Normally, the persistent state of an object resides on a single node in one object store, however, the availability of an object can be increased by storing its state in more than one object store. Transactions can be used for ensuring that states of the replicas remain mutually consistent. A number of replica consistency techniques have been developed [3].

Industry backed Common Object Request Broker Architecture (CORBA) has adopted transactions as the application structuring paradigm for manipulating long-lived objects. Main features of CORBA are: (i) *Object Request Broker (ORB)*, which enables objects to invoke operations on objects in a distributed, heterogeneous environment. Internet Inter-ORB-Protocol (IIOP) has been specified to enable ORBs from different vendors to communicate with each other over the Internet. (ii) *Common Object Services*, a collection of 'middleware' services that support functions for using and implementing objects. Such services are considered to be necessary for the construction of any

distributed application. These include transactions (the Object Transaction Service), concurrency control, persistence, and many more [4].

4.2. Process Groups

Process groups with ordered group communications also provide a set of facilities for building available distributed applications. The building of such applications is considerably simplified if the members of a group have a mutually consistent view of the order in which events (such as message delivery, process failures) have taken place. Design and development of fault-tolerant group communication protocols for distributed systems satisfying certain order properties has therefore been an active area of research [e.g., 5,6,7]. Below we present some relevant concepts pertaining to process groups.

A *group* is defined as a collection of distributed processes in which a member process can communicate with other members by multicasting to the full membership of the group. A given process can be a member of more than one group. Let $g = \{P_1, P_2, \ldots P_n\}$ be a process group. When $P_i \in g$ multicasts (or delivers) a message *m* it actually does so only to (or from) those processes which it *views* as functioning members of g. P_i delivers its own messages also by executing the protocol in operation. We require the property that members of a group deliver identical messages in identical order. In particular, this means that a given multicast is *atomic*: either all the functioning members are delivered the message or none. Clearly, this would be an ideal property for replicated data management: each process manages a copy of data, and given the above property, it is easy to ensure that copies of data do not diverge. However, achieving this property in the presence of failures is not simple.

For example, a multicast made by a process can be interrupted due to the crash of that process; this can result in some connected destinations not receiving the message. Process crashes should ideally be handled by a fault-tolerant protocol in the following manner: when a process does crash, all functioning processes must promptly observe that crash event and agree on the order of that event relative to other events in the system. In an asynchronous environment this is impossible to achieve: when processes are prone to failures, it is impossible to guarantee that all non-faulty processes will reach agreement in finite time [8]. This impossibility stems from the inability of a process to distinguish slow processes from crashed ones. Asynchronous protocols can circumvent this impossibility result by permitting processes to *suspect* process crashes and to reach agreement only among those processes which they do not suspect to have crashed.

A process group therefore needs the services of a *membership service* that executes an agreement protocol to ensure that functioning processes within any given group will have identical views about the membership. When g is initially formed, each functioning P_i installs an initial view V^0_i, say, $V^0_i = \{P_1, P_2, \ldots P_n\}$. If P_i is unable to communicate with some $P_k \in V^0_i$ its membership service installs a new view that does not include P_k. Let $V^0_i, V^1_i, V^2_i, \ldots V^r_i$ be the series of views P_i has thus sequentially installed over a period of time, until it crashes or leaves the group g. The membership service ensures that the sequence of views installed by any two functioning member processes of g that do not suspect each other are identical. View updates must not interfere with normal multicasts:

we require that message delivery be 'atomic' with respect to view updates. As a consequence, any two functioning processes deliver the same set of messages between two consecutive views that are identical. This atomic property has been called *virtual synchrony* [5]. Fig. below illustrates this: a crash occurs during a multicast; view V^2 is installed either after (fig. 5(a)) or before (fig. 5 (b)) the delivery of the multicast message. Virtual synchrony enables dynamic control over group membership: new members can be brought in without causing interference with on-going multicasts.

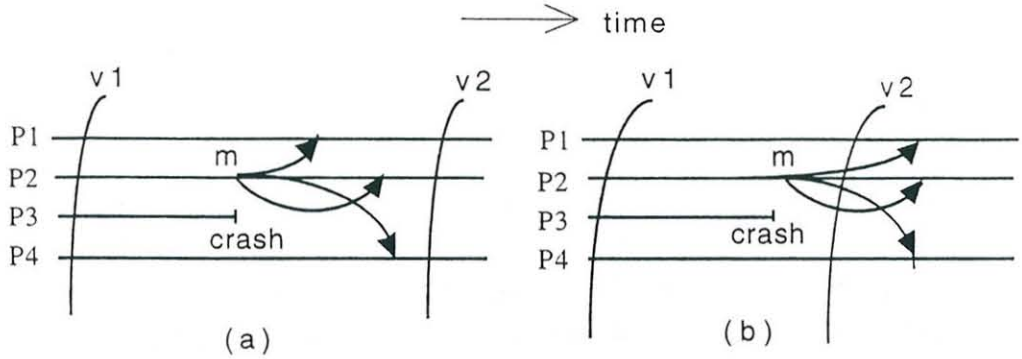

Fig. 5: Virtual synchrony

Finally, a few words on the treatment of partitions. Despite efforts to minimise incorrect suspicions by processes, it is possible for a subgroup of mutually unsuspecting processes to wrongly agree (though rare it may be in practice) on a functioning and connected process as a crashed one, leading to a 'virtual' partition. There is thus always a possibility for a group of processes to partition themselves (either due to virtual or real network partitioning) into several subgroups of mutually unsuspecting processes. Modern membership services are capable of maintaining view consistency in the presence of real or virtual partitions by ensuring that: (i) the functioning processes within any given subgroup will have identical views about the membership; and (ii) the views of processes belonging to different subgroups are guaranteed to stabilise into non-intersecting ones.

Process groups can be implemented as a middlware service, and there are many research efforts to build such a service on top of ORBs. Unlike transactions, no standard has yet been developed, a situation that we expect will change in the near future.

5. Applications of Transactions and Process Groups

We discuss below how transactions and process groups can be used for providing better class of solutions.

5.1. Fault-tolerant clusters

Process groups provide a generic solution to decentralised configuration management of arbitrarily large processor clusters. Membership service, at the granularity of processors, can be used for enabling each functioning processor to maintain mutually consistent membership and processor load information. Any deterministic algorithm can be used by each of the processors to determine how the incoming requests can be shared.

In a simple scheme, the router/gateway (that uses NAT technique) translates the incoming packet addresses to a broadcast address and broadcasts them on the cluster LAN, and can leave it to the machines to decide who should serve the request. An alternative scheme would require the router/gateway also to be a member of the processor group, and thus maintain membership and load information; based on this information, the router can forward the incoming request to a member processor.

5.2. Wide area load distribution

The techniques discussed above can also be used for creating general purpose, open solutions for wide area load distribution in place of rather specialised, proprietary solutions exemplified by the DistributedDirector product discussed earlier. The DNS server and Web servers can be made members of a group to enable the DNS server to maintain membership and load information. This way the probability of the server directing requests to failed or overloaded Web servers is minimised. A generalisation is possible where by a number of DNS servers can be incorporated in the group for maintaining mutually consistent membership and load information, therby obtaining tolerance against DNS server failures and partitions.

5.3. Replica management

Object replicas must be managed through appropriate replica-consistency protocols to ensure that object copies remain mutually consistent. Consistency could be either *strict* (an update at any replica is propagated to other copies 'straight away'), or *lazy* (updates are propagated in background). A major advantage of strict consistency is that clients always get consistent, fresh information. Unfortunately, strict consistency reduces update performance, so does not scale well. Lazy consistency on the other hand can scale well, but freshness of information at any given replica cannot be guaranteed.

Any practical system is likely to contain a mixture of the two [9]. For example, one could imagine maintaining strict consistency within a 'primary' cluster, with remaining clusters being updated lazily. However, certain data items across all the replicas may well need to be kept strictly consistent. Lazy updates could be carried out as a series of transactions initiated by the primary. Both transactions and process groups provide complementary mechanisms for implementing replica consistency. Nevertheless, work is required in developing scalable mixed consistency solutions.

5.4. End to end reliability

So far in our discussions, we have concentrated on issues concerning reliability of Web servers. However, this is only a part of the story. Typically, a distributed application will also involve processing at a client's side, so issues of client side reliability need to be taken in to account. For example, if a user purchases a cookie (a token) granting access to a newspaper Web site, it is important that the cookie is delivered and stored if the user's account is debited; a failure could prevent either from occurring, and leave the system in an indeterminate state. Providing *end-to-end transactional integrity* between the client (browser) and the Web server is important: in the previous example, the cookie *must* be delivered once the user's account has been debited. Providing such a guarantee was

difficult with the original "thin" client model of the Web, where browsers were functionally barren. With the advent of Java it is now possible to empower browsers so that they can fully participate within transactional applications [10].

5.4. Advance application building environments

Middleware services such as CORBA services referred to earlier provide generic facilities for the construction of fault-tolerant distributed applications in the Internet environment. A number of factors need to be taken into account in order to make these applications fault-tolerant.

First, most such applications are rarely built from scratch; rather they are constructed by composing them out of existing applications. It should therefore be possible to compose an application out of component applications in a uniform manner, irrespective of the languages in which the component applications have been written and the operating systems of the host platforms. Second, the resulting applications can be very complex in structure, containing many temporal and data-flow dependencies between their constituent applications. However, constituent applications must be scheduled to run respecting these dependencies, despite the possibility of intervening processor and network failures. Third, the execution of such an application may take a long time to complete, and may contain long periods of inactivity (minutes, hours, days, weeks etc.), often due to the constituent applications requiring user interactions. It should be possible therefore to reconfigure an application dynamically because, for example, machines may fail, services may be moved or withdrawn and user requirements may change.

Recent work on *transactional workflow systems* has shown that they provide the right set of facilities for application composition and execution enabling sets of inter-related tasks to be carried out and supervised in a dependable manner [11,12]. Further, they can be designed and implemented as a set of CORBA services to run on top of a given ORB. Wide-spread acceptance of CORBA and Java middleware technologies make such systems ideally suited to building dependable Internet applications.

6. Concluding Remarks

We have reviewed current approaches to building high-volume, highly-reliable, Web services. These approaches either use proprietary solutions and/or *ad hoc* techniques that do not scale well. Service providers are looking to computer vendors to provide low-cost, scalable fault-tolerant solutions. The prime requirement is to minimise reliance on specialist equipment and techniques for delivering core services. We have discussed how software implemeted fault-tolerance techniques (transactions and process groups) can be applied for creating scalable solutions.

Acknowledgements

This work has been supported in part by Hewlett-Packard Laboratories, Bristol, the Italian Consiglio Nazionale delle Ricerche (CNR), the Italian Gruppo Nazionale Informatica e Matematica (GNIM) and ESPRIT LTR working group Broadcast (project No. 22455).

References

[1] Microsoft Corporation, "How Microsoft manages www.microsoft.com," Microsoft TechNet, vol.5 no.6, June 1997. Available at <URL:http://www.eu.microsoft.com/syspro/technet/tnnews/features/mscom.htm>

[2] G.D. Parrington, S.K. Shrivastava, S.M. Wheater and M.C. Little, "The design and implementation of Arjuna", USENIX Computing Systems Journal, vol. 8 (3), pp. 255-308, Summer 1995.

[3] P.A. Bernstein et al, "Concurrency Control and Recovery in Database Systems", Addison-Wesley, 1987.

[4] R. Orfali, D. Harkey and J. Edwards, "The essential distributed objects", John Wiley and Sons Ltd., 1996.

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

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

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

[8] M. Fischer, N. Lynch and M. Paterson, "Impossibility of Distributed Consensus with One Faulty Process", J. ACM, 32, pp 374-382, April 1985.

[9] J. Gray, P. Helland, P. O'Neil and D. Shasha, "The dangers of replication and a solution", ACM SIGMOD Record, 25 (2), pp. 173-182, June 1996.

[10] M.C. Little and S. K. Shrivastava, "Java Transactions for the Internet", 4th USENIX Conf. on Object Oriented Technologies and Systems, COOTS, Santa Fe, April 1998.

[11] D. Georgakopoulos, M. Hornick and A. Sheth, "An overview of workflow management: from process modelling to workflow automation infrastructure", Intl. Journal on distributed and parallel databases, 3(2), pp. 119-153, April 1995.

[12] F. Ranno, S.M. Wheater and S.K. Shrivastava, "A System for Specifying and Co-ordinating the Execution of Reliable Distributed Applications", Distributed Applications and Interoperable Systems, eds: H. Konig, K. G. Geihs and T. Preuss, Chapman and Hall, ISBN 0 412 82340 3, pp. 281-294, 1997.

DISCUSSION

Rapporteur: Dr Rogério de Lemos

During the talk Professor Vogt asked how a client was able to know whether a particular server replica had the lowest bandwith, unless the choice of server could be hidden from the client. Professor Shrivastava answered that the configuration that was described was a system that one would like to have, and for that, it should incorporate some of the features that he had previously referred to as 'magic'. He continued by saying that there were a lot of techniques to implement the magic, however, in practice they have not worked very well.

Professor Lobelle questioned whether the assumption of having few hundreds of replicas was realistic since what is usually needed is at most half a dozen replicas. Professor Shrivastava agreed with the statement saying that the need for maintaining consistency in a few hundred of replicas, perhaps, might not be a problem.

Mr Kay asked if it was not the case that load balance could be achieved by exploiting the randomness of the workload. Professor Shrivastava agreed with the comment and added that one hoped that it would work because there was no means to control load balancing.

While Professor Shrivastava was talking on how to maintain mutually consistent copies of data, Professor van Roy queried whether the strict consistency could be considered synchronous while lazy consistency was asynchronous. Professor Shrivastava agreed with the statement, and stressed that any practical solution would involve a mixture of the two.

Mr Kay asked if during a transaction service, once a transaction has started and suddenly there was a line break, the user would not know whether the transaction had successfully finished or not. Professor Shrivastava replied that that was what he meant by "end-to-end reliability" because it was not enough to start a transaction and forget about its outcome, there was a need to implement on either side of a transaction the correct protocols. Mr Kay then enquired whether it was not the case of having the screen involved with the transaction, to which Professor Shrivastava agreed.

After the talk Professor Kopetz asked to what extent the presented middleware technique was consuming resources that otherwise would be available for normal use. Professor Shrivastava answered that at the middleware level the problem was not computing power but on message traffic.

Professor van Roy enquired what kind of transaction support within Java was being advocated because Java does not have any "hooks" to support transactions. Professor Shrivastava answered that there already existed a Java standard JTS that complies with CORBA OTS almost one to one, allowing the transaction services to be implemented in Java.

Professor Vogt asked how easy it was to extend the cluster previously presented with more workstations. Professor Shrivastava answered that although he had no first hand experience in doing this, he believed that if a system, such as ISIS which gives group membership services, is used, then the solution would be straight forward, however, the real problem would be to put the application together. Professor Vogt continued by asking whether the problem was being transferred to the application level. Professor Shrivastava replied that he had presented a collection of techniques that could be used, rather than giving a specific solution.

Professor Randell enquired whether the provision of dependable services had been exacerbated by early decisions on Web protocols, which were taken without exploiting what was already known from the world of transaction processing. Professor Shrivastava answered that they did not cause any deep technical problem, he was of the opinion that the real problems were at application level, or at the intra-trust level: for example, to obtain the right business models, which perhaps was not a computing science problem.