*Managing Data Redundancy in Interoperable Heterogeneous Environments**

Hassan N. Srinidhi Bellcore

ABSTRACT: The need for interoperable heterogeneous systems has been recently realized and many international, national and industry standards activities have been launched to achieve this goal. These interoperable heterogeneous systems should have the capability to interoperate with other systems irrespective of their suppliers and vintages. The need for interoperability among telecommunication systems has been recognized in Bellcore and has lead to the development of a software architecture for interoperability called the *OSCA*™ *architecture*.[1] The OSCA architecture separates business processes into three layers, called the *data layer,* the *processing layer* and the *user layer,* to maximize the reuse and accessibility of the functionality within each of these layers. Each of these layers are realized using multiple deployable units called *building blocks* that offer well-defined functional interfaces called *contracts* to other building blocks. The OSCA architecture requires a set of principles to be followed by the building blocks of each layer. Specifically, the functionality of the *data layer building blocks* (DLBBs) is to provide

1. OSCA is a trademark of Bellcore.

*An earlier version of this paper appeared in the IEEE Computer Society Proceedings of the RIDE-IMS '93 Workshop. We are grateful to the IEEE Computer Society for permission to include passages that previously appeared there.

open access to complete and consistent corporate data, maintain semantic integrity of all corporate data and manage redundant copies of the corporate data. Each DLBB is designated as the *steward* of some corporate data and is responsible to provide open access to that data, maintain semantic integrity of that data, and manage redundant copies of that data. The stewarded corporate data is always considered correct and consistent, and is the primary source of updates for all redundant copies of that data in other building blocks.

This paper addresses the problem of managing redundant copies of corporate data by a steward when these redundant copies are supported in heterogeneous environments in other building blocks. This paper shows the desirable categories of redundant data, establishes architectural restrictions on redundant copies, and describes practical management schemes and algorithms to manage redundant copies.

The paper briefly describes the motivation for managing redundant data using the OSCA architecture, the types of redundant data possible in the OSCA architecture and the OSCA architecture principle to manage redundant copies of corporate data. The paper then establishes the problem constraints for redundancy management in heterogeneous OSCA environments, and identifies the issues in managing redundant data. Addressing these issues, the paper describes a set of architectural rules for managing redundant data in the context of the OSCA architecture, and presents two practical management schemes and an algorithm for managing redundant data in OSCA environments. The paper also addresses the required contracts for redundancy management and the extent of data consistency that can be realized using various transaction models. The data redundancy management rules, the management schemes and the algorithm presented in this paper provide a practical solution for managing redundant data in large corporations.

---

1. Introduction

Data redundancy has been used in the automated systems of large corporations as a means to achieve improved performance, higher availability and higher reliability. The performance gains due to redundant data have been best realized in an environment where application programs frequently read data and seldom update the data. In this case the version of the data read by an application program may not be very important. However when updates are frequently made to the data, the level of consistency between redundant copies of data or between related data becomes very important for reliable business processing.

A fundamental fact today is that a significant amount of redundant information is resident within the automated systems of large corporations. This fact alone does not fully characterize the problem; rather the problem is worse because this redundant information frequently becomes inconsistent and requires frequent corrections to restore consistency. The data inconsistency problems that exist today can be attributed to many reasons:

- Using manual entries to input information into systems or to reconcile data inconsistencies between islands of automation. This often results in conversion errors or entry errors.
- Lack of appropriate semantic checks on data either in the application programs or in the database management systems. Although current database management systems are improving in providing much-needed semantic capabilities, there is very little semantic support across platforms, much less across heterogeneous platforms. Much of this support must still be built into the application programs, but is frequently missing.

- Close coupling of data with business processing functions, thus providing only application-specific views on the data rather than providing a generic (application-independent) access to the data as required by an information model.[2] This encourages different applications to maintain their own redundant copies of the same data. Maintaining redundant data consistency between these copies is hard because of a lack of automated means of update and because application-specific formatting and views are imposed on the data.

The typical approach used in automated systems when data inconsistencies are found is to "request for manual assistance". Initially this was a good decision since the humans are well versed at recognizing and doing something to resolve the data inconsistencies. Unfortunately, with the increased volume in processing, the automated systems produce and/or detect inconsistencies very rapidly requiring many humans (in the order of thousands of employees in large corporations) in order to resolve these inconsistencies. This reduces the overall value of the automated systems. As the number of automated systems is ever increasing, the data redundancy among them is also increasing. It is crucial to manage redundant data across systems and maintain required levels of consistency between data across systems in large corporations. If this is not done, the delays due to inconsistencies will result in lost revenue and reduced levels of service.

In order to manage redundant data in large corporations, it is essential:

- to be able to identify complete and consistent sources of corporate data, and support them in appropriate vendor platforms that satisfy the performance, availability and reliability requirements of these corporate data sources
- to have architectural rules (restrictions) on how redundant copies may be obtained from these corporate data sources and used
- to provide well-defined open interfaces to access and maintain both the corporate data sources and the redundant copies of corporate data and

2. The term *information model,* also called a logical data model (LDM), is used to indicate an extended entity-relationship (EER) model which includes precise definitions of entity meta-types in terms of their sets of available operations.[18] Thus it can be regarded as an object model, since it supports modeling of data operations as well as data structures.[9][10]

- to identify mechanisms by which the defined consistency requirements on the redundant copies can be maintained.

The needs mentioned above are best realized in the context of open systems[1] that interoperate with other open systems because it is often the case that the corporate data sources and the redundant copies are supported on different vendor platforms that satisfy the roles played by these data. For instance, the corporate data sources may be supported on a high performance mainframe machine while the redundant data may be supported on a less-expensive work station. In general, because of diverse needs of applications in large corporations, there is a need to support data on a wide variety of computing environments, to allow a wide choice of data base management systems, and to be compatible with a variety of data architectures, but yet have interoperability among the systems built on these diverse environments. The Bellcore OSCA architecture[2] is an interoperability architecture that supports this idea and has been developed to satisfy the interoperability needs among telecommunication systems. Section 1.1 describes the OSCA architecture, the types of redundant data in the OSCA architecture and the OSCA architecture principle to manage redundant copies of corporate data.

1.1 The OSCA Architecture

The OSCA architecture[2] is an implementation-independent system design framework intended to provide corporations the flexibility to combine software products in ways which best satisfy their business needs and to provide access to corporate data by all authorized users. It does this by promoting the interoperability of software products that consist of large programs, transactions and databases. Interoperability is the ability to interconnect software products irrespective of their suppliers and vintages, to provide access to corporate data and functionality by any authorized user, and to maintain that interconnection and access over changes in suppliers and vintages.

Fundamental to the OSCA architecture is the requirement of applying the notion of separation of concerns to business aware functionality. Business aware functionality is functionality that is characteristic of a business. It relates to an understanding of the business, such as providing information describing the business or performing processes characteristic of the business. The notion of *separation of concerns* requires that business aware functionality be separated

(grouped) into the "layers"[3] or categories of corporate data management functionality (*data layer*), business aware operations and management functionality (*processing layer*), and human interaction functionality (*user layer*), to maximize the reuse and accessibility of the functionality within each of these layers. The software which implements the functionality in each layer is partitioned into *building blocks,* and these building blocks must adhere to specific principles described in the technical advisory on OSCA architecture.[2] Thus the data layer is partitioned into *data layer building blocks* (DLBBs),[3] processing layer into *processing layer building blocks* (PLBBs) and user layer into *user layer building blocks* (ULBBs). Each DLBB is said to *steward* an allocated portion of the corporate data. Stewardship implies that the DLBB is solely responsible for providing the complete, consistent, and semantically valid value(s) of the data it stewards to all authorized requests. Specifically, DLBBs must ensure semantic integrity of the stewarded data, manage redundant copies of that data and provide open access of the stewarded data to all authorized users.

The notion of separation of concerns in the OSCA architecture also requires that no building block will contain business aware functionality belonging to more than one layer. The building blocks are supported by a platform of business-independent functions (called *infrastructure services* that includes DBMSs, distributed transaction processing services, directory services, etc.) and interact with each other using well-defined interfaces called *contracts*. A contract is defined by a *contract specification,* which is a document precisely defining the functionality and the way in which the functionality provided by a contract is *invoked,* and the support commitment(s) that are available. Contract specifications must be delivered with the building block providing the specified contracts. Contract specifications describe contracts in terms of functionality, interface syntax and semantics, response time, availability, transaction paradigms supported, and like details. Contract specifications are implementation-independent and can possibly be implemented using a variety of implementation technologies. The building blocks in the OSCA archi-

3. OSCA architecture layering does not correspond to OSI layering; no hierarchy is implied. For example, the user layer can communicate with the data layer without passing through the processing layer.

tecture do not need to know the implementation details of other building blocks in order to successfully interoperate; they only need to be able to invoke the contracts offered by other building blocks using standard interface notations (for example, Abstract Syntax Notation One, ASN.1[4]) indicated in their contract specifications. In this manner the implementation of each building block is isolated from the other allowing building blocks in heterogeneous environments to interoperate. Thus a stewarding DLBB can update a redundant copy in another building block using an update contract defined on the redundant copy even if the data in the two building blocks are supported using heterogeneous database management systems. An update contract invocation on a redundant copy may result in a single or multiple commit actions depending on what is allowed by the contract specifications and what is intended by the particular contract invocation.

For more details on the OSCA architecture, the reader is referred to the technical reference on the Bellcore OSCA architecture[2] and other related papers.[5] [6] [7]

1.1.1 Types of Redundant Data in the OSCA Architecture

The OSCA architecture recognizes *private redundant data* that are owned within individual building blocks. Private redundant data is a replicated copy or a partially replicated copy of some stewarded data obtained from the stewarding DLBB at some point in time, and having well-defined consistency requirements with respect to that stewarded data. The redundant data are owned by a building block and *should not be visible* for general retrieval and updating purposes outside the building block owning it.

The intent of private redundant data is to give applications relevant data that they can use and modify at will. For example, planning systems (PLBBs) may get a private redundant copy of current telephone network traffic information and perform projections on this information for what-if studies that determine future loads on telephone networks. In the process these systems may manipulate their copy in any manner they like. Obviously these changes should not be shown to other building blocks and hence should be private to the planning systems. If another building block requires this projected load and network char-

acteristics information, then a steward should be designated to provide that information. This steward should provide the data after appropriate semantic checks on the information.

Besides application-specific needs mentioned above, private redundant copies may also be used to satisfy performance or availability requirements that cannot be satisfied by directly accessing the information from the steward.

Private redundant data may occur in any layer of the architecture including the corporate data layer. We use the term *private redundant copy* in this paper to refer to private redundant data owned by a specific building block. From the point of view of controlling the extent of data redundancy, private redundant copies are strongly discouraged and the need for any such copy should be justified.

The OSCA architecture also recognizes *shared redundant data* to meet performance, availability, or alternate view needs. This is a replicated copy or a partially replicated copy of some stewarded data obtained from the stewarding DLBB at some point in time, and having well-defined consistency requirements with respect to that stewarded data. The shared redundant data are housed[4] in a DLBB and are *visible only for general retrieval* outside the building block housing that data, i.e., they can only be read. We use the term *shared redundant copy* in this paper to refer to shared redundant data housed in a specific DLBB.

There are many reasons for not allowing building blocks to directly update a shared redundant copy:

- First, by definition, a shared redundant copy can be a partial replicate and may not necessarily be always consistent with the stewarded data. It would be meaningless to directly apply updates to a copy that is already out of date and moreover these updates cannot be shared with other building blocks unless the stewarding DLBB accepts these updates.
- Second, the stewarding DLBB provides a point of concurrency control for all updates on the stewarded data. So an additional level of delay is avoided by sending the updates directly to the

4. Since the DLBB with shared redundant data should not make updates on that shared data independent of the stewarding DLBB (as in the case of ownership of the data) nor does it steward the data, we say the data is housed in the DLBB to imply neither owned nor stewarded data.

stewarding DLBB instead of sending updates to a shared redundant copy.

- Third, the implementation of integrity constraints in a shared redundant copy may differ from that of the stewarding DLBB and further additional constraints may also be imposed on the data. For example, a shared redundant copy may be a partially replicated copy having information only about telephone circuits of large centrex customers (say, with more than 500 telephone lines) in the New York city while the steward may have telephone circuits information of all residential and business customers in the entire state of New York. The shared redundant copy rejects all updates which do not satisfy these restrictions eventhough the update may be valid for the steward. Thus a shared redundant copy should not decide whether to accept or reject an update. That would have to be the stewarding DLBB's decision.
- Fourth, even if the integrity constraints implemented in a shared redundant copy are identical to that of the steward, the steward cannot even accept a tentative update validation performed by a shared redundant copy as it would create a dependency between the steward and that shared redundant copy resulting in future release dependencies and accessibility assumptions. If the integrity constraints are changed in the steward and not in the shared redundant copy, or vice versa, then this discrepancy would cause confusion to the users. Hence it is not desirable to perform validations in a shared redundant copy.

Shared redundant data is housed only in DLBBs for two reasons. First, the security and integrity requirements associated with the corresponding portion of the corporate data should be observed by the building block housing the shared redundant data, as the building blocks sharing this data assume the data to be accurate. Secondly, the principle of separation of concerns in the OSCA architecture stresses that the data layer that should be providing the functionality of all shared corporate data. Shared redundant data does not cease to be corporate data just because it is redundantly copied, and hence must be provided in a DLBB.

Shared redundant data may lag in consistency with stewarded data according to some predefined schedule. However, there are advantages of having shared redundant data:

- First, it provides another point of access to (a copy of) corporate data besides the stewarding DLBB for that corporate data. This reduces the load on the stewarding DLBB and could solve performance and availability problems of the stewarding DLBB for retrievals, and may provide an overall load balance.
- Second, certain building blocks may not need the level of consistency provided by the stewarding DLBB, or may need an alternate view (including partial views) or a distinct temporal view of data which can be satisfied with shared redundant data.
- Third, the total extent of redundant data is reduced as several building blocks can use the shared redundant data instead of obtaining individual private redundant copies. This is especially true if a shared redundant copy is installed in the same environment as the building blocks needing a redundant copy such that the shared redundant data can be accessed with minimum communication delays.
- Fourth, as a consequence of the reduced number of private redundant copies of data, the number of copies the stewarding DLBB has to update is reduced thus reducing the load on the stewarding DLBB.
- Fifth, a DLBB may house shared redundant data from more than one stewarding DLBB, providing an aggregate view.

The term redundant data is used to mean shared and private redundant data collectively. The OSCA architecture allows redundant data to be physically present among the various building blocks, whenever necessary, for reasons of performance, availability, supporting alternate views, or supporting inter-DLBB semantic integrity constraints.[5] Possible consistency requirements for redundant data are that the redundant data must be in *lock-step synchronization, eventually consistent,* or in *lagging consistency*[11] with the corresponding portion of corporate data. Lock-step synchronization means that any change to the stewarded data must be reflected in the redundant copy before the change to the corporate data is made available (for example, two-phase

5. Semantic integrity constraints are relationships between data that must always hold irrespective of the business aware processes manipulating the data. These relationships are typically captured in an information model. Relationships between data in different DLBBs is called *inter-DLBB semantic integrity constraints* and relationships between data in the same DLBB are called *intra-DLBB semantic integrity constraints*.

commits can be used to achieve this). Eventually consistent means that any change to the stewarded data will be made available prior to the change being reflected in the redundant copy, but the changes will be propagated to the redundant copy in a short time, for example, in a few seconds to a few minutes. The consistency synchronization window can be flexibly defined to take place within specific time periods (for example, within a few minutes) or at specific events (for example, after every ten updates). Lagging consistency is a degenerate case of eventual consistency in which the redundant data may never be consistent with the corresponding portion of corporate data, with propagation delays amounting to several minutes, hours, or even days. The consistency synchronization window here can be flexibly defined to take place at specific time intervals, at specific time points, at specific events, or on demand from the BB having a redundant copy.

1.1.2 The OSCA Approach to Redundant Data

The objective of the OSCA architecture is to eliminate data redundancy at the logical level, and minimize and manage it at the physical level. In order to accomplish this objective, the OSCA architecture has the following DLBB principle called managing redundancy:

> *Each DLBB must provide means whereby updates to the corporate data that it stewards can be passed to building blocks having redundant copies of that data, and must not propagate updates received from another stewarding DLBB for any of its shared redundant data.*

This paper elaborates on the redundancy management rules, and describes two schemes and an algorithm for managing redundant data.

1.2 Outline of the Paper

Section 2 of this paper describes the constraints and identifies the various issues in managing redundant data. Section 3 lays out the rules for managing redundant data. Section 4 presents two schemes for managing redundant data that can be used for satisfying lock-step, eventual and lagging consistency requirements on redundant data, and an algorithm that can be used to satisfy eventual or lagging consistency on redundant data. Section 5 presents the conclusions. Most of the material presented in this paper can be found in a Bellcore Special Report addressing this topic.[13]

2. *The Problem Domain*

The solutions proposed in this paper for managing redundant data in OSCA environments assume the following constraints:

- System requirements: Different systems have different performance, accuracy and reliability requirements depending on their functionality. These in turn dictate, respectively, the availability, consistency and recovery requirements of the data (be it the corporate data or the redundant data) with which the systems are working.
- Environment of the redundant data: The redundant data, in relation to the stewarded data environment, may be supported in a distinct homogeneous data base management system (HoDBMS) environment, in a heterogeneous DBMS (HDBMS) environment, in a distributed DBMS (DDBMS) environment, or in the same environment as the corporate data (which is not often the case). Since all the DBMS environments do not yet support compatible transaction processing or concurrency mechanisms, the data environment influences the solutions proposed for the problem of managing redundant data.
- Autonomy of building blocks: The OSCA architecture principles provide *local autonomy*.[6] This includes communication autonomy (freedom to schedule a response to a request when invoked via its defined contracts) and execution autonomy (freedom to set execution priorities for different categories of contracts invoked from different building blocks).

Given the above constraints, this paper addresses the following issues related to data redundancy management in the context of the OSCA architecture. For each issue, we provide cross-references to one or more subsections that address the issue:

1. The architectural constraints that should be followed in managing data redundancy (see Section 3).
2. The transaction models useful for redundancy management (see Section 4.2).

6. The term local autonomy has been defined differently in Du and Elmagarmid.[12] We borrow the term here, but define it slightly differently in relation to building blocks.

3. The management schemes useful for maintaining lock-step consistency, eventual consistency or lagging consistency in redundant data (see Section 4.3).
4. Maintaining data consistency to suit the performance, accuracy and reliability requirements of various classes of applications (see Section 4.4).

3. Rules for Managing Redundant Data

This section presents rules to manage redundant data with necessary explanations. Most of these rules are also found in the OSCA Technical Reference.[2]

Rule 1 - If access requirements cannot be met by the stewarding DLBB, then shared redundant data should be used whenever applicable and practical, and in preference to private redundant copies.

Shared redundant data may eliminate the need for private redundant copies, and should be considered as a first choice alternative prior to considering private redundant copies. Shared redundant data may provide less timely consistent data than stewarded data and therefore the choice of shared redundant data must consider any risk to the accuracy and integrity of results of processes using the shared redundant data.

Rule 2 - Only updates made to the stewarded data are valid updates.

Given that a stewarding DLBB is responsible for the semantic integrity of its stewarded data, it is appropriate to say that only those updates that have been made to the stewarded data are taken to be correct. If updates on private redundant copies have to be also made on the stewarded data, then the building block with the private redundant copy will have to send those updates to the stewarding DLBB. The stewarding DLBB will be the final judge to decide if these updates are appropriate or not. If the updates are committed in the private redundant copy and the stewarding DLBB does not accept them, then the building block with the private redundant copy may choose to rollback to its previous state thus undoing the action of the updates. Shared redundant copies provide only retrieval contracts to other building blocks by definition, and the question of updates does not arise.

Rule 3 - Shared redundant copies are obtained only and directly from the stewarding DLBB.

There are two reasons for obtaining a shared redundant copy directly from the stewarding DLBB and not from another DLBB holding a shared redundant copy. First, the DLBB obtaining the shared redundant data, by definition, is responsible for the security and integrity of such data. Since the DLBB stewarding the portion of corporate data has the complete and current security and integrity information on that data, the correct information is provided to the DLBB obtaining the shared redundant copy from the stewarding DLBB. Secondly, this approach establishes the required consistency criterion for the data in the shared redundant copy that is to be provided by the stewarding DLBB.

The installer of a shared redundant copy negotiates with the administrators of the stewarding DLBB for the needed consistency criterion for that shared redundant copy. The consistency requirements of the shared redundant copies can be maintained within the stewarding DLBB or in an infrastructure service that assists in redundancy management (for example, in a redundancy management service or strategically within the DBMSs themselves). Maintaining consistency requirements in an infrastructure service that manages redundancy is the preferred solution.

Rule 4 - A shared redundant copy does not propagate updates to other private redundant or shared redundant copies.

There are many important reasons not to allow propagation from shared redundant copies:

- Shared redundant copies need not be full replicates and need not support the same set of integrity constraints, they can impose additional ones. Also, they need not be in sync with the stewarded data. This means that private or shared redundant copies obtained or maintained from other shared redundant copies must have similar view and integrity requirements, and must have lesser consistency needs with respect to stewarded data. If we extrapolate this chain, we see a series of redundant copies with increasing fragmentation and increasing lag, a differential that is not quite tractable. The whole scheme can go haywire especially when failures happen in any part of the chain

and the required update frequency commitments cannot be met. It is true that failures can happen in the stewarding DLBB, but a stewarding DLBB can be designed to handle such failures, and since the path length from the stewarding DLBB to a copy is just one, we have a lesser probability for failures.

- A redundancy chain like the one described above can potentially make the data redundancy problem to proliferate as the stewarding DLBB loses control over the copies and innumerable private redundant copies may be made from shared redundant copies as they can bypass the required negotiation process for updates from the stewarding DLBB. This defeats the objective of minimizing the extent of data redundancy (see Section 1.1.2). Moreover, as pointed out in Section 1.1.1, installing shared redundant copies close to the building blocks needing it should alleviate the need for private redundant copies.
- Each shared redundant copy updating other copies must keep track of where updates were successful and provide backup to copies which have failed, a functionality already supported in the stewarding DLBB, i.e., we have a duplication of functionality.

Hence, the DLBB holding the shared redundant copy acts as a passive source for that data. In other words, this DLBB is not responsible for actively propagating updates (that it obtains from the stewarding DLBB) to other building blocks who may have obtained private redundant copies of data from it. It is the responsibility of the other building blocks desiring the data to obtain it on their own by explicit requests, if they had obtained private redundant copies from this DLBB.

This rule minimizes the progressively lagging consistency that could result as updates are passed from one redundant copy to another, by requiring automatic updates to be provided by the stewarding DLBB only. Since the number of redundant copies should be limited and closely managed, the stewarding DLBB is the primary data from which all other copies are directly related.

Rule 5 - Updates to a shared redundant copy are made only by the stewarding DLBB.

This rule is a corollary of rules 2 and 4. Rule 4 indicates that a shared

redundant copy cannot get updates from other shared redundant copies or private redundant copies. Rule 2 indicates that only updates made to the stewarded data are valid updates. Hence, all updates must be received directly from the stewarding DLBB. The stewarding DLBB has to commit the update and provide the update to the shared redundant copies depending on their consistency requirements. An update can be committed by the stewarding DLBB together with a shared redundant copy in one distributed transaction (for example, using two-phase commit protocols), or the update may be committed by the stewarding DLBB first and then propagated to the shared redundant copies. In either case, the stewarding DLBB should handle the concurrency control across multiple updates, ensure that the semantic integrity constraints hold and make the update.

> ***Rule 6 - A private redundant copy requiring automatic updates must be obtained from the stewarding DLBB; otherwise a private redundant copy may be downloaded from a shared redundant copy.***

If there is a need for receiving updates automatically, a private redundant copy must be obtained from the stewarding DLBB and the required consistency or view requirements must be established with the stewarding DLBB. If a shared redundant copy is able to satisfy the consistency or view needs of a private redundant copy, the private redundant copy may be obtained from the shared redundant copy. However, if further updates are necessary on the private redundant copy, then these are refreshed from the shared redundant copy by the building block which owns the private redundant copy by explicitly making requests for a more recent copy.

> ***Rule 7 - The building block having a redundant copy is responsible for its copy.***

A building block containing a private redundant copy knows best the usage of and the needs for that copy. The building block may provide alternate views of this data, or merge this data with other data, etc., depending on user requirements. Hence it is best left to the building block to manage its private redundant copy. In other words, the building block can exercise local autonomy (described in Section 2) as far as the private redundant copy is concerned. The building block may also establish a consistency requirement with the stewarding DLBB to automatically update the private redundant copy.

A DLBB housing a shared redundant copy is responsible to provide the appropriate views expected of that copy. However, a DLBB containing a shared redundant copy cannot exercise local autonomy entirely as the updates on the data are made only from the stewarding DLBB as explained in Rule 5. The number of intra-DLBB integrity constraints on a partial replicate shared redundant copy may differ from that of the stewarding DLBB. If this is the case, appropriate interpretation of the updates sent from the stewarding DLBB may be the responsibility of the DLBB with the shared redundant copy if the stewarding DLBB does not perform that interpretation.

4. *Schemes for Managing Redundant Data*

This section identifies the required contracts for data redundancy management, describes transaction processing models for redundancy management, illustrates two management schemes and an algorithm that can be used for distinct consistency requirements on redundant data and then indicates solutions that can be employed to satisfy different application requirements.

4.1. Contracts for Redundancy Management

It was mentioned in Section 1.1.2 that a stewarding DLBB has to manage redundant copies of its stewarded data. A stewarding DLBB may be required to invoke a redundancy management contract each time changes are processed against the stewarded data, such as when certain attributes or entities have been changed. This is the simplest case. A stewarding DLBB may also invoke a redundancy management contract according to some fixed frequency schedule. This is a more expensive process, requiring that the DLBB be invoked via timer services at certain intervals to examine whether it should send updates to registered building blocks. The stewarding DLBB will also have to keep track of the date/time of an update to send updates after a certain time. An on-demand redundancy management contract is a retrieval request by the building block owning the redundant copy and does not require any additional functionality.

The processing of redundancy requests can occur in the on-line mode or in the batch mode. Typically, lock-step consistency requires the on-line mode. Eventual consistency and lagging consistency can be supported by either mode.

To facilitate automatic updates from the stewarding DLBB, a building block having a redundant copy must provide appropriate contractual support on its redundant copy. If a building block supports a private redundant copy and requires automatic updates, then pre-defined create/update/delete (CUD) contracts have to be offered on the private redundant copy for the purposes of updating the copy from the stewarding DLBB. These CUD contracts must *only* be accessible by the stewarding DLBB.

If a DLBB supports a shared redundant copy of data, then it offers retrieval contracts on that data. In addition, pre-defined CUD contracts have to be offered on the shared redundant copy if automatic updates are desired from the stewarding DLBB. Again, these CUD contracts must *only* be accessible by the stewarding DLBB.

4.2. Transaction Models for Redundancy Management

Transaction processing is an important factor in managing redundant data and providing the required degree of consistency in the redundant copies. The transaction management characteristics of the interaction paradigm among the building blocks has a great deal of bearing on the degree of consistency that can be achieved in a redundant copy.

A transaction is a collection of actions which has the so-called ACID properties:

- Atomicity: Either all of the actions are performed, or (effectively) none of them are.
- Consistency: A transaction takes the database from one consistent state to another consistent state.
- Isolation: The actions of a transaction are performed in effective isolation from the actions of other transactions.
- Durability: Once a transaction has been completed and committed, the effects survive any combination of system failures.

The usual criterion for isolation is serializability. This means that when multiple transactions are executed concurrently, the results should be the same as if they had executed serially in some order. The mechanisms for ensuring serializability are usually called *concurrency control* protocols. One of the most common of these is *two-phase locking,* in which each transaction goes through two phases with regard to locking data items accessed by it. During the first phase the transaction acquires locks as needed to access data items. During the second phase the transaction releases locks. Once any lock has been released, no further locks may be acquired. This guarantees serializability.[14]

The most well known protocol for ensuring atomicity and durability in a distributed system is the *two-phase commit* protocol. One of the participants in the distributed transaction plays the special role of the "coordinator." In the first phase each participant reports to the coordinator whether it is prepared to commit or not. In the second phase the coordinator directs them all to commit or all to abort, depending on whether or not ***all*** reported that they were prepared. When a participant reports that it is prepared, it must remain prepared through any combination of failures. If the coordinator directs it to commit, it ***must*** be able to commit.

In distributed systems, especially heterogeneous systems, it can be very difficult to achieve serializability and atomicity in an efficient manner, and redundant copies may not require them in all cases. Therefore, sometimes the transaction properties can be relaxed depending on the consistency requirements of redundant copies.

Four possible client-server interaction paradigms are of primary interest to us here:

- Queued Message (QM)
- Independent Invocation (I-I)
- Dialog with Distributed Transaction Processing (DTP)
- Dialog with Local Transactions (DLT)

The first two are *single request-response* paradigms. An interaction consists of a single request from the client optionally followed by a single response from the server. No state information is retained between successive requests; each request is handled in isolation from preceding or succeeding requests. The last two are *dialog* paradigms. An interaction consists of a series of interaction steps, each consisting

of a request and response. State information is retained throughout the dialog, so that information developed in one step can be referred to in succeeding steps. Typically a communications "session" or "application association" is established to serve as the context for the dialog.

In the QM paradigm a request is sent from a client building block to a server in a store-and-forward manner requesting the execution of an operation by the server. The incoming message to the server, the processing at the server, and the outgoing response from the server are all covered by a common transaction umbrella, under the control of the server. In the event of a system crash at the server in the midst of processing the request, the recovery process not only rolls back any partial processing, but also restores the incoming message on the input message queue so that it is ready to be processed again. In the event of a system crash at the client or the server after the request has been processed and committed, but before the response has been delivered to the client, the recovery process leaves the outgoing response still on the output message queue waiting to be delivered. Once the client has sent the request message, it will eventually get processed and the response returned even in the event of multiple crashes at the server and/or client. The transfer of messages from the output queue of either the client or the server building block to the input queue of the other building block is also assured through a handshaking protocol. Thus, the QM paradigm is resilient to communications crashes as well as client and server crashes.

The QM paradigm can be used for satisfying eventual or lagging consistency requirements on a redundant copy. The characteristics of the QM paradigm ensure that the redundant copy will certainly be updated eventually even if multiple crashes occur at the building block with the redundant copy. This implies that once the redundancy message is sent from the stewarding DLBB, the stewarding DLBB is assured that the update will be made in the redundant copy.

In the I-I paradigm a request is sent from a client building block to a server requesting the execution of an operation by the server. The operation may be executed as a transaction, but transaction management is completely under the control of the server. If the server crashes after the message is received but before the operation is executed and the response sent, it is up to the client to discover that fact and take appropriate recovery action. The server will typically have no memory after recovery that the message was ever received. Typically the I-I

paradigm is synchronous; i.e. the client process blocks while waiting for a response. However, that is not necessarily the case. It depends entirely on the client.

The I-I paradigm can also be used for satisfying eventual or lagging consistency requirements on a redundant copy. However, since an update of the redundant copy is not always assured, the stewarding DLBB may have to resend the redundancy message if a response is not received within a time window from the building block having the redundant copy. It is important to use idempotent[7] operations in this case as otherwise undesirable data inconsistencies may result.

In the DTP paradigm sequences of operations at the server are executed as a subtransaction of a distributed transaction. Whenever a subtransaction is to be committed, the server must enter into a distributed commit protocol (e.g. two-phase commit) with the client and/or any other building blocks which are part of the distributed transaction. Typically the client will initiate the commit process; that is, the client will indicate to the server the points at which work should be committed. Thus the interaction consists of a sequence of operations followed by a distributed commit, then another sequence of operations followed by a distributed commit, etc.

The DTP paradigm can be used for lock-step synchronization of redundant copies. Assuming that we have a valid update at the stewarding DLBB, two situations can arise here. First, the building block with a redundant copy accepts to commit an update. There is no problem in this situation and the question of reissuing the redundancy message does not arise. Second, the building block with a redundant copy crashes before or while processing the redundancy message, or does not correctly participate in the distributed commit protocols because of erroneous behavior. In this case the stewarding DLBB may commit the update independent of the redundant copy and resynchronize the redundant copy (with redundancy messages) upon recovery, or reject the update with appropriate messages to the invoker requesting a retry later. Although the former approach should be the more common approach, the latter approach may be taken depending on the

7. An idempotent operation is an operation that can be applied on some data any number of times, but still produces the same result. For example, updating the value of an attribute to a specific value is an idempotent operation whereas incrementing the value of an attribute by 10 is not idempotent because it produces a different result each time it is applied.

importance of the role played by the redundant copy. For example, if the redundant copy is being used for implementing some crucial inter-DLBB semantic integrity constraints and if other inter-DLBB semantic integrity constraints may also be compromised if the redundant copy is not synchronously updated, then the business decision could be to reject the incoming update. The DTP paradigm is supported by the ISO TP[16] draft standard and the X/Open DTP[17] standard.

In the DLT paradigm the operations at the server are executed as a stand-alone transaction, not a part of a distributed transaction. Typically the client will indicate to the server when work should be committed, and the server will then simply attempt to commit its own work without any further coordination with the client or any other building block. Thus the interaction consists of a sequence of operations followed by a commit at the server, then another sequence of operations followed by a commit at the server, etc.

The DLT paradigm can also be used for satisfying eventual or lagging consistency requirements on a redundant copy. However, since an update of the redundant copy is not always assured, the stewarding DLBB may have to resend the redundancy message if a response is not received within a time window from the building block having the redundant copy. Again, it is important to use idempotent operations in this case as otherwise undesirable data inconsistencies may result.

4.3. Management Schemes for Redundant Data

This section describes a scheme and an algorithm for providing eventual or lagging consistency in redundant data, and another scheme for lock-step updating of redundant data. A combination of these schemes can be used by a stewarding DLBB to manage the consistency requirements of the redundant copies.

4.3.1 The Linked Contracts Table Scheme

This scheme is useful to maintain eventual or lagging consistency in redundant copies. The QM, I-I or DLT transaction paradigms described in Section 4.2 can all be used within this scheme.

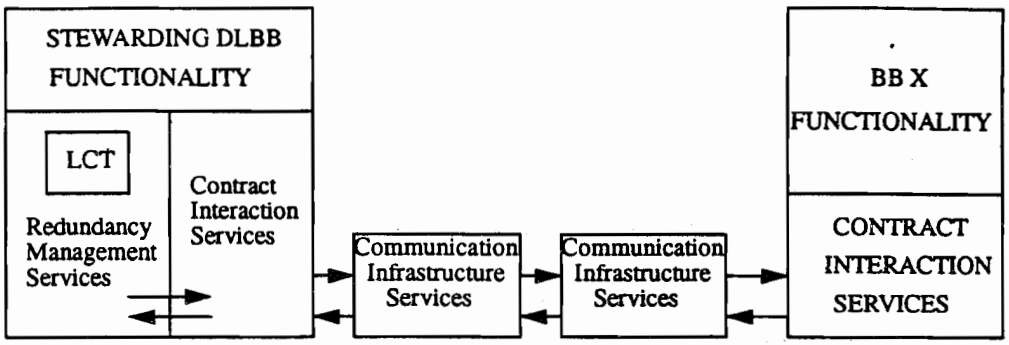

Figure 1: Redundant Data Management Using Linked Contracts Table.

Figure 1 illustrates the scheme for a building block (BB) X having a redundant copy of the stewarded data. The Redundancy Management Services (RMS) and the Contract Interaction Services (CIS) may be part of the infrastructure services installed with the stewarding DLBB. The RMS infrastructure functionality may initially be developed as part of the stewarding DLBB functionality, and later replaced by an actual vendor product or by appropriate DBMS functionality.

Each redundant copy requiring updates is responsible for establishing the update frequency with the RMS. The update frequency, the contracts to be invoked for updating redundant data and possibly alternate data views of the redundant copies are maintained in the linked contracts table (LCT). The LCT may be part of the RMS or may be part of a data directory which is accessed by a RMS. In either case, provisions must be present for adding, deleting or updating the contents of the LCT.

The RMS uses the view information in the LCT for a redundant copy (typically for a shared redundant copy than a private redundant copy) to filter and provide meaningful updates to the redundant copy. If the view information for a redundant copy is not maintained in the LCT, then the building block having the redundant copy will be responsible for translating and interpreting the updates sent from the RMS.

In this scheme when an update contract is invoked, the update contract provider in the stewarding DLBB commits the update if all the semantic integrity rules are satisfied. The update is then provided

to the RMS. The RMS consults the LCT to determine which redundant copies should get the update and when, performs appropriate translations for views stored in the LCT, and provides CIS with the information in the LCT to invoke contracts in building blocks containing redundant copies of the stewarded data. The RMS must ensure that each redundant copy is able to perform the update in the same order as the stewarding DLBB and repeats a redundancy message when necessary.

Since the stewarding DLBB makes the updates prior to sending update contracts to the redundant copies, this scheme can be used for maintaining eventual or lagging consistency in redundant copies. Updates can take the form of *full extraction and load* or *incremental extraction and update*. The process of full extraction and load is straightforward, but involves large amounts of data and updating redundant copies is time consuming. On the other hand, incremental extraction and load requires a careful audit trail of the updates to be maintained by RMS for each redundant copy, but updating redundant copies is less time consuming. The latter approach can be used for online synchronization to meet stricter consistency requirements while the full extraction and load process is suitable for batch synchronization. We present a practical algorithm in the next subsection that allows for incremental updating of redundant copies that can be used for providing eventual or lagging consistency in redundant copies.

4.3.1.1 The Epoch Propagation Algorithm

In this algorithm the RMS may propagate a batch of updates, called an epoch. The epochs provide a window of consistency for the redundant copies. The length of an epoch corresponds to the number of update requests (each of which may create, delete or modify a number of entity instances) present in the epoch. Choosing a long epoch would mean that the redundant copies would be less consistent with the steward and that may not satisfy many application requirements. On the other hand, choosing an epoch length of one update request is equivalent to the situation described above in the IPA algorithm.

There are several approaches that can be used to choose the epoch length. We describe three possible approaches in this section. The first and simplest approach, called the *fixed epoch length* approach, is to choose the epoch length to be a fixed number of incoming update

requests (for example, 10 incoming update requests) to the RMS such that the epoch satisfies the consistency requirements of all copies. Here the RMS uses the same set of incoming update requests to filter and then propagates non-empty epochs to redundant copies appended with a serial number. If all the update requests are filtered out from an epoch, then RMS ignores them and assigns serial numbers in a sequential order only to the non-empty epochs sent to a redundant copy. Since the filtering condition varies from copy to copy, RMS will have to keep track of the last serial number sent to each copy and also correlate the serial numbers assigned for a particular epoch across the copies in order to determine if all the responses have been received for that epoch from all the copies.

The fixed epoch length approach will result in filtered epochs with varying number of update requests in them from one epoch to the next. Receiving varying number of update requests may be a problem for some copies because that may prevent them from optimizing their performance for other accessing building blocks. However, the advantage of this approach is that the RMS can make a simple correlation between the responses received for the epochs with the common epoch formed in the RMS for all copies, thus simplifying the bookkeeping in the RMS and the logic needed to report back to the stewarding DLBB or the system administrator of a successful propagation of the update requests. It may also be possible to choose an epoch length such that it reduces the overhead in the network and the RMS, and at the same time satisfies the consistency requirements for all copies.[8]

A second and equally simple approach, called the *fixed time duration epoch* approach, is to choose the epoch to be the incoming update requests received during a certain fixed time duration. Here the stewarding DLBB sends the update requests to the RMS as and when they are processed. The RMS saves these updates in an update log, forms epochs by writing delimiters periodically, filters the epochs and sends non-empty epochs to redundant copies appended with a serial number. If all the update requests are filtered out from an epoch, then RMS ignores them and assigns serial numbers in a sequential order only to the non-empty epochs sent to a redundant copy. Since the filtering

8. Polyzois and Garcia-Molina report results of their test findings on epoch length.[18] But the epoch length should be carefully chosen after taking into account the parameters of a given network and processing environment.

condition varies from copy to copy, RMS will have to keep track of the last serial number sent to each copy and also correlate the serial numbers assigned for a particular epoch across the copies in order to determine if all the responses have been received for that epoch from all the copies.

The fixed time duration epoch approach will result in varying epoch lengths from one epoch to the next because the rate of update contract invocations in the steward will vary from time to time and so will the incoming update requests to the RMS. In addition the filtered epochs will also be varying in length and may not be suitable for BBs trying to optimize their performance for other accessing building blocks. The advantage of this approach, like the fixed epoch length approach, is that it simplifies the bookkeeping in the RMS and the logic needed to report back to the stewarding DLBB or the system administrator of a successful propagation of the update requests. Extensive knowledge of the rates of update contract invocations in the stewarding DLBB is necessary in order to choose an appropriate time duration for the epoch that satisfies the consistency requirements for all copies.

The third approach, called the *copy-based epoch* approach, provides filtered epochs to each redundant copy according to a prespecified propagation condition for that copy. The propagation condition can be a complex boolean condition formed based on the number of update requests in the filtered epoch, time duration since the last propagation, time of day, etc. For example, a propagation condition may be to send filtered epochs that have accumulated at least 10 update requests during the hours of 8 am and 5 pm, and to send filtered epochs every half hour outside these hours. The RMS has to continually process the update requests as they arrive to check if the propagation condition for a redundant copy is met and send the filtered epochs to that copy if that is the case. The Epoch Propagation Algorithm (EPA) in Table 1 describes the copy-based approach. The epochs for the various redundant copies are sequentially numbered and maintained using pointers to a common log of incoming update requests as described in the EPA algorithm in Table 1. The RMS keeps track of the progress of update request processing in each copy by maintaining a response pointer to the latest epoch that has been acknowledged.

The copy-based epoch approach could require considerable bookkeeping in the RMS to keep track of individual epoch limits for each

Step	RMS	BB with a redundant copy
1	Receive update requests from the steward and save the update requests in a common log.	
2	If the propagation condition is met for a redundant copy, store a pointer to the common log to indicate a new epoch beginning for that copy and the next sequential epoch #. Filter accumulated updates in the previous epoch and issue them to the copy after appending the corresponding epoch #. Check for timeout or responses.	Whenever a filtered epoch with an epoch # is received: If this epoch # is already processed, go to step 4. If there are no missing epoch #s, then process the epoch, save the epoch # and go to step 4. If missing epoch #s, save current epoch and request the RMS for missing epochs.
3	If copy indicates it is missing epochs, then send missing epochs. If timeout occurs before response and if resend limit for that epoch has not been reached for a copy, then resend epoch. If resend limit is reached, then inform system administrator.	Process all the missing epochs and saved epochs in their epoch # order. If any problems in processing an epoch, then request RMS to resend that epoch and process again.
4	If successful response is received from a copy before timeout, then note the epoch #(s) in the response and advance response received pointer for that copy to the most recent epoch acknowledged.	Respond back to the RMS about the update success for the received epoch(s).
5	Inform the steward or the system administrator that a particular update was successful after receiving positive response for that update from all the copies that were issued the update.	

Table 1: Epoch Propagation Algorithm (EPA).

copy and logic to check if the propagation condition is met for each copy. The previous two approaches allow update requests in entire epochs to be reported back to the stewarding DLBB to have been successfully propagated. Since epoch boundaries and their intersection across copies could considerably vary in the copy-based approach, the RMS has to correlate responses for each update request (instead of a whole epoch) and report back to the stewarding DLBB or the system administrator of the successful propagation of that update request. However, the advantage of this approach is that it is versatile and caters to the tailored needs of each redundant copy.

Irrespective of the approach used, the filtered epochs that are received are applied by the copies in the order of their epoch number. If received epochs could not be entirely applied, then redundant copies may request the RMS to resend an epoch. In such a situation, it will be up to the redundant copy to either commit the epoch to the extent it was able to process, or commit none at all. The received epochs are acknowledged by the redundant copies after successful application of the update requests in these epochs. Responses for multiple received epochs can be sent together in the same message to the RMS as long as the response is sent within a pre-specified time window.

The RMS resends filtered epochs if responses are not received from a redundant copy. Such resends are more expensive than in the IPA algorithm because of the amount of time required in the RMS to filter each lengthy epoch. In addition the lengthy epochs can also cause overhead in network traffic. But a pre-defined number of resends are necessary to unblock redundant copies that are blocked due to failures, and to inform the system administrator in case of repeated errors. The performance of the EPA algorithm suffers only if there are many epoch resends that counter the advantage of delaying the updates. However, this is also an useful technique and a good alternative when lock-step updates are not possible, especially for maintaining semantic integrity constraints. The queued message (QM) paradigm, the independent invocation (I-I) paradigm, and the dialog with local transactions (DLT) paradigm can be used to implement this algorithm. A version of the EPA algorithm has been successfully implemented using the QM paradigm and has been demonstrated to cater for both eventual and lagging consistency requirements of redundant copies across heterogeneous environments.

4.3.2 The Synchronous Update Scheme

This scheme is useful for lock-step updating of redundant copies. The DTP paradigm described in Section 4.2 must be used here.

If lock-step or synchronous updating of redundant data is desired, then the update contractor of the stewarding DLBB cannot make updates in isolation from the RMS. The information about redundant copies requiring lock-step updates may be kept in a resident table or as part of a data directory accessed by the RMS. Again, this information includes the needed contracts to update the redundant copies and possibly information about the view supported by the redundant copy. Provision must be there to add, delete or change entries in this table.

In this scheme when an update contract is invoked, the update contractor processes the update, ensures all the semantic integrity constraints can be satisfied, but provides the RMS with the result of the update prior to committing the result. The RMS consults the resident table to determine which redundant copies need lock-step updates, performs appropriate translations for views stored in the table for these copies, and provides contractual information to CIS. CIS then uses the transaction monitor facilities to invoke a distributed transaction across those building blocks containing lock-step synchronous redundant copies of the stewarded data. The scheme is shown in Figure 2 for a building block (BB) X having a lock-step synchronous redundant copy of the stewarded data. The transaction monitor facilities may utilize a two-phase commit protocol to implement the distributed transaction. A discussion on distributed transactions using the X/Open DTP model appears in Mills.[15]

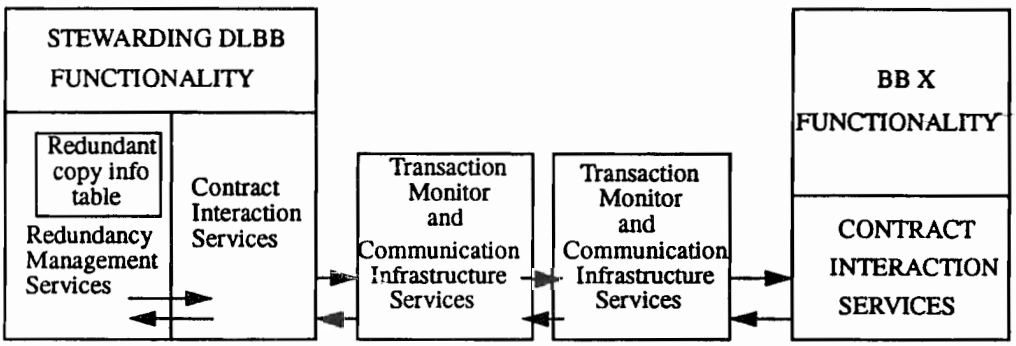

Figure 2: Lock-step Updating of Redundant Copies.

The transaction monitor is responsible for the coordination of global commit/abort operations. If a two-phase commit protocol is used, then all participating building blocks that are prepared to commit must commit. Hence, a participating building block must store the results in permanent storage prior to issuing a "ready to commit" message. This allows the results to be committed upon recovery if the building block were to fail prior to the actual commit.

4.4 Satisfying Varying Requirements on Data

In this section we present possible approaches that can be taken to satisfy the system requirements (mentioned in Section 2) for different cases.

Stewarded data can satisfy needs for highly accurate data. However, if the stewarding DLBB cannot satisfy very high performance requirements, then possible solutions are:

- to support the stewarding DLBB on a very high performance platform,
- to horizontally fragment the stewarded data such that the load on each fragment can be managed,
- to off-load the stewarding DLBB for retrieval operations (especially for ad hoc queries) with a closely consistent (see Section 4.3.1) shared redundant copy, or
- to offload the stewarding DLBB for retrieval operations with a lock-step synchronized shared redundant copy (see Section 4.3.2) if consistency of retrieved data should be 100%.

The solution used varies from case to case. A combination of the above solutions may be used if appropriate. In general, using private redundant copies should be considered as the last option only if no better alternative exists for a given situation.

If lock-step synchronized updates on redundant data satisfy the accuracy requirements, but lock out access to much needed access to data for long durations of time, then possible solutions are to provide a fine level of locking granularity or to use optimistic algorithms on top of distributed transaction protocols to delay locking an item of data for the longest duration possible.

Low or medium accuracy requirements on redundant data can be met by using a delayed updating scheme for redundant data (see Section 4.3.1) or by using batch update or batch retrieval contracts (see Section 4.1).

Very high requirements on the availability and reliability of data can be met by supporting stewarded data on fault-tolerant systems, and maintaining mirrored disks and frequent back-ups of the data. High availability requirements on data can also be met by installing shared redundant copies close to the systems needing access to the data (see advantages of shared redundant data in Section 1.1.1). Medium or low requirements on availability and reliability of data can be easily met by periodic backing up of stewarded data.

5. *Conclusions*

We have described an interoperability architecture called the OSCA architecture and the approach taken for managing redundant data in the context of this architecture. The OSCA architecture favors supporting the corporate data resource of large corporations in diverse computing and data environments. The data redundancy management rules, management schemes and an algorithm presented in the context of this architecture offer a practical solution for managing redundancy to suit varying needs of applications in autonomous heterogeneous environments.

Acknowledgments

The author acknowledges John Mills, Fen Kung, Aloysius Cornelio, Gomer Thomas, Deb Mukhopadhyay, and many others for their excellent comments.

References

[1] ISO/IEC CD 10746-2, *Basic reference Model of Open Distributed processing-Part 2: Descriptive Model,* July 1991.

[2] *The Bellcore OSCA™ Architecture,* Bellcore Technical Reference TR-STS-000915, Issue 1, October 1992.

[3] H.N. Srinidhi, "Guidelines for Decomposing Information Models into Data Layer Building Blocks," USING™ 92 Conference Proceedings, May 1992, pp. 211–227.

[4] D. Steedman, *ASN.1 - The Tutorial & Reference,* Technology Appraisals Ltd., 1990.

[5] J. A. Mills; "Semantic Integrity of the Totality of Corporate Data;" *Proc. of the First International Conference on Systems Integration,* April 1990.

[6] J. A. Mills and L. Ruston; "The OSCA Architecture: Enabling independent product software maintenance," *Proc. of EUROMICRO '90 Workshop on Real Time,* June 1990.

[7] J. A. Mills; "Interoperability of Network Data Functionality with Operations Systems Data Functionality," *Proc. of TELECOM 91,* Oct. 1991.

[8] *Information Modeling Concepts and Guidelines,* Bellcore Special Report SR-OPT-001826, Issue 1, January 1991.

[9] *A Reference Model for Object Data Management,* Final Revision, ANSI X3 Document Number OODB 89-01R8, August 10, 1991.

[10] *Modeling Principles for Managed Objects,* OSI Network Management Report TR-102, January 1991.

[11] A. Sheth and M. Rusinkiewicz, "Management of Interdependent Data: Specifying Dependency and Consistency Requirements," Proc. of the Workshop of Replicated Data, Houston, Texas, November 1990.

[12] W. Du and A. K. Elmagarmid, "Quasi Serializability: a Correctness Criterion for Global Concurrency Control in InterBase," Proc. of Intl. Conf. on Very Large Data Bases, Amsterdam, August 1989.

[13] *Management of Redundant Data in OSCA™ Environments,* Bellcore Special Report SR-STS-002310, Issue 1, July 1992.

[14] K. P. Eswaran, J. N. Gray, R. A. Lorie and I. L. Traiger, "The Notions of Consistency and Predicate Locks in a Relational Database System," Communications of the ACM, Vol. 8, No. 11, Aug. 1976, pp. 624-633.

[15] J. A. Mills, "Large Scale Interoperability, Distributed Transaction Processing, and Open Systems," USING™ 92 Conference Proceedings, May 1992, pp. 199–209.

[16] *Information Processing Systems-Open Systems Interconnection-Distributed Transaction processing - Part 1: OSI TP Model,* ISO/IEC 10026-1:1991 (E), Interim final text.

[17] *Distributed Transaction Processing Reference Model,* X/Open Guide, October 1991, X/Open Company Limited.

[18] C. A. Polyzois and H. Garcia-Molina, "Evaluation of Remote Backup Algorithms for Transaction Processing Systems," Proc. of the ACM SIGMOD Int. Conf. on Management of Data, June 1992.