DATA CACHING IN AN INFORMATION RETRIEVAL SYSTEM

Hector Garcia-Molina
Rafael Alonso
Daniel Barbara
Soraya Abad

CS-TR-065-86

December 1986

DATA CACHING IN AN INFORMATION RETRIEVAL SYSTEM

Hector Garcia-Molina
Rafael Alonso

Department of Computer Science
Princeton University
Princeton, NJ 08544

Daniel Barbara
Soraya Abad

Departamento de Matematicas y Ciencias de la Computacion
Universidad Simon Bolivar
Caracas, Venezuela

ABSTRACT

Currently existing computer communication networks give users access to an ever growing number of information retrieval systems. Some of those services are provided by commercial enterprises (examples are Dow Jones [Dunn1984] and The Source [Edelhart1983]), while others are research efforts (such as the Boston Community Information System [Gifford1985]). In many cases these systems are accessed from personal or medium size computers which usually have available sizable amounts of local storage. To improve the response time of user queries, it becomes desirable to cache data at the user's site. However, to reduce the overhead of maintaining multiple copies, it may be appropriate to allow copies to diverge in a controlled fashion. This makes it possible to propagate updates to the copies efficiently, e.g., when the system is lightly loaded, when communication tariffs are lower, or by batching together updates. It also makes it possible to access the copies even when the communication lines or the central site are down. In this paper we present the notion of *quasi-copies* which embodies the ideas sketched above. We also define the types of deviations that seem useful, and discuss the available implementation strategies.

Index Terms: Distributed data management, distributed systems, information retrieval systems, caching, cache coherency, data sharing, data replication.

DATA CACHING IN AN INFORMATION RETRIEVAL SYSTEM

Hector Garcia-Molina

Rafael Alonso

Department of Computer Science

Princeton University

Princeton, NJ 08544

Daniel Barbara

Soraya Abad

Departamento de Matematicas y Ciencias de la Computacion

Universidad Simon Bolivar

Caracas, Venezuela

1. INTRODUCTION

In many of today's information retrieval systems (IRS's) all the stored data (e.g., the abstracts of journal articles, the airline schedules) resides at a central node. This central site can be reached by a large number of remote terminals connected via relatively slow communication lines. Users at these terminals do no local processing; they simply send their queries to the central machine and wait for their replies. Data can be added or deleted at the central site, but in many cases it cannot be updated.

A number of developments are slowly changing this IRS model. First, the number of users is growing rapidly. In our ''information society'' it is becoming increasingly

important to have access to timely information. At the same time, the number of personal computers, at home and in the workplace, has grown tremendously, giving more people the hardware necessary to access the IRS's. Cable TV networks and satellite are providing the capability to at least receive large amounts of information.

Another development is that the distinction between "classical" IRS's, with low update activity, and databases (or transaction processing systems), with higher update rates, is blurring. Users connected to IRS now want to have access to rapidly changing data like stock prices and news wire stories. At the same time, users would like to submit update requests based on the information they are accessing. For example, users may want to reserve airline flights, buy theater tickets, pay their utility bills, or purchase stock [Russell1986]. It is only natural that the IRS, which already handles the airline schedules, the theater schedule, and the banking information, should be able to give users these additional services.

The increased IRS services and requirements will tax the both the processing and communication capacity of the central site. There are a number of potential solutions to this problem, but the one we will focus on in this paper is *data caching*. This solution is becoming feasible precisely because the IRS is frequently accessed from personal or mini computers with substantial processing and storage capacity (for example, in 1984, Dow Jones estimated that about 125,000 of its 165,000 customers used personal computers [Dunn1984]). Caching can improve system performance in two ways. First, it can eliminate multiple requests for the same data. For example, consider an automobile manufacturing plant where a number of people are interested in news wire stories on trade and protectionism. In this case, it makes sense to cache the relevant articles at the company's local computer, eliminating redundant requests to the central IRS site. A second way in which caching can improve performance is by off-loading work to the remote sites. For

instance, if a user is interested in chemical companies he may store the latest stock prices of those companies at his own computer. There he can run his own analysis programs on the data, without using any more central cycles.

In principle, caching can off load work from the central site and reduce the communication traffic. However, caching has an associated cost. Every time a cached value is updated at the central site, the new value must be propagated to the copies. Furthermore, the propagation must be done immediately if cache consistency (or *coherency*) is to be preserved. (A cached value for an object is consistent if it equals the value of the object at the central site.) This propagation cost can be significant, especially in light of our earlier comments regarding the increasing update activity for IRS's.

Caching has been successfully used in other environments, but there are some important differences. In a computer hardware cache [Smith1982], it is not expensive to keep the cached and main memory data consistent. This is because updates are small (e.g., a byte is modified) the communication delays are short, and the number of copies is small (e.g, in snooping cache architectures typically there are less than 10 caches connected to a memory system). In a network file system [Walsh1985] local copies of the file pages being accessed are stored in local workstations. Here the updates typically occur at the workstation and are then moved to the file server. The transfer units are pages and the communication network is usually a high bandwidth local area one. Caching works fine in this case, as long as the number of client workstations to a file server is relatively small (e.g., less than 10).

In an IRS, on the other hand, the communication costs can be much higher. For instance, users typically communicate over telephone lines. Also, the number of caches may be quite large. Finally, the updates can be large (e.g., the abstract of an article or the article itself can be added to a file).

In light of these difficulties, it is important to explore strategies for making update propagation less costly while still retaining the inherent advantages of caching. In this paper we study two such strategies. Both involve taking advantage of the application semantics. The first idea is to let the user explicitly define the information that is of interest and to only cache it. This obviously reduces the need to refresh data that is not going to be used.

The second idea is to allow, whenever possible, a weaker type of consistency between the central data and its copies. For instance, the user interested in the stock prices of chemical companies may be satisfied if the prices at his computer are within five percent of the true prices. This makes it unnecessary to update the cached copy every single time a change occurs. When the deviation exceeds five percent, then a single update can bring the cached copy up-to-date. At the manufacturing company, users may tolerate a delay of one day in receiving the articles of interest. If the system takes advantage of this, it can transmit all the articles during the night when communication tariffs are lower. If a communication or central node failure occurs and its duration is less than 24 hours, then users can continue to access information that is correct by their standards.

We call a cached value that is allowed to deviate from the central value in a controlled way a *quasi-copy*. Quasi-copies have the potential for reducing update propagation overhead and giving the system flexibility for scheduling the propagation at convenient times. Note that the information flow in an IRS with quasi-copies is similar to the flow in many real organizations. The manager of a company is not told every time an employee is hired or leaves. The information is filtered so that he only is informed periodically of personnel changes, or if an exceptional condition occurs (e.g., a mass exodus of employees). Hence, the manager's view of the company (the cached data)

deviates from the true state (the central value). Similarly, when a person desires news, he subscribes to magazines and newspapers. The news arrives periodically and there is again a discrepancy between the local and ''central'' data. In human organizations, people have little control over this process, e.g., Time magazine arrives every week and the New York Times every day, and there is no way to change this. In a computerized IRS, however, we can let users precisely define the limits of divergence of quasi-copies, and the system can take advantage of this to improve performance.

In this paper we will assume that all information or data is controlled at a single central site. This site executes all updates and hence has the most up-to-date version of all data. Usually remote users only read data. If they do want to modify something, they submit an update transaction to the central site. If the modifications are based on data read from the IRS, the reads must occur at the central site at the time the transaction runs, not at the remote node. To illustrate, let us return to our stock price example. Suppose that the IRS also allows users to purchase stock. If a user observes at his terminal that the price of a certain stock is good, he can submit a transaction to the central site to purchase some amount. However, the ''real'' price, i.e., the price at the central site, can differ from the value observed by the user when he made his decision. Hence, the user must either be willing to buy stock at a ''slightly'' different price or must include in his update transaction code to read the price once again and abort the operation if the price is no longer acceptable. (Note that this is the way stocks are usually purchased in reality.) This decision is very similar to that faced by users of database browsers based on the idea of *portals* [Stonebraker1982].

Our model could easily be extended to one with several central sites, each controlling a fragment of the data, as long as modifications to a particular datum could only take place at one computer. Fragmenting the database like this is another way of reducing the

workload at the central site, but for simplicity, we will continue to assume that there is a *single* central site.

However, extending our model to allow updates to a datum to originate at multiple sites (e.g., at a user remote machine and at the central site) is *not* simple and will not be considered here. Central data control is essential to our approach since it simplifies the types of inconsistencies that can occur in a distributed system with replicated data. For a survey of distributed control strategies for replicated data see [Davidson1985].

Even though quasi-copies seem to be crucial for effective caching in an IRS, very little is known about them. Hence, the objective of this paper is to study data caching and quasi-copies and to attempt to answer some of the basic questions. What types of quasi-copies are most useful? How can they be defined? How can conventional data consistency constraints (e.g., a manager's salary must be greater than his/her employee's salary) be enforced at the cached copies when the individual values can fluctuate? Quasi-copies can be implemented in a variety of ways. For instance, values that diverge too much can be invalidated or refreshed. Data sent to the caches can include an automatic expiration time and date. The quasi-copy requirements can be enforced at the central or at the remore sites. In this paper we will survey the various implementation strategies and their tradeoffs.

It is important to emphasize that caching and quasi-copies may not be the indicated solution for all IRS's. In some cases the overhead of either propagating updates or managing quasi-copies may not justify caching; or perhaps certain information must always be the latest available (i.e., a user may be interested in the most current price of gold). Yet, before deciding if a system can benefit from caching it is important to understand what the options are, and this is what we intend to do here. Also, it should be kept in mind that the ideas we will present are quite general. In any particular system, only a

small subset of them may be implemented. But again, it is important to understand the choices before making decisions.

In the following section we define our model more precisely and introduce some terminology. There are two types of conditions that can be specified for quasi-data: selection and replication. They are discussed in Sections 3 and 4. The impact of transmission delays and failures is analyzed in Section 5, while other implementation issues are covered in Section 6.

2. THE MODEL

We start by defining more precisely our model and introducing notation that will be used in the rest of the paper. The database is stored at the central node, C, and consists of a set of objects O. Each object $x \in O$ can have a number of values (or fields) associated with it (e.g., object John has name, address, salary values), but for simplicity we assume there is just one value. As is customary, we use the same symbol x to represent both the object and its value. As an object is modified, new versions are created (and old versions discarded). We represent the latest version of object x by $v(x)$. For example, say x is a computer program in a software IRS and it has version 5. When the program is modified (e.g., to fix a bug), the new program will have version 6. Typically, version 5 will no longer be available. It will sometimes be necessary to refer to the value of an object x at a time t. We represent this by $x(t)$. (Incidentally, we assume that all sites have accurate and synchronized clocks [Lamport1978].)

A set of nodes N ($C \notin N$) may contain quasi-copies of the objects. The quasi-copy of object $x \in O$ at node $j \in N$ is x^j and is called an image of x. When the identity of node j is not important, we represent the image as x'. The set of objects that have quasi-copies at node j are the objects cached at j. Note that the caches at different nodes can have different objects, and objects can be cached at 0,1,2,... or all nodes.

Users define how quasi-copies are managed by giving two types of conditions: selection and replication. The selection conditions specify which object images will be cached at the user's site. The replication conditions define the allowable deviations between an object and its images. In our stock market example, the user issues a selection condition to indicate that he wants copies of the stock prices of chemical companies. His replication condition would then state that a five percent variation between the central and his site is acceptable. We now discuss these types of conditions in more detail in the next two sections.

3. SELECTION CONDITIONS

In a computer hardware cache, the decision as to what is hold in the cache is made automatically by the system. For example, the system might store every word that is fetched. To make room for the word, it may purge the least-recently-used (LRU) word from the cache.

In an IRS, the same types of automatic strategies could be used to make caching decisions. However, if the system does not know what deviations between the copy and the central object are allowable, then the copy must be kept up-to-date.† A better strategy may be to let the user specify what data is to be cached, and at the same time, define the allowable deviations. A selection condition lets the user do this. It consists of some or all of the following items:

(1) *Selected object(s)*. The condition can explicitly list the objects involved in the selection or can give an expression that evaluates to a set of objects. For example, if a relational language [Date1975] is used for the expression, then the condition.

SELECT NAME, PRICE

† As discussed in the Introduction, the cost of doing this can be high.

FROM STOCKS

WHERE TYPE = ''Chemical Company''

can be used. It selects the NAME and PRICE attributes (or fields) of tuples (or records) that represent chemical companies. In our terminology, each NAME or PRICE value selected is an object that must be quasi-copied. There are many other languages and models for selecting data or information [Date1975], but since they are well known, we will not cover them here.

(2) *Subscribers*. This is simply a list of nodes (subset of N) that desire a copy of the object (or that no longer want a copy if the condition is turning off the selection).

(3) *Add/Drop*. This item specifies whether the objects are being selected or de-selected. That is, if ''Drop'' is specified, then the images at the subscriber nodes are removed from the caches (if they existed).

(4) *Replication Conditions*. The given replication conditions specify how the quasi-copies are to be managed once the copy is made. The various possible types of replication conditions are discussed in the next section.

(5) *Static/Dynamic*. If the selection is static, then the objects are selected once when the condition is issued by a user. If it is dynamic, then changes in the data will continuously trigger a re-evaluation of the selection expression, and objects will be added or dropped dynamically. For example, if the expression given in item (1) above is static, no new stocks will be cached at the remote site. If it is dynamic, then every time that a new stock is added at the central site, a check will be made. If the stock is of a chemical company, then a copy will be made. When a company changes its classification from chemical to something else, its copy will be purged. Note that a dynamic selection will usually be of type ''Add'' (item 3).

(6) *Enforcement.* A selection condition can be of two types: compulsory or advisory. If it is compulsory, then the system must guarantee that the selected objects are cached as requested. (See Section 5 below for details of what this guarantee entails.) If it is advisory, then the caching is viewed exclusively as a performance enhancement. In this case, the selection condition is taken as a "hint," and may or may not be followed by the system.

The advantage of compulsory selection is that the knowledge of the selection condition can be used for query optimization. To illustrate, let us return to the STOCKS expression given earlier. Suppose that it is compulsory and that the user searches for the stock price for company "AJAX" at his computer. If the stock is not found locally, then AJAX is not a chemical company. No other action is necessary since the user is only interested in chemical companies. Similarly, a query to evaluate the average stock price for chemical companies can be executed locally. If the selection was advisory, then the search for company AJAX would have to be continued at the central site. Furthermore, queries like the average stock price one would have to be executed at the central site.

The advantage of advisory selection is that it gives the system greater flexibility. If the central site is overloaded, the caching of objects can be delayed or eliminated. Similarly, if storage space is limited at the remote site, data can be purged.

In practice, a judicious combination of compulsory and advisory selections may be best. For example, consider a legal IRS that contains summaries of court cases. The system also has an inverted list index that is used to locate summaries given a set of key words. In this case it may be advantageous to cache all objects that make up the index in a compulsory fashion, and the most relevant summaries in an advisory way. This way, queries can be processed locally yielding a list of summary identifiers. Requests would only be make to the central site to fetch summaries not found locally.

(7) *Triggering Delay*. When a dynamic selection is made, a change to the central database may cause a new object to be added to (or dropped from) the selection list. In some cases, it may be desirable to delay the addition (or deletion) of the object. For example, if in an abstracts IRS a user selects abstracts on ''compilers'' and gives a 24 hours delay, then new abstracts on the topic can be batched together and sent more efficiently. In a Stocks IRS, if a user selects stocks with a price less than 100 dollars, then a delay of one hour can eliminate repeated additions and deletions of a stock whose price is fluctuating close to 100 dollars.

When a user wants to give a triggering delay, he states the maximum allowable delay Δ. The system is then free to add (or delete) an object to the selected set any time between the time the triggering occurs and Δ seconds later.

Note that a triggering delay can be used with either compulsory or advisory selections. If the selection is compulsory, then the cached data can be used for query optimization, but the results may not include the latest information. Let us return to the examples that were used to illustrate compulsory selections. Say a user has selected stocks for chemical companies. He has specified a compulsory selection and a triggering delay of 1 hour. Say he searches for the stock price of company ''AJAX'' and it is not found in the cache. In this case it is *not* necessary to look for company AJAX at the central site, even though it might have been created there within the last hour. The user has indicated that he can tolerate a delay of up to one hour for hearing about new chemical companies. Hence, the search for company AJAX need not involve recently selected objects.

Incidently, setting the triggering delay Δ too low might make it hard to implement a compulsory selection. For instance, if sending a message from the central to a remote site takes T_D seconds, then the system cannot guarantee that the selected data will be at the remote site in less than T_D seconds. We will return to this issue in Section 5.

4. REPLICATION CONDITIONS

Once an object has been selected for replication, the replication condition(s) specify the allowable deviations of the image. The replication conditions are enforced only when an image exists.

The default condition defines the allowable values for an image, even if no other conditions are given.

Default Replication Condition: An image x' must have a value previously held by the object. That is,

$$\forall \text{times } t \geq 0 \exists t_0 \text{ such that } 0 \leq t_0 \leq t \text{ and } x'(t) = x(t_0)$$

Users may specify additional constraints. Actually, any constraint on the values of the objects and images could be defined; however, our goal here is to identify and understand the more useful ones. These constraints are:

(1) *Delay Condition.* This is similar to the selection triggering delay. It states how much time an image may lag behind its object. For object x, and allowable delay of α is given by the condition

$$\forall \text{times } t \geq 0 \exists k \text{ such that } 0 \leq k \leq \alpha \text{ and } x'(t) = x(t-k)$$

Since this defines a window of acceptable value, we use the notation $W(x) = \alpha$ to represent this condition.

(2) *Version Condition.* A user may want to specify a window of allowable values, not in terms of time, but of versions. For example, if an object represents a VLSI circuit, it may be useful to require a copy that is at most 2 versions old. We represent this condition as

$V(x) = \beta$, where x is the object and β the maximum version difference. That is , $V(x) = \beta$ is the condition

$$\forall \text{times } t \geq 0 \exists k, t_0 \text{ such that } 0 \leq k \leq \beta \text{ and } 0 \leq t_0 \leq t \text{ and } v(x(t)) = v(x(t_0)) + k \text{ and } x'(t) = x(t_0)$$

(3) *Periodic Condition.* With a periodic condition a user indicates that the image must be refreshed periodically. For instance, a user may desire the stock prices every day when the market closes. The condition $P(x) = \alpha,\beta$ states that the image of x must match the object at time α, and must be refreshed every β seconds thereafter. In other words, $P(x) = \alpha,\beta$ is the condition

$$\forall \text{times } t \geq 0 \exists n \text{ such that } n \geq 0 \text{ and } \alpha + n\beta \leq t < \alpha + (n+1)\beta \text{ and } x'(t) = x(\alpha + n\beta)$$

(4) *Arithmetic Condition.* If the value of an object is numeric, the deviations can be limited by the difference between the values of the object and it's image. That is, we may state that

$$\forall \text{times } t \geq 0 \quad / x'(t) - x(t) / < \varepsilon$$

or that

$$\forall \text{times } t \geq 0 \left/ \frac{x'(t) - x(t)}{x(t)} \, 100 \right/ < \varepsilon \, \%$$

We represent the first condition by $A(x) = \varepsilon$; the second one by $A(x) = \varepsilon \, \%$.

(5) *Probabilistic Conditions.* Some users may be willing to tolerate deviations as long as they do not occur too often. The condition $R(x) = T, \varepsilon$ states that in any time period of length T the probability that the image of x differs from x is less than of equal to ε. In

other words, the total amount of time that the image differs should be less than or equal to εT.

(6) *Compound Conditions*. Yet more conditions can be built out of the elementary ones we have listed by connecting them with logical "OR", "AND", and "NOT" operators. For example, the condition

$$W(x) = 1 \text{ hour AND } V(x) = 2$$

specifies that x' can lag one hour behind x, unless x has been modified more than two times within this hour. The condition

$$W(x) = 1 \text{ hour OR } V(x) = 2$$

means that x' can always log behind x by an hour. It can even log longer if the image is still within 2 versions of x.

(7) *Multi-object Conditions*. So far we have only discussed constraints on a single object and its image. However, there can also be constraints among objects. For example, if $x_1, x_2, \ldots x_n$ are stock prices in an IRS, and $\bar{x}$ is their average, then we have the constraint $\bar{x} = \text{average } (x_1, x_2, \ldots x_n)$. A user that reads the stock prices and their average would like to see the condition hold.

Unfortunately, if a user is reading images, then the conditions on the central objects, usually called consistency constraints, may not hold. To illustrate, consider two objects x, y and the constraint $x + y \leq 10$. Say that x's image has the replication condition $A(x) = 3$ and y has $A(y) = 1$ (see item 4 above). Initially, we have the following situation:

$$x = 7 \qquad x' = 7$$

$$y = 3 \qquad y' = 3$$

An update transaction decreases x by 2 at the central site. Since $A(x) = 3$, the image does not have to be updated:

$$x = 5 \qquad x' = 7$$

$$y = 3 \qquad y' = 3$$

Note that the multi-object constraint holds at both sites: $x + y \leq 10$ and $x' + y' \leq 10$. Next, a second update increases y by 2. Since $A(y) = 1$, this change does have to be propagated. The situation is now:

$$x = 5 \qquad x' = 7$$

$$y = 5 \qquad y' = 5$$

Although the multi-object constraint holds at the central site, it does not hold at the copy site.

We give the user that wants to read consistent copies, two choices: The first is to explicitly give the system the constraints that must be satisfied. When a constraint is violated, then the missing updates must be propagated. In our example, either the central or the remote site detects that the second update makes $x' + y' > 10$, and the first update is also sent (or requested).

A second option is to state that a group of objects $x_1,...x_n$ have constraints but not give them explicitly. This option is useful when the users cannot list all their constraints or when it is too expensive to check them. In this case the system must ensure that updates to $x_1,...,x_n$ are applied in order at the copies. That is, let T_0 be the last update transaction whose modifications were applied to one or more of $x_1',...,x_n'$. Let $T_1,...,T_m$ be other updates to one or more of $x_1,...,x_n$ that were not propagated because they did not

violate any single object replication conditions. Finally, let T_{m+1} be an update that does modify one or more of $x_1,...x_n$ and does have to be propagated. Then all the updates of $T_1,...,T_m,T_{m+1}$ must be made on the images $x_1',...x_n'$, to avoid violating any constraints at the remote site.

(8) *Time Varying Conditions.* It may also be useful to allow the parameters of the replication conditions to vary over time. For example, if a user is planning an important financial operation in 30 days, he may want his data to have a smaller window as the day of the transaction approaches. Thus, he may define $W(x) = (k - 30)$ minutes, where k is the day and x is an object of interest.

5. TRANSMISSION DELAYS AND FAILURES

Before discussing mechanisms for implementing quasi-copies, we must address two ''complications'' that may make it difficult to enforce the conditions given by a user: transmission delays and failures. To illustrate, consider the condition $A(x) = \varepsilon$ and assume that an increment of more than 2ε is about to occur at the central site. The condition indicates that the difference between x and x' should ''never'' be larger than ε, and hence the update to the image must be performed ''at the same time'' as the object is changed. Strictly speaking, this is not possible.

One way around this problem is to make the data inaccessible while it is being updated. This way the change to object and image will appear atomic, for no one will be able to observe a state where the changes have not completed. To achieve this atomic commit a 2-phase commit protocol (or a similar strategy) must be used. The first step is to make the image inaccessible (by sending a message to the remote node). The second step is to update the object and send a message to the remote node updating and releasing the image.

The problem due to failures is similar. For example, if the central site fails just after the 2ε increment is made but before the charge is propagated to x', then the condition $A(x) = \varepsilon$ will be violated. A 2-phase commit protocol again "solves" the problem. If the failure occurs when the values are inconsistent, then at least the image will be inaccessible and it will not be possible to observe the discrepancy.

The 2-phase commit solution has two obvious disadvantages: update overhead is higher and data can be inaccessible. In an IRS where we are already talking about differences between objects and images, it may be unnecessary to hide the differences caused by non-atomic updates. Furthermore, these differences may be very difficult to observe in the first place, so it may make even less sense to hide them.

For instance, in our previous failure example, suppose that x is updated at the central site (without 2-phase commit) and then this site fails. It is true that $A(x) = \varepsilon$ is violated, but since the central site is down, no one can read x to discover the discrepancy. Similarly, in the no-failure case, say that the update message to the remote site takes t seconds. Then for t seconds $A(x) = \varepsilon$ will not be true. However, to see this at the remote site, one would have to read x after it is changed, send it to the remote site in less than t seconds, and compare it to x'. If the messages from central to remote site arrive in order, then this would not be possible.

Hence, a second way around the problems of message delays and failures is to state that (a) the conditions given by the user will hold in any data read and collected within the system, and (2) the system will make its best effort to enforce the conditions. The second clause is needed to rule out a system that only propagated updates to the remote sites when it noticed that someone was trying to check the conditions. The first clause states in an indirect way that conditions need not hold while updates are being propagated and when the central site or its communication lines are down.

While the ''correctness'' criteria we have given is useful in some environments, it does have one drawback: it gives the user little feedback as to when his constraints do hold. If the user requests $A(x) = 2$ and he reads $x' = 3$ at his terminal, he does not know if x is really within 2 units of 3, or if $x = 1000$ and the system has been diligently trying for the last minute or the last year to inform his system of the change.

To resolve this ambiguity, we propose a third, intermediate solution. The central site, when operational, will make sure that each remote site receives a message at least every δ seconds. This means that if the central site notices that no message has gone out to a site in $\delta - T_D$ seconds, where T_D is the maximum message delay, then it will send a ''null'' message. Null messages are numbered just like regular messages, and all messages must be received in order. When a site j notices that δ seconds go by without a message from the central site, it declares the central site failed, and sets a local variable $C\_FAILED(j)$ to true.

Then we interpret every condition $C(x)$ on object x set by a user at node j as

$$C(x) \quad \vee \quad W(x) = \delta \quad \vee \quad C\_FAILED(j)$$

(Condition W is defined in Section 4.) This means that all conditions have an implicit delay window of δ and do not have to be enforced if the central site is down. With this approach, the user can display C_FAILED at the same time he displays his data, and interpret it accordingly. In our running example, if C_FAILED is false, he will know that at most δ seconds ago, x was within 2 units of the value 3.

In summary, there are several ways for coping with transmission delays and failures: using 2-phase commit, adding implicit delay and failure constraints, and simply tolerating ''system'' discrepancies. The choice of strategy will depend on the performance required and the flexibility of the users.

6. IMPLEMENTATION

The selection and replication conditions we have presented give the system flexibility in propagating updates to the quasi-copies. This has the potential of reducing communication traffic and improving performance. However, this also gives the system more responsibilities, for it now must decide when and how to propagate the updates. In particular, the burden of checking the replication conditions may outweight the benefits, so the system or its designer must be smart enough to only utilize quasi-copies (as opposed to exact copies or no copies at all) only in those cases where there can be a benefit.

In this section we outline the mechanisms that can be used to implement quasi-copies. At the same time, we discuss the situations where they can be most appropriate. We start by listing the base parameters that can greatly affect the selection of a strategy. Then we present what we consider the three major decisions that must be made in implementing quasi-copies. Each of these decisions leads to a particular type of mechanism. Finally, we discuss other secondary choices that must be made.

The efficiency and best strategy for quasi-copies will usually be determined by the base parameters of the system; processor power, storage capacity, communication topology and bandwidth, update profile, and query profile. The processing power of the central and remote nodes clearly plays an important role since condition checking and caching consume CPU cycles. Storage capacity at the remote nodes is essential for caching. Communication costs also play a key role. In particular, the broadcast capabilities of the central site are of central importance. For example, if broadcasts are cheap (e.g., if the central site communicates with users via radio [Gifford1985] or cable-TV), then the central site could broadcast all updates and new data and the remote sites could filter it. Such an approach would be prohibitive with point-to-point communications. The update

profile can also make or break a quasi-copy strategy. By profile, we mean their frequency, periodicly, size (in bytes), whether lulls in activity exist, and so on. Similarly, the query patterns are important for caching efficiency.

In this paper we do not wish to narrow ourselves down to a particular set of base parameters, so our discussion will continue to be general. Given a set of base parameters, the implementor must make three fundamental choices that will dictate the mechanism to use:

What to Propagate. When the central site wishes to inform remote sites of an update, it can send the following types of messages:

(a) *Data Message.* A data message contains the new values. These values should overwrite those found in caches.

(b) *Invalidation Message.* An invalidation message identifies the objects that have changed, but does not contain the new values. An invalidation message usually causes the remote node to purge from its cache the referenced images.

(c) *Version Number Message.* This message identifies the objects and provides their new version numbers. The new data values are not included. (The time of update could be included instead of, or in addition to the version number. However, this information can sometimes be inferred from the message arrival time.) The remote node uses the version number to decide if it should purge an image.

(d) *Implicit Invalidation.* In this last case, the central node sends *no* message. Instead, it uses a technique called *aging* to automatically invalidate an image after a certain time. That is, when a copy is made, it is sent with a time limit. The remote node then guarantees that it will purge the image at the latest when the limit expires.

If communications are cheap, then propagating updates is reasonable. If they are

not, or in some special cases, some of the other approaches may be useful. Implicit invalidation incurs the least overhead and is especially attractive if objects are large or communications faulty. For example, train schedules can be issued, as they are in reality, with an expiration date. When the schedule expires, a new copy must be requested explicitly if there is still interest. Of course, implicit invalidation works best when the time of future update can be predicted (or controlled). This is the case with train schedules. If modifications occurred at unpredictable times, then the expiration date issued with the schedule would not be very useful.

Invalidation and version number messages also have reduced communication overhead, but have more than implicit invalidation. They are especially attractive for broadcast environments, where the central node can inform everyone of changes but only those that actually need the new data request it. One application that already uses this strategy is catalogs for department stores. When a new catalog appears, all customers are mailed a postcard informing them. Interested customers must then pick up their copy at the store. Catalogs can also have expiration dates. Usually, the new catalog appears months before the old one expires, and this is why postcards are sent. This also illustrates that a mixture of mechanisms can be utilized. Version number messages are desirable when version replication conditions have been specified. If the messages are broadcast, the work of checking versions can be off loaded to the remote site. Each remote site can check its own conditions and decide what data must be purged.

When to Propagate. When an update arrives at the central site, it does not have to be propagated immediately. As a matter of fact, there are several choices:

(a) *Last Minute.* The updates can be delayed up to the point where a replication or selection condition can be violated.

(b) *Immediately*. Updates could be propagated as soon as they occur.

(c) *Early*. Updates can also be propagated at any other time, after they arrive but before a condition is violated.

(d) *Late*. A last choice is to delay the installation of an update at the central site. If the update is not installed, the conditions cannot be violated, and the propagation can be delayed.

Immediate propagation should only be used when the selection and replication delays require it, or when evaluating the conditions is too expensive. Last minute propagation has the greatest potential for reducing communication costs since it allows as many as possible updates to be ''batched'' together. However, sometimes early propagation can take advantage of lower communication tariffs or processor idle time. For example, telephone calls are usually more expensive from 8:00 AM to 5:00 PM. Suppose that a delay condition $W(x) = 5$ hours has been defined, and that updates to x have taken place at 7:00 AM. If the telephone is going to be used, it is clearly better not to use last minute propagation (at 12:00 Noon), and to transmit the new data just before 8:00 AM.

Late propagation gives us even greater flexibility and potential for batching together even more updates. However, delaying updates at the central site is an inconvenience since the database there is made to diverge from the ''real world.'' Hence, late propagation is only an option when the applications can tolerate such divergence.

Conceptually, delaying updates at the central site is like increasing the delay window $W(x)$ of the objects. However, the advantage of delayed updates is that it can make implicit invalidation work more efficiently. When a remote node requests a copy of object x, the central site can respond with an expiration time t_e in the future. Now, if

updates can be put off until time t_e, the copy will not have to be explicitly invalidated.

To illustrate, consider a VLSI design IRS. A user wants to examine a particular microprocessor chip at his workstation for one day, and has defined a window $W(x) = 1$ hour. If updates cannot be delayed, they would have to be transmitted to the workstation at most one hour after they take place. If the microprocessor is large, this entails a lot of effort. In addition, the user will be inconvenienced as the chip changes while it is being examined. A better approach in this case may be to "check out" the chip, as in a library. Modifications are postponed at the central site for 23 hours after the copy is sent out. After the day is over, the user can request a new copy if he is still interested.

Division of Labor. The management of quasi-copies is done jointly by the central and remote nodes. Some of the work can be done either centrally or remotely, so it is important to distribute the work effectively.

The central issue is who checks the selection and replication conditions. If the central node checks them, then it must have a model of the state of the remote node. For instance, suppose that a condition is $A(x) = 2$ and x is currently 5. An update $x = 6$ arrives. To know if the condition is violated, the central node must know the value of x'. If $x' = 5$, then the condition still holds; if $x' = 3$, then it does not. These models consume storage and processing resources at the central site and may cause it to become a bottleneck.

Checking conditions at the remote nodes eliminates the need for models. Furthermore, each node need only do the checks that involve it. On the other hand, the communication needs increase since the central node must transmit the data used in the checks.

As mentioned earlier, a high bandwidth broadcast channel may make it feasible to

off load work to the remote sites. One possibility is to broadcast the entire database periodically and let each node take what it needs [Gifford1985]. In this case, the remote nodes check all conditions. Other possibilities are to broadcast only new data and updates, version numbers, or invalidation. As the amount of data transmitted decreases, the work at the central site increases.

One type of check that is especially well suited for the remote nodes is the multi-object one. In this case, a constraint like "number of reservations is less than or equal to the number of available seats" must be enforced. As updates arrive at a remote site, these checks can be made. Missing updates can be requested (or data can be purged) if the constraints are violated.

Other Decisions. In addition to what and when to propagate, and the division of labor, there are other important decisions to make. Here we will briefly touch on some of them, mainly to illustrate their diversity.

So far we have assumed that each image can have its own conditions. However, a number of restrictions can be made to make the management of quasi-copies simpler. For instance, we might require that the images of a given object at all nodes have the same conditions and values. Similarly, we can require that all images at a node have the same condition (e.g., all data at this node will have a window of 1 hour).

When a remote node needs to purge an object from its cache, it needs to select a victim (out of the advisory ones). It may be possible to use any of the standard cache replacement algorithms (such as LRU, or least-frequently-used), however, it may be desirable to consider more sophisticated policies. For example, if the objects in the cache have widely different sizes (and we expect this to be the case), it may be more appropriate to delete a very large item rather than many small ones; this would be particularly effective if the large object were associated with a single user, or if scanning it

took so long that the extra cost of fetching the data from the main site could be amortized over the life of a long interaction. It is also possible that certain objects have a high probability of being accessed together (e.g., the quarterly reports of companies in a given field, or the price of stocks of companies involved in a takeover battle). Clearly, it would be beneficial to either cache all the related items or none of them. Finally, there might be data objects that, from the known semantics of the application, should always be cached. For example, during a bibliographic search the researcher might constantly refer to a set of key passages in the literature.

A related issue is whether the node should inform the central site of a purge. If the object continues to be selected, the central site will continue to send updates. If the image was purged, there will be no place to store the new values. Thus, the remote site may notify the central site to cancel the selection. On the other hand, the remote site may decide that it is not worth the effort, or that it may try to re-cache the object when an update arrives.

7. TWO EXAMPLES

In order to illustrate the use of the various concepts and mechanisms we have discussed, we now present two more complete examples. The first is the Boston Community Information System that has been implemented in MIT and is described in [Gifford1985]. The system consists of a central site with various databases, including two wire services. The users are located throughout the Boston area. The central site transmits data on the subcarries of a normal FM station. The data received at the remote sites is processed by a personal computer.

The central site is constantly transmitting the entire database. At the receiving end, each user selects the type of data that he is interested in. The PC acts as a filter and stores locally the parts of the database that are of interest.

In this system caching is clearly a must. If no local data were stored, a user query could only be answered by waiting for the answer to be transmitted. As of 1985, the time to transmit the database was 4 hours. This means that on the average a query would take 2 hours! Selection conditions are also important, for without them the entire database would have to be stored locally. The selection conditions are of type add, dynamic, and compulsory (see Section 3). The triggering delay is set by the system and is equal to the time it takes to transmit the full database. Objects are dropped from the cache manually by each user.

Data updates are not described in [Gifford1985], but could easily be incorporated into the system. Given the way data is transmitted, the only choice is to define delay conditions (see Section 4) for all objects with a window equal to the database transmission time. (Periodic conditions would be too restrictive in this case because they would force the system to broadcast at fixed times.)

Multi-object conditions, if they are required, pose an interesting problem. Say for example that there is a consistency constraint that involves objects x and y. At the end of a database transmission the images will satisfy the constraint. Let $x'(a)$ and $y'(b)$ be the images at this time. During the next database transmission, there can be a time when one image has been updated but not the other. Since the values $x'(a+1)$ and $y'(b)$ may be inconsistent, they cannot be read by the same query. The only reasonable choice is to save the old values until the end of a database transmission, at which time they can all be installed atomically. Since it would be costly to save all the old values, it would be desirable to let users define their multi-object constraints explicitly. This way, the system would only save old versions involved in these types of conditions.

In addition to broadcasting the full database periodically, the system also transmits new items with a higher frequency. This in essence defines two types of triggering

delays. When a selection condition is first installed, it may take longer to cache data that is old. However, once the initial database transmission delay takes place, new objects that satisfy the condition will be cached more promptly. The ability to broadcast some data at a higher frequency opens the door for a number of additional improvements. For example, the database could be divided into fragments and each fragment could have a different triggering delay defined. Updates do not have to be transmitted as part of the database; they can be broadcast any time after they take place. This shortens the window for the replication conditions, and again the system can give some fragments shorter windows. All of these improvements will become more important as the size of the database and its full transmission time increases.

The system designers also plan to add a dial-in capability to the system. This can let users define selection and replication conditions that are stricter than the default ones, and can include non-delay ones like version number or arithmetic conditions. The system would use these conditions to schedule the transmission of data.

As our second example, consider a calendar service that is to be provided to a hypothetical large corporation. The calendar is to be accessed from a large number of minicomputers located at the various departments and branch offices. Each calendar is a sequence of ''days'' 1, 2, 3, ... Each day object contains a list of events and descriptions, but the internals are not relevant here. There can actually be a set of different calendars, each one identified by a ''type'' (e.g., for the New York location, for the managers, etc.). The most common operation, as one might expect, is to look up the events for a given calendar and day. Of these, the lookups for the current day are the most frequent. Entries in a calendar can be added, deleted or modified. Such modifications typically must ensure that the obvious consistency constraints are satisfied (e.g., two events should not conflict).

There are a number of ways to implement the calendars. A full centralization approach is simple but leads to long delays for queries and poor availability. If the calendars are replicated but tight consistency is required, then all updates must be propagated immediately, regardless of how congested the systems and the communication network is. If control of the copies is distributed, then concurrent updates must be synchronized. This can be very difficult if network partitions occur. Furthermore, since updates originate at various places, it is not possible to batch them together for broadcast efficiency. On the other hand, a centralized solution with remote quasi-copies, while not perfect, does have some nice features. It can yield a good query response time and availability, while at the same time limiting the communication traffic.

To design the calendar system with quasi-copies we must chose the selection and replication conditions that will be enforced. Here we will illustrate by chosing a set of reasonable ones, but clearly other choices are possible. We dynamically select to cache the next 30 days, starting at the current day c, for those calendars of interest to the local community. When a new calendar type is added to a selection, we would like to get it as soon as possible, so we define a small triggering delay. We make the selections compulsory so that all queries regarding the next month can be executed locally.

We use time varying replication conditions. If a day in a calendar x represents c or $c+1$ (the current or the next day), then we define $W(x) = 15$ minutes and $V(x) = 3$. This ensures that all copies will be at most 15 minutes or three versions out of date for events occurring today or tomorrow. For days $c+2$ through $c+29$, we define a periodic condition $P(x)$ = 6am day 0, 24 hours to guarantee that the calendars are refreshed at 6am every day. In addition we define $V(x) = 3$ to ensure bursts of update activity are propagated quickly. Finally, we define each calendar to be a group of objects with multi-object constraints. This way, if a transaction schedules two events in the same calendar,

say one for day $c+1$ and the other for $c+2$, both will appear in the cached copies, even if the second one did not violate the simple conditions.

To cope with failures, it does seem convenient to have a *C_FAILED* (j) accessible to users at site j, as discussed in Section 5. The interval for sending "I am alive messages," δ, should be set at 15 minutes to be compatible with the replication conditions for the current day.

As we know, there are many ways to implement the selection and replication conditions we have given. In any case, since the conditions are simple and uniform for all calendars and sites, the overhead of managing the quasi-copies should be low. To propagate an update, sending the new value seems most appropriate here. However, implicit invalidation can be used to remove the current day from the cache at the end of the day. That is, all data broadcast for a day x includes an implicit expiration time of $x+1$, meaning that at day $x+1$ the data is no longer valid and should be purged. With respect to when to propagate, last minute propagation appears to be the most appropriate for this application. The division of labor is also clear: the central node executes updates and checks the conditions; the local nodes execute most of the queries.

8. CONCLUSIONS

In practice, quasi-copies are already in use for all types of information and data. However, they are mostly used in an ad-hoc fashion, outside of computer systems, and without any validity guarantees. In this paper we have suggested that quasi-copies can be useful in a computerized IRS, reducing substantially the overhead of managing replicated data and making data available during failure periods. We have formally defined the notion of quasi-data, presented the types of conditions it can satisfy, and discussed the mechanisms with which a system can take advantage of the added flexibility.

As our two examples illustrated, caching and quasi-copies can potentially improve performance and availability. However, there are also potential problems. In summary, (1) if there is poor locality of reference, i.e., if it is difficult to predict future accesses, then caching may not pay off; (2) if the selection and replication conditions are complex, then the overhead of checking them may outweight the savings; (3) if the user does not understand his application, then it may be difficult for him to define useful conditions; and (4) the user may need to see the latest values in the database, in which case quasi-copies are not of much use. All of these items must be kept in mind when considering caching as an alternative.

Caching and quasi-copies also raise a number of challenging optimization questions: How much data should be cached? Which of the mechanisms we have outlines is better suited for a particular application? At the user end, there are also open questions: What language is used to define conditions? How does a user determine the delays or deviations that are best suited for him? We believe that these questions will be answered as we obtain more experience with caching and quasi-copies.

9. REFERENCES

Date1975. Date, C. J., ''An Introduction To Database Systems,'' *Addison-Wesley* (1975).

Davidson1985. Davidson, Susan B., Garcia-Molina, Hector, and Skeen, Dale, ''Consistency in Partitioned Networks,'' *ACM Computing Surveys* **17**(3) (September 1985).

Dunn1984. [Interview], ''Bill Dunn of Dow Jones: The Data Merchant,'' *Personal Computing*, pp. 162-176 (December 1984).

Edelhart1983. Edelhart, Mike and Davies, Owen, ''OMNI Online Database Directory,'' *Collier MacMillan Publishers* (1983).

Gifford1985. Gifford, David K., Lucassen, John M., and Berlin, Stephen T., ''The Application of Digital Broadcast Communication to Large Scale Information Systems,'' *IEEE Journal on Selected Areas in Communication* (May 1985).

Lamport1978. Lamport, Leslie, ''Time, clocks, and the ordering of events in a distributed system,'' *Communications of the ACM* **21**(7), pp. 558-565 (July 1978).

Russell1986. Russell, George, ''Manic Market,'' *Time (international edition)*, pp. 64-70 (November 10, 1986).

Smith1982. Smith, Alan J., ''Cache Memories,'' *Computing Surveys* **14**(3) (September 1982).

Stonebraker1982. Stonebraker, Michael and Rowe, Lawrence T., ''Database Portals: A New Application Program Interface,'' Electronics Research Laboratory Report UCB/ERL M82/80, U.C. Berkeley (November 2, 1982).

Walsh1985. Walsh, Dan, Lyon, Bob, and Sager, Gary, ''Overview of the Sun Network File System,'' *Proceedings USENIX Winter Conference 1985* (January 1985).