THE DEMARCATION PROTOCOL: A TECHNIQUE FOR MAINTAINING ARITHMETIC CONSTRAINTS IN DISTRIBUTED DATABASE SYSTEMS

Daniel Barbara
Hector Garcia-Molina

CS-TR-320-91

April 1991

The Demarcation Protocol: A Technique for Maintaining Arithmetic Constraints in Distributed Database Systems

Daniel Barbará
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

ABSTRACT

Traditional protocols for distributed database management have high message overhead, lock or restrain access to resources during protocol execution, and may become impractical for some scenarios like real-time systems and very large distributed databases. In this paper we present the demarcation protocol; it overcomes these problems through the use of explicit arithmetic consistency constraints as the correctness criteria. The method establishes safe limits as ''lines drawn in the sand'' for updates and gives a way of changing these limits dynamically, enforcing the constraints at all times.

April 23, 1991

The Demarcation Protocol: A Technique for Maintaining Arithmetic Constraints in Distributed Database Systems

Daniel Barbará
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, NJ 08544

1. Introduction

Traditional protocols used to manage distributed data, such as two-phase commit, require one or more rounds of messages, lock or restrain access to resources during the protocol execution, and have a high overhead. Moreover, these protocols could render data unavailable during failure periods, decreasing system availability. This may be impractical for some scenarios, like those involving very large distributed data bases or real-time systems. In this paper we propose an alternative protocol that overcomes these problems through the use of explicit consistency constraints as the correctness criteria. The method gives the nodes involved substantial autonomy for performing changes to individual data items.

To illustrate, consider an application where ''items'' are stored in two separate locations. Assume there must be sufficient stock of these items among the two locations. This could, for instance, correspond to a military application where planes are stationed at two different bases, with the requirement that at all times the total number of stationed planes cannot be below some specified limit. (Perhaps out of fear of being short of planes in case of an attack.) Or it could also correspond to a retail business application where the ''parts'' stock at two warehouses must be maintained above a certain limit all the time. Transactions will run trying to withdraw or add to the stock, and the system should verify that the constraint is obeyed at all times. In the military application, planes can be put out for maintenance, or sent off to missions, but the minimum number of planes should be kept stationed at all times.

In this example, we have two nodes, each storing the value of the stock kept at the location. Let A and B be the data items at locations a and b respectively. Let the constraint be $A + B \geq 100$. Consider a typical transaction that attempts to withdraw Δ units from A:

$$\text{If } A - \Delta + B < 100 \text{ then abort}$$
$$\text{else } A = A - \Delta$$

In a conventional transaction processing system, such a transaction would have to lock data item B at location b, and A at a, verify whether the updated value satisfies the constraint and in that case, update A and follow a two-phase commit protocol to ensure that the transaction commits. Notice that this would require two rounds of messages, and will lock and limit access by other transactions to A and B during protocol execution, resulting in a high overhead. The protocol could also render the data unavailable if one node or the network fails during execution, thus decreasing the availability of the system. This can occur because the two-phase commit protocol can block, i.e., after a failure the nodes may not be able to determine the outcome (commit or abort) of the transaction. Therefore, the nodes cannot release the locks and other transactions cannot run.

Alternatively, we could have a variable A_l at node a that acts as a limit, and state that transactions can continue withdrawing units of A as long as the final value of A remains above A_l ($A \geq A_l$). Similarly, a variable B_l will be stored in node b, serving as a limit for the updates made to B ($B \geq B_l$). (Think of A_l and B_l as ''lines drawn in the sand.'') The transactions will produce correct results as long as we ensure that $A_l + B_l \geq 100$. Notice that now, A (or B) can be modified by a transaction without involving the other node, as long as the updated value remains larger than the limit A_l (B_l). In these cases, what used to be a global transaction has become a local one, so there is no need for global concurrency control nor a two phase commit protocol. This increases data availability since transactions of this type may run even if the other node (or the network) is unavailable. One would expect that in many applications there is often slack in the constraints, so that in a very large number of cases transactions will be able to run locally as illustrated. In the aircraft example, say each base has 80 aircraft and that a total of 100 aircraft must be kept between the two bases. If we set $A_l = B_l = 50$, we satisfy the global constraint, leaving each base with a slack of 30

aircraft. Thus, each base could dispose of 30 aircraft without consulting the other.

The limits do not have to be static, but can change over time as needed. However, the changes have to be made in such a way that the constraint $A_1 + B_1 \geq 100$ is obeyed at all times. This is the goal of the *demarcation protocol* presented in this paper. The protocol is designed for high autonomy. Clearly, some changes in limits must be delayed because they are not ''safe;'' however, these delays do not block safe changes nor transactions that operate within the current limits. Besides a protocol for changing limits, we also need a *policy* for selecting the values of the desired new limits. (In our aircraft example, do we set $A_1 = B_1 = 50$ or should we pick $A_1 = 30, B_1 = 70$?) Policies will also be discussed in this paper.

Our approach does *not* guarantee global serializable schedules. Local executions are serializable, though, since each node uses conventional techniques (e.g., two phase locking) to run local transactions. Our approach does guarantee that inter-node constraints, which we assume are given, are satisfied. A conventional global concurrency control mechanism, on the other hand, would guarantee globally serializable schedules and the satisfaction of all constraints without the need to explicitly give them to the system.

One could argue that having to explicitly list the global consistency constraints is a disadvantage of our method. We can counter this argument, first by noting that with conventional approaches programmers still have to know and understand consistency constraints in order to write transactions. (A transaction must preserve all constraints.) Second, by knowing the constraints, the system can exploit their semantics, yielding better availability and performance.

Third, we are only talking of specifying constraints that span more than one node (local control ensures local constraints are satisfied). If we look at inter-node consistency constraints in practice, we see that they tend to be very simple. It is very unlikely for instance that we encounter an application with employee records in New York and an

index to those records in Los Angeles. Data that is closely interrelated tends to be placed on a single node. † If we look at the types of constraints that are found in databases [DAT83], we claim that the following ones are the most likely to involve data stored in different nodes of a distributed system:

(1) Arithmetic inequalities. For example, the available funds for a customer at an ATM machine should be less than or equal to the actual balance of his or her account.

(2) Arithmetic equalities. For instance, the hourly wage rate at one plant must equal the rate at another plant.

(3) Referential integrity constrains. Example: if an abbreviated customer record exists on one node, then the full customer record must exist at headquarters.

(4) Object copies. The employee benefits brochure at a site must be a copy of the brochure at the personnel office.

Our main focus in this paper will be on arithmetic inequalities. If an arithmetic equality is tight, e.g., $A = B + \delta$, then maintaining it will be expensive. Every change involves two phase commit or the equivalent, since each time A changes, B must immediately change. However, in many applications a tight equality can be treated as an inequality, e.g., $|A - B - \delta| \leq \varepsilon$. This is simply the two constraints $A - B - \delta \leq \varepsilon$ and $-A + B + \delta \leq \varepsilon$, which can be handled via our demarcation protocol. In Section 6 we show that the same principles that are used for arithmetic constraints can be applied to existential and copy constraints.

This paper is organized as follows. We start by listing related research in Section 2. In Section 3 we offer additional examples and an informal description of our approach. The protocol and a policy under a particular arithmetic inequality are presented in Section 4, while the generalization to arbitrary arithmetic constraints is in Section 5. In Section 6 we discuss other types of constraints.

---

† In a parallel database machine or in a local cluster, there may be complex inter-computer constraints. But in this case, the autonomy of each computer and network delays are not the critical issues. We are focusing on geographically distributed systems.

2. Related Research

There has been a lot of recent interest in trying to reduce the delays associated with conventional transaction processing and on exploiting semantics. In this section we briefly summarize research that is related to the demarcation protocol.

The idea of setting limits for updates has been suggested informally many times, e.g., [HS80, DAV82]. These ideas were formalized by Carvalho and Roucairol [CR82] in the context of enforcing assertions in distributed systems. Their limit changing protocol is more general than ours, but it is substantially more complex. For example, to maintain a constraint distributed among n nodes (each holding one variable) requires n^2 limit variables, while our protocol only uses n limits, one at each node. Their protocol requires many more update messages when limits are changed, and also forces changes to be done serially. Furthermore, [CR82] does not discuss policies for changing limits.

The demarcation protocol is related in a way to O'Neil's escrow mechanism [ONE86]. This technique was devised to support high-speed transaction updates without locking items, by enforcing integrity constraints. However, the mechanism was designed to be used only in a single database management system. (The application to the management of replicated data was proposed in the paper as a research topic.) Also, Kumar and Stonebraker [KuS88] have proposed a strategy for the management of replicated data based on exploiting application semantics. They present an algorithm to implement the constraints $B > B_{\min}$ and $B < B_{\max}$. However, their technique relies heavily in the commutativity of the transactions involved and does not generalize to arbitrary arithmetic constraints. Soparkar and Silbershatz [SS90] have developed a protocol to partition a set of objects across nodes. A node may ''borrow'' elements from neighbors. This approach does not deal with arbitrary constraints but does guarantee serializable global schedules.

The notion of quasi-copies is defined in [ABG90] as a way of managing copies that may diverge in a controlled fashion. The goals of this work are the same as for the demarcation protocol. However, quasi-copies are not useful for arithmetic constraints. For managing copies, the notion of quasi-copies is more flexible in some ways. For instance, one may specify that a copy must be equal to some value that the primary had within the last 24 hours. This type of constraint is not handled by the demarcation

protocol. On the other hand, with quasi-copies updates may only occur at a primary location. The demarcation protocol allows updates at any site containing an arithmetic value.

Finally, there are other papers that deal with weaker notions of serializability and use of application semantics [FGL82], [LBS86], [GAR83], [KS88], [DE89], [FZ89].

3. Examples

In this section we present two examples that illustrate how the demarcation protocol works and the kinds of problems that are to be dealt with.

3.1 The Sufficient Supply Problem

We return to the example presented in Section 1. The proposed method of operation imposes some restrictions. As pointed out, a transaction that attempts to lower A (or B) under the limit A_l (B_l) would be aborted. The only way to run such a transaction would be first to get the site to lower its limit. Since this is not a ''safe'' operation (lowering A_l may violate the constraint $A_l + B_l \geq 100$), it can only be achieved by asking the other node to raise its limit first. The only safe operation in this example is to raise the limit above the current value.

To see how limits can be changed, assume that $A = 61$ while $B = 69$, and the limits are chosen initially to be $A_l = 45$ and $B_l = 55$. Figure 3.1 shows the setting of the base scenario for this example. Notice that transactions at node a can update A without further intervention from b, as long as the final value remains greater than or equal to 45. The same is true for b and B, as long as $B \geq 55$. Assume now that for some reason, node a wants to raise its limit A_l to 50. Since this is a safe operation, a can go ahead and do it. Node a will send a message to b informing it of the increment to A_l by 5 units. Upon receipt of this, b may lower B_l by 5 to 50 (an originally unsafe operation). No reply to a is necessary. If instead of raising A_l, a wants to lower it, the situation is not as easy. Lowering the limit is an unsafe operation in this case, so node a would have to send a request to b asking it to raise B_l by the necessary amount. Node b is free to reject this request. However, if it does honor it by raising B_l, it will send a a message informing it, and only then a could lower A_l. If the nodes follow this protocol, it can be assured that

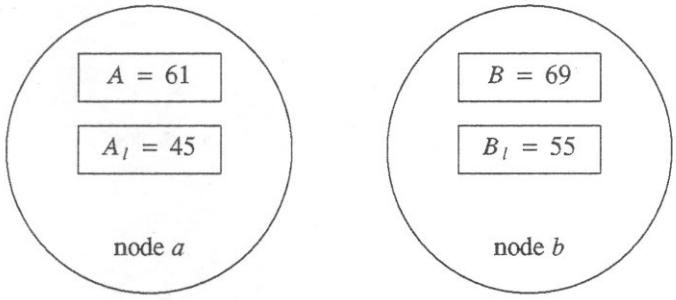

Figure 3.1 The base scenario for the sufficient supply problem

at all times $A_l + B_l \geq 100$.

This is essentially the way the demarcation protocol operates. Whenever a node wishes to perform an unsafe operation, it requests that the other node perform a corresponding safe operation and waits for notification. Notice that the demarcation protocol is not two-phase commit. (The decision to give another node slack by increasing a limit is made by one node only.) The nodes are still autonomous and there is no need for locking remote resources. While limits are being changed, transactions that modify A or B can still run (as long as $A \geq A_l$, and $B \geq B_l$.)

There are still two issues to be discussed. The first is the establishment of a policy for when to invoke the protocol. The policy is orthogonal to the protocol, and the changes can be triggered at any time. For instance, the changes could be triggered whenever A (B) gets too close to A_l (B_l). The second issue is the selection of the new limits. Each node needs a formula for computing the new limits when a change is to be made. For instance, in this example it may be desirable to split the ''slack'' evenly, i.e., to make the distance between A_l and A equal to the one between B_l and B.

As an example, using the initial values of Figure 3.1, consider a transaction that updates B to 57. Say that $B - B_l = 2$ is considered ''close,'' so the change mechanism is triggered. However, node b does not know how to split the slack since it does not know the value of A, plus it cannot lower B_l safely anyway. Therefore, b sends a message to a requesting that A_l be raised, and including the current value of B. Upon receipt of this message, a knows that $B = 57$ and $A = 61$. The slack is $A + B - 100 = 18$. Subtracting half of this from each value we get $A_l = A - 9 = 52$, and $B_l = B - 9 = 48$. These

are the new limits that split the slack evenly. Finally, a can safely increase A_l from 45 to 52 (increment of 7), sending a message to b, allowing it to decrement B_l by 7 to 48.

3.2 The Budget Problem

So far we have only discussed one type of constraint, i.e., $A + B \geq \delta$. Do the ideas generalize to other types of arithmetic inequalities? Fortunately, they do. For any constraint, some operations are safe while others are not. To illustrate, consider the following example: ensuring that project expenses E do not exceed budget B. Assume that we store B at a node b and E at another node e. Node b could be located at company headquarters, while e is at the project location. We require that at all times $E \leq B$. Both the expenses and the budget get to be updated over time. For the demarcation protocol we shall keep two limits E_l and B_l such that $E \leq E_l$, $B_l \leq B$, and $E_l \leq B_l$. In the supply example, it was safe to increase the limits of both variables. In this budget problem, the limit E_l can be *decreased* safely by e, while B_l can be *increased* safely by b. On the other hand, the operations of incrementing E_l at e, or decrementing B_l at b are unsafe. Notice again that once one node performs a safe operation, it leaves room for the other to perform what was originally an unsafe change. Consider that initially $E = 0$, $B = 20$ and that limits E_l and B_l have been set to 10. As long as E stays under 10, node e is free to modify E without consulting b. Similarly, node b can lower B to 10.

4. A Protocol and a Policy

In this section we present the demarcation protocol and its associated policies. To simplify the explanation, in this section we assume a particular constraint, $A \leq B + \delta$. In Section 5 we show how to generalize to an arbitrary inequality and to more than two variables. We also show a particular policy choice for splitting slack, although many others are possible.

4.1 The Demarcation Protocol

The protocol consists of two operations, one for changing a limit and one for accepting the change performed by the other node. Recall that the constraint we are dealing with is $A \leq B + \delta$. Let the predicate SAFE(X,σ), where X is either A or B, and

σ is a desired change in value, be defined as follows:

$$\text{SAFE}(X,\sigma) = \begin{cases} \text{TRUE} & \text{if } X = A \text{ and } \sigma \leq 0 \\ & \text{or } X = B \text{ and } \sigma \geq 0 \\ \text{FALSE} & \text{if } X = A \text{ and } \sigma > 0 \\ & \text{or } X = B \text{ and } \sigma < 0 \end{cases}$$

Essentially, SAFE is TRUE if we are trying to decrement the limit of A or increment the limit of B. The other two operations are unsafe.

Let us also define the following predicate to signal when the change in a limit exceeds its data value:

$$\text{LIMIT\_BEYOND}(X,\sigma) = \begin{cases} \text{TRUE} & \text{if } X = A \text{ and } A_I + \sigma < A \\ & \text{or } X = B \text{ and } B_I + \sigma > B \\ \text{FALSE} & \text{otherwise} \end{cases}$$

When we refer to one of the values as X, we will use Y to refer to the complementary variable in the constraint $A \leq B + \delta$, i.e., $Y = A$ if $X = B$ and viceversa. We use the notation $N(X)$ for the node holding X. The demarcation protocol is composed by two procedures: *change_limit*() and *accept_change*():

P1: *The Demarcation Protocol*

change_limit(X,σ)
  if SAFE(X,σ) is FALSE *then*
    Send message to $N(Y)$ requesting it to perform **change_limit**(Y,σ)
  else if LIMIT_BEYOND(X,σ) is TRUE *then*
    abort the change
  else
    { $X_I \leftarrow X_I + \sigma$;
    send message to $N(Y)$ requesting it to perform **accept_change**(Y,σ) }.

accept_change(Y,σ)
  $Y_I \leftarrow Y_I + \sigma$.

Conventional database techniques should be used at each node to make changes in the limits and variables atomic and persistent. For instance, the values of the limits should

not be lost due to a node failure. Loss of messages is undesirable but does not cause the constraint to be violated. For example, say $N(A)$ decreases its limit by 5 and sends an *accept_change* message to $N(B)$. If this message is never delivered, B_l will be 5 units higher that it need be. This is safe, but means that $N(B)$ will be unable to ''use'' these 5 units. Thus, for proving our protocol correct (Theorem 4.1) we make no assumptions about message delivery. However, in practice it is desirable to use persistent message delivery (messages are delivered eventually, without specifying how long this might be). Also note that since messages from different calls to *change_limit* only include decrements/increments, they need not be delivered in order at the other node.

For simplicity, in the protocol we have assumed that nodes are always willing to cooperate with their partners. In reality, nodes need not comply with *change_limit* or *accept_change* requests. When node *a* gets a request from *b* to decrease A_l by 10 units, *a* is free to ignore the request, decrease A_l by 10, or to decrease A_l by whatever amount it wishes. Similarly, when *a* receives *accept_change*(A, 10), it may increase *A* by any amount up to 10. Of course, in most cases it is advantageous for *a* to perform the full increment, as indicated by the code. This is what we assume here.

Theorem 4.1: The demarcation protocol ensures that at all times $A_l \leq B_l + \delta$, assuming that the system starts with limits A_l^0 and B_l^0, where $A_l^0 \leq B_l^0 + \delta$.

Proof: All the increments or decrements to limits are done by adding a value v_i or subtracting a value u_i to the old limits, where *i* is simply an ascending index. For data value *A*, v_i^A represents a change performed via the *accept_change* call, while u_i^A represents a change made via *change_limit*. For *B*, v_i^B represents a change made using *change_limit*, and u_i^B one made using *accept_change*. At any time, we have

$$A_l = A_l^0 + \sum_{i \leq k_1} v_i^A - \sum_{i \leq k_2} u_i^A$$

and

$$B_l = B_l^0 + \sum_{i \leq k_3} v_i^B - \sum_{i \leq k_4} u_i^B,$$

where k_1 and k_3 are the indexes of the last increments seen by nodes $N(A)$ and $N(B)$ respectively, and k_2 and k_4 are the indexes of the last decrements seen by nodes $N(A)$ and $N(B)$ respectively.

Every increment v_i^B performed by *change_limit* produces an equivalent increment v_j^A done by *accept_change*. The increments for A may not be done in the same order as for B. However, the increment v_i^B is always done before the increment of the same magnitude v_j^A is done. Thus,

$$\sum_{i \le k_1} v_i^A \le \sum_{i \le k_3} v_i^B$$

By a similar argument,

$$\sum_{i \le k_4} u_i^B \le \sum_{i \le k_2} u_i^A$$

We can combine these two inequalities with $A_l^0 \le B_l^0 + \delta$ by adding all left hand sides and all right hand sides, obtaining that at all times

$$A_l \le B_l + \delta \quad \bullet$$

Corollary 4.1 Using the demarcation protocol, we ensure that at all times $A \le B + \delta$.

Proof: The line in *change_limit* that checks LIMIT_BEYOND(X, σ) ensures that $B \ge B_l$ and $A \le A_l$. Then, by Theorem 4.1, the result follows. •

Corollary 4.2 Suppose that $A_l^0 = B_l^0 + \delta$, that all update activity stops and that all messages have been delivered. Then $A_l = B_l + \delta$ (note equality).

Proof. Same as the proof for Theorem 4.1, except that after all messages are delivered,

$$\sum_{i \le k_1} v_i^A = \sum_{i \le k_3} v_i^B \qquad \text{and} \qquad \sum_{i \le k_4} u_i^B = \sum_{i \le k_2} u_i^A \quad \bullet$$

4.2 Policies

The policies associated with the demarcation protocol specify when to initiate limit changes, how to compute new limits, and what to do in case a transaction tries to change the data value beyond its limit. We describe here the framework for such policies, and present some choices for them. However, the reader should bear in mind that other policies exist.

We begin by explaining how a transactions will actually perform changes on the data items. In order to change a value, a transaction will use a system call *change_value*(X,θ), where X is the data item and θ is the amount (positive or negative) to change. Within this call, we will have invocations of three policies. The first will be triggered whenever the change would exceed the limit. The second will be fired up when the final value gets ''too close'' to the limit. The last one will be triggered if the final value gets ''too far'' from the limit. These policies are implemented by procedures associated with the constraint that we are enforcing. Since a data value may be involved with more than one constraint (and thus, have more than one limit), we will have to test the limits on a per constraint basis. We will denote the constraints by the symbol Φ_j, where $1 \leq j \leq m$ and m is the number of constraints. Up to this point we have only discussed the constraint $A \leq B + \delta$, but we are now generalizing to emphasize that the policies are constraint dependent.

Before presenting the *change_value* procedure, we introduce the following predicate, for the case Φ_j is the constraint $A \leq B + \delta$. (A generalization is presented in Section 5.)

$$\text{VALUE\_BEYOND}(X,\theta) = \begin{cases} \text{TRUE} & \text{if } X = A \text{ and } A + \theta > A_l \\ & \text{or } X = B \text{ and } B + \theta < B_l \\ \text{FALSE} & \text{otherwise} \end{cases}$$

This predicate is analogous to LIMIT_BEYOND, in this case checking whether a change would violate the limit constraint.

The system call for changing a data value is as follows. Again, X refers to the variable being changed (A or B is our sample constraint). The limit for X under constraint Φ_j is X_{l_j}. Parameter T_{code} is a pointer to the code that runs the calling transaction and is explained below.

change_value(X, θ, T_{code})
  for each constraint Φ_j ($1 \leq j \leq m$) in which X is involved *do*
    { *if* Φ_j.VALUE_BEYOND(X, θ) is TRUE *then*
      { Fire up process $\Phi_j.policy1(X, \theta, T_{code})$ at $N(X)$
      abort calling transaction; }
    if $|X + \theta - X_{l_j}| < \Phi_j.\varepsilon$ *then*
      Fire up process $\Phi_j.policy2(X, \theta, T_{code})$ at $N(X)$;
    if $|X + \theta - X_{l_j}| > \Phi_j.\beta$ *then*
      Fire up process $\Phi_j.policy3(X, \theta, T_{code})$ at $N(X)$ }
  $X \leftarrow X + \theta$.

The procedures $\Phi_j.policy1$, $\Phi_j.policy2$ and $\Phi_j.policy3$ are associated with the constraint Φ_j. The first policy is invoked when a change exceed one of the limits. In some cases *policy*1 may be null, but in other it may be important to initiate some action, like for example trying to increase the limit. The code pointer parameter T_{code} is useful for restarting the transaction once the limit has been changed. (An alternative would be to move the ''abort transaction'' command in *change_value* into *policy1*. This way, *policy1* could chose between aborting the transaction or simply delaying it until the limit has been successfully changed.) The second procedure deals with the case where the value is getting close to the limit. What close means is defined by the constraint specific constant $\Phi_j.\varepsilon$. It may be desirable at this point to initiate a change in limits. For instance, if a base is running low on planes, it may be a good idea to renegotiate its limits with the other base. The third procedure handles the case in which the value is getting too far away from the limit, as defined by constant $\Phi_j.\beta$. In our example, if a base has too many planes, it may wish to notify the other base to arrange new limits.

A fourth policy is needed to cover the case where procedure *change_limit* encounters a change that exceeds the data value. In Section 4.1, we opted for aborting the change, but in general we can have a policy that decides what action is to be taken. This $\Phi_j.policy4$ can be invoked when the LIMIT_BEYOND check in *change_limit* detects a violation. Due to space limitation we will not present the modified *change_limit* procedure here; a more general version is presented in Section 5.

The policies must also implement some form of load control. For example, say the system has reached a point where $A = A_l$ and $B = B_l$. Transactions that attempt to

increase A will repeatedly try to increase A_l by sending a message to $N(B)$ to increase B_l. Since B_l cannot be raised, all these attempts will fail. Instead of wasting $N(B)$'s time, $N(A)$ may remember how many times it has attempted to change the limits and based on that, decide to wait for a given period of time before trying again.

Finally, there is the issue of how to compute the limits. A formula should be agreed upon to compute the new values when needed. We illustrate one possible formula for splitting the available slack, for the constraint $A \leq B + \delta$. (A generalization is presented later.)

$$A_l = A + (B - A + \delta)k \tag{4.1}$$

$$B_l = A_l - \delta$$

Equation (4.1) is derived by computing the slack between A and B, and giving a fraction k of it to variable A as the ''room to move up''. The remaining $1 - k$ of the slack is given to variable B. By setting up k, one can tune the system in order to favor or constraint more one of the two variables. By setting $k = 0.5$ one divides the interval evenly.

Notice, however, that computing the new limits by using (4.1) requires information about both variables A and B, and neither one of the nodes know it. However, we can design *policy2* and *policy3* to overcome this problem as follows.

Φ_j.**policy2**(X, θ, T_{code})
send message to $N(Y)$ requesting it to perform Φ_j.**split_slack**(X)

Φ_j.**policy3**(X, θ, T_{code})
send message to $N(Y)$ node requesting it to perform Φ_j.**split_slack**(X)

Φ_j.**split_slack**(X);
*** local variable is Y; value of remote variable is parameter X; ***
compute new limits, $Y_{l_j}^{new}$ and $X_{l_j}^{new}$, using Eq. (4.1) and X, Y values.
let $\sigma = Y_{l_j}^{new} - Y_{l_j}$;
if SAFE(Y, σ) *then* invoke Φ_j.**change_limit** (Y, σ)
else send message to $N(X)$ requesting it to perform Φ_j.**split_slack**(Y)

Notice that we prefixed the function *split_slack* with the constraint identification Φ_j. In general, the way the slack is split will depend on the specific constraint. (The

generalization of Eq. (4.1) is given in Section 5.) It may be desirable to include a timestamp in a *split_slack* message, so that the receiving node may discard messages that took ''too long'' in transit and represent stale data. Also note we have ignored load control issues in these policies.

To illustrate how limit changing and slack splitting work, consider the following example. Assume that we are using the single constraint $A \leq B + \delta$ so that all our constants and policies refer to it. Initially $A = 0, B = 20$, with $\delta = 10, \beta = 20$ and $\varepsilon = 2$. The limits have been set to $A_l = 15$ and $B_l = 5$, assuming that we are using $k = 1/2$. Figure 4.1 shows the initial scenario.

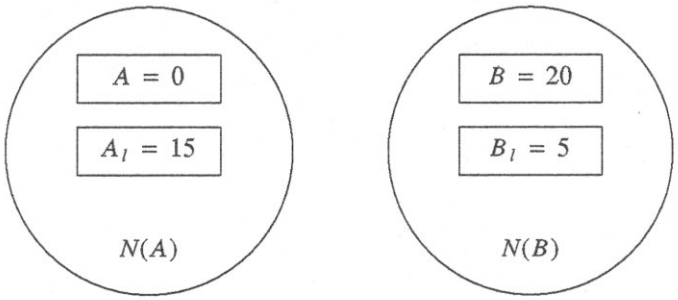

Figure 4.1 Initial scenario.

Now assume that a transaction calls *change_value* at $N(B)$ to update B to 30. (Notice that this update is allowed since $B \geq B_l$.) Since $B - B_l > \beta$, the request to change limits is initiated by $N(B)$, by triggering *policy3*. A message is sent to $N(A)$ requesting it to perform *split_slack*($B = 30$). Using Equation 4.1, node $N(A)$ computes the new limits obtaining a value of 20 for A_l. (The slack is 30 - 0 + 10 = 40; half of this is added to A to give the desired new limit.) To obtain $A_l = 20$, $N(A)$ must add 5 to the current limit; since this is an unsafe increment it does not perform the change. Instead, it sends a message to $N(B)$ requesting it to perform *split_slack*($A = 0$). Node $N(B)$ will compute the same limits, and this time the change in B_l will be performed there, updating it to 10. Node $N(B)$ will send a message to $N(A)$ requesting *accept_change*(A,5). Finally, $N(A)$ will raise A_l to 20. This example illustrates the case where *split_slack* is called twice. In other cases, *split_slack* is only called once (e.g., from initial scenario, A is updated to 15).

Now consider a scenario with concurrent updates. Starting with the values of Figure 4.1, assume both $N(A)$ and $N(B)$ complete transactions. The transaction at $N(A)$ updates A to 13, while the one in $N(B)$ updates B to 5. Both nodes send their new values to the other node, since both want to change their limits. Upon receipt at $N(A)$ of the new value of $B = 5$, the limit A_l is computed to be 14 and therefore the change is made, and a message to $N(B)$ is sent requesting it to perform *accept_change*$(B, -1)$. In the meantime, $N(B)$ processes the first message from $N(A)$, computing a desired new limit B_l^{new} of 4. Since this change is not safe, a message is sent to $N(A)$ requesting that the change is made there first. Next, each node receives the second message generated by its partner. When the *accept_change*$(B, -1)$ arrives at $N(B)$, B is updated to 4. When the second *split_slack*$(B = 5)$ message arrives at $N(A)$, it is ignored. ($A = 13$, $B = 5$ in Eq. (4.1) implies that A_l should be 14, but that is already the value of the limit, so nothing is done.) The final values are thus $A_l = 14$, $B_l = 4$, which evenly split the slack. Notice how the limits converge to the correct values, without any tight coordination. Each node still makes decisions autonomously.

In general it is difficult to prove formal properties for the policies, first because they can be arbitrary programs, and second because on purpose we do not wish to guarantee any properties that may hurt autonomy. For instance, for slack splitting, one may be tempted to prove that the selected limits are indeed the ones that split the current slack. However, there is no such thing as the ''current slack'' as the nodes may autonomously change A and B at any time (within limits). To enforce such a property, we would have to restrict updates in one way or another, which is clearly undesirable.

In the sample slack splitting policies, we simply assume that the value received in a *split_slack* message is current, and act accordingly. If indeed it is current, then the slack is split properly; if not, the selected limits will be out of date. But remember that no matter what the policies do, the underlying constraints are always guaranteed by the demarcation protocol.

5. Generalizing the Protocol

In this section we generalize the demarcation protocol to operate on an arbitrary constraint of the form $c_1A + c_2B \leq \delta$, where $c_1, c_2 \neq 0$. We also discuss how to extend it to more than two data items.

We start by generalizing the predicates SAFE, LIMIT_BEYOND and VALUE_BEYOND for the constraint Φ_j: $c_1A + c_2B \leq \delta$. As before, we use X to refer to either A or B, and X_{l_j} for its limit. We use the notation c_{X_j} to refer to the corresponding constant in the constraint (either c_1 or c_2).

$$\Phi_j.\text{SAFE}(X,\sigma) = \begin{cases} \text{TRUE} & \text{if } c_{X_j}\sigma \leq 0 \\ \text{FALSE} & \text{otherwise} \end{cases}$$

(For instance, for the constraint $A \leq B + \delta$, if we wish to increase the limit B_l by $\sigma > 0$, the predicate SAFE would be TRUE.)

$$\Phi_j.\text{LIMIT\_BEYOND}(X,\sigma) = \begin{cases} \text{TRUE} & \text{if } (X_{l_j} + \sigma - X)c_{X_j} < 0 \\ \text{FALSE} & \text{otherwise} \end{cases}$$

(For instance, for the constraint $A \leq B + \delta$, the predicate is true if $A_l + \sigma < A$.)

$$\Phi_j.\text{VALUE\_BEYOND}(X,\theta) = \begin{cases} \text{TRUE} & \text{if } (X + \theta - X_{l_j})c_{X_j} > 0 \\ \text{FALSE} & \text{otherwise} \end{cases}$$

(For the constraint $A \leq B + \delta$, the predicate is true if $A + \theta > A_l$.)

Procedure *change_value* (Section 4.2) is unchanged, except that it uses the new definition of VALUE_BEYOND given here. Procedure *accept_change* (Section 4.1) is simply generalized for an arbitrary constraint Φ_j, i.e., Y_l is replaced by Y_{l_j}. Procedure *change_limit* must be modified as follows. In the original procedure (Section 4.1), when one node changed its limit by σ, the second one would perform a change of the same value later. For the general constraint, however, the sign of the second change depends on the constants c_1 and c_2. For example, for the constraint $A + B \leq \delta$, a positive change in A's limit must be followed by a negative change in B's limit. In the procedure below, we also incorporate *policy4* discussed in Section 4.2.

Φ_j.**change_limit**(X,σ)
  if Φ_j.SAFE(X,σ) is FALSE *then*
    Send message to $N(Y)$ requesting it to perform
      Φ_j.**change_limit**(Y, $-sign(c_{1_j})\,sign(c_{2_j})\,\sigma$)
  else if Φ_j.LIMIT_BEYOND(X,σ) is TRUE *then*
    { fire up Φ_j.*policy*4(X, σ);
    abort this change }
  else
    { $X_{I_j} \leftarrow X_{I_j} + \sigma$;
    send message to $N(Y)$ requesting it to perform
      Φ_j.**accept_change**(Y, $-sign(c_{1_j})\,sign(c_{2_j})\,\sigma$) }.

For splitting the slack in the constraint $c_1A + c_2B \leq \delta$, $c_1, c_2 \neq 0$, we generalize Equation 4.1 as follows:

$$A_I = A + (\delta - c_1A - c_2B)\frac{k}{c_1} \tag{5.1}$$

$$B_I = \frac{B - (\delta - c_1A - c_2B)k}{c_2}$$

Equation 5.1 can be easily derived from $c_1A + c_2B \leq \delta$, noticing that the slack is equal to $\delta - c_1A - c_2B$.

To conclude this section, let us consider constraints of more than two variables. First, observe that if only two nodes are involved, nothing is different. For example, if we have the constraint $A + B + C \leq \delta$, and the items A, B are stored in one node, while C is stored at another, A and B can be treated as a single variable as far as the demarcation protocol is concerned. That is, it would be enough to have two limits: AB_I and C_I and follow the protocol.

If there are three or more nodes involved, then whatever node performs a safe operation must indicate what other node gets the amount. For unsafe operations, a node must select a particular node for its request to change limits. For instance, consider the inequality $A + B \leq C + \delta$. Say $N(A)$ wants to raise its limit, an unsafe operation. $N(A)$ has a choice: it may ask $N(B)$ to lower its limit, or $N(C)$ to increase its. Suppose that $N(C)$ is selected. If $N(C)$ does raise C_I, a message to perform *accept_change* is sent only to $N(A)$, so only it consumes the available slack. Now suppose that both $N(A)$ and $N(B)$ want to raise their limits concurrently, and send requests to $N(C)$. $N(C)$

should indicate to each the amount of their change. For example, if $N(C)$ raises C_I by 10, it may indicate to $N(A)$ that A_I can be raised by 6 and to $N(B)$ that B_I can be raised by 4. The generalizations we have illustrated here are straightforward and will not be discussed further.

6. Other Constraints

In the introduction we argued that inter-node constraints tend to be simple. So far we have studied one class of such simple constraints, arithmetic inequalities. In this section we illustrate how the demarcation protocol and its policies can be used to manage other types of simple constraints. The key idea is to convert these other constraints into arithmetic inequalities.

To illustrate, consider a referential constraint of the form

$$Exist(A,a) \Rightarrow Exist(B,b) \tag{6.1}$$

where A, B are data objects, a, b are nodes, and $Exist(A,a)$ is a predicate that is TRUE if object A is stored in node a and $Exist(B,b)$ is TRUE if B is at node b.

We can define the following function

$$f(X,x) = \begin{cases} 1 & \text{if } Exist(X,x) = \text{TRUE} \\ 0 & \text{if } Exist(X,x) = \text{FALSE} \end{cases} \tag{6.2}$$

where X stands for either A or B, and x corresponds to either node a or b. With this equation, the referential constraint becomes:

$$f(A,a) \leq f(B,b) \tag{6.3}$$

This constraint can now be enforced via the demarcation protocol, as long as we interpret arithmetic an operation on $f(X,x)$ to be the appropriate create or delete operation. That is, changing $f(A,a)$ from 1 to 0 means that A is deleted at a; changing it from 0 to 1 means that A is created. We must also define policies that implement the correct semantics for this case. One possibility is to define the following policies. (Policies 2 and 3 are not necessary in this case.)

Φ_j.**policy1**($f(X,x)$,δ,T_{code})
*** An invalid change to $f(X,x)$ has been attempted;
first change limit and then try transaction later ***
Φ_j.**change_limit**($f(X,x)$,δ);
resubmit later T_{code}.

Φ_j.**policy4**($f(X,x)$,δ)
*** An unsafe limit change has been attempted ***
Φ_j.**change_value**($f(X,x)$,δ,*null*)
resubmit Φ_j.**change_limit**($f(X,x)$,δ).

To illustrate, assume that A and B are stored in nodes a and b respectively. Thus, $f(A,a) = 1$, and $f(B,b) = 1$. Equation 6.3 must be enforced at all times. Therefore, the system establishes two limits $f(A,a)_l$ and $f(B,b)_l$, such that at all times $f(A,a) \leq f(A,a)_l, f(B,b) \geq f(B,b)_l$, and $f(A,a)_l \leq f(B,b)_l$. In this initial scenario, these limits can be $f(A,a)_l = f(B,b)_l = 1$. The safe operations are for a to decrease $f(A,a)_l$ and for b to increase $f(B,b)_l$.

Assume now that there is a transaction T that wants to delete B at b. The transaction will try to change $f(B,b)$ by -1 to 0. However, since the limit $f(B,b)_l = 1$, the transaction will be aborted and *policy1* fired. This policy will force the change of -1 on limit $f(B,b)_l$, invoking *change_limit*($f(B,b),-1$). However, this is not a safe operation for b, so a message to a will be sent requesting it to change $f(A,a)$ by -1 to 0. In turn, a will invoke *change_limit*($f(A,a),-1$). Since the desired new limit $f(A,a)_l = 0$ violates the constraint $1 = f(A,a) \leq f(A,a)_l$, *policy4* will be triggered (and the limit change aborted). This policy will force the change in $f(A,a)$ by -1 to 0, deleting A from a. The policy also resubmits the change in the limit $f(A,a)_l$ to 0. This time, the change will be successful. As a consequence of the change, a message will be sent to b allowing it to change $f(B,b)_l$ by -1. This makes $f(B,b)_l = 0$. When T is resubmitted and attempts the deletion of B again, it will find that the new limit allows it, and will proceed to delete B. Thus, the existential constraint is obeyed at all times. Note that if T is resubmitted early, before $f(B,b)_l$ has had a chance to change, unnecessary (but unharmful) messages will be triggered. One way to avoid this is to have the change in the limit $f(B,b)_l$ trigger the re-submission of the pending transaction.

While this may not be the most efficient way to enforce existential constraints, we believe it is very useful to have a uniform strategy for handling distributed constraints. The demarcation protocol provides such a strategy.

As stated in Section 1, approximate equality constraints of the form $|A - B - \delta| \leq \varepsilon$ can be implemented via the two constraints $A - B - \delta \leq \varepsilon$ and $-A + B + \delta \leq \varepsilon$. Each constraint can then be enforced by the demarcation protocol. We do not have space to describe the associated policies in detail but the key idea is as follows. Node a, holding A, will have an upper and a lower bound for A. Changes to A that leave it in this range can be done autonomously. If A gets ''too close'' to its upper limit, and thus ''too far'' from its lower one, two messages are sent to b: one stating that A's lower limit has been moved up (a safe operation for a), the second requesting that b up its lower limit (so that A upper limit can be moved accordingly). After these operations finish, A range will have moved up, so that A is no longer too close to its limit. To avoid unnecessary messages, the first message sent by a needs to be processed by b before the second one. What we have described here can be accomplished by writing the appropriate policies for the two constraints we have.

Finally, note that the demarcation protocol may also handle object copy constraints (see Section 1). The basic idea is to associate with each object a version number or counter that is incremented each time period. Constraints such as ''the copy of O at a site must be within two version of the master copy'' can be translated into arithmetic inequalities that can be managed via the demarcation protocol.

7. Conclusions

We have presented a strategy for enforcing arithmetic inequalities in distributed databases. Limits are defined for each of the participating variables; a node is free to update a variable as long as it stays within its bounds. The demarcation protocol is used to change the limits in a dynamic fashion. We also showed how this strategy can be used for other types of constraints, such a referential constraints and approximate equalities.

Intuitively, we may view a system that uses the demarcation protocol as a ''spring.'' When a constraint has a lot of slack, limits will not be tight, and many transactions will be able to perform their updates locally. This corresponds to a loose spring. As the slack

is reduced, the spring is compressed, and more and more transactions will hit against the limit. The transactions will be more expensive to run, as they will require negotiations with the other node to change limits. When there is no slack e.g., $A = A_1$ and $B = B_1$, the spring is compressed the most. Even at this point, the system may be more efficient than a conventional one (which requires two phase commit for every transaction) because transactions that move a variable away from its limit may be done locally.

There are two ways to implement the demarcation protocol in a database system. One is to use an existing system, and provide the user with a library of procedures (e.g., *change_value*), sample policies (e.g., for splitting slack), and definitions (e.g., for limit variables). The user's code could then call these procedures to update values or change limits. The disadvantage of this approach is that a user could circumvent the rules, e.g., by updating a constraint variable directly and not through *change_value*. The other option is to incorporate the procedures into the system itself. The database administrator (or possibly an authorized user) would define the constraints and policies and give them to the system. The variables involved would be tagged. When a transaction updated one of these variables, it would trigger the necessary procedures. With either one of these approaches, there are clearly many issues that still need to be resolved, such as language used for defining the constraints, load control strategies, dealing with contradictory constraints, and so on.

Acknowledgements

We would like to thank Luis Cova and Ken Salem for helpful discussions, and Naftaly Minsky and Patrick O'Neil for pointing out some useful references.

8. References

[ABG90] R. Alonso, D. Barbará, and H. Garcia-Molina, "Data Caching Issues in an Information Retrieval System," *ACM Transactions on Database Systems,* Vol. 15, No. 3, September 1990.

[CR82] O.S.F. Carvalho and G. Roucariol, "On the Distribution of an Assertion," *Proceedings of the ACM-SIGOPS Symposium on Principles of Distributed Computing*. Ottawa, 1982.

[DAT83] C.J. Date, ''An Introduction to Database Systems,'' Volume 2, Addison-Wesley.

[DAV82] S. B. Davidson, ''An Optimistic Protocol for Partitioned Distributed Database Systems,'' Ph.D. Dissertation, Princeton University. October 1982.

[DE89] W. Du, and A. Elmagarmid, ''Quasi-Serializability: A Correctness Criterion for Global Concurrency Control in InterBase,'' *Proceedings of the 15th Conference on Very Large Data Bases,* Amsterdam, Aug. 1989.

[FGL82] J.M. Fischer, N.D. Griffeth, and N.A. Lynch, ''Global States of a Distributed System,'' *IEEE Transactions on Software Engineering,* SE-8, 3. May 1982.

[FZ89] M. F. Fernández and S. B. Zdonik, ''Transaction Groups: A Model for Controlling Cooperative Work,'' *Proceedings of the Third International Workshop on Persistent Object Systems.* Queensland, Australia, January 1989.

[GAR83] H. Garcia-Molina, ''Using Semantic Knowledge for Transaction Processing in a Distributed Database,'' *ACM Transactions on Database Systems,* Vol.8, No. 2, June 1983.

[HS80] M.M. Hammer and D. W. Shipman, ''The Reliability Mechanisms of SDD-1: A system for Distributed Databases,'' Computer Corporation of America Technical Report CCA-80-04, January 1980.

[KS88] H. F. Korth, and G.D. Speegle, ''Formal Model of Correctness Without Serializability,'' *Proceedings of the ACM SIGMOD International Conference on Management of Data,* Chicago, June 1988.

[KuS88] A. Kumar, and M. Stonebraker, ''Semantics Based Transaction Management Techniques for Replicated Data,'' *Proceedings of the ACM SIGMOD 1988 International Conference on Management of Data,* Chicago, 1988.

[LBS86] N.A. Lynch, B. Blaustein, and M. Siegel, ''Correctness Conditions for Highly Available Replicated Data,'' *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Systems,* Calgary, August 1986.

[ONE86] P. O'Neil, ''The Escrow Transactional Method,'' *ACM Transactions on Database Systems,* Vol. 11, No. 4, December 1986.

[SS90] N. Soparkar, A. Silberschatz, ''Data-Value Partitioning and Virtual Messages,'' *Proceedings of the Conference on Principles of Database Systems*, 1990.