UPDATE PROPAGATION IN BAKUNIN DATA NETWORKS

Boris Kogan
Hector Garcia-Molina

CS-TR-091-87

May 1987

UPDATE PROPAGATION IN BAKUNIN DATA NETWORKS[†]

Boris Kogan
Hector Garcia-Molina

Department of Computer Science
Princeton University
Princeton, N.J. 08544

Abstract

In a Bakunin network data are replicated at all nodes in order to achieve very high data availability. Nodes operate autonomously, executing transactions even when they are cut off from the rest of the system. This means that transactions may read stale data. In spite of this, serializability can be guaranteed by placing restrictions on the types of transactions that a node can execute. These restrictions take the form of an acyclic Read-Access Graph. In addition, a special update propagation protocol is used to ensure that all nodes see data updates in the same order. In this paper we present several such protocols. The protocols take advantage of the particular structure of the read-access graph to expedite propagation. We also define the notion of virtual serializability. It is a weaker form of serializability that allows speedier propagation.

1. Introduction.

High data availability in the face of failures can be achieved through data replication and distribution. However, conventional concurrency control mechanisms [Bern81] halt transaction processing when certain communications failures occur. Thus, if a node or group of nodes is cut off from the rest of the system, they will be unable to access the data they have. This clearly hurts availability. In some applications it is important *not* to halt transaction processing when communication failures occur, and hence researchers have been studying more flexible mechanisms.

Until recently, a clean partition model has been used to study such high availability mechanisms [BlKa85], [Davi84], [Giff83], [Park81], [SkWr84]. This model assumes that at any given time it is possible to tell whether the network is partitioned and if so, how the nodes are grouped. Although arguments can be presented in favor of this model, one must acknowledge that it does not always correctly depict real systems. In reality, it may be very difficult to detect when partitions occur and when they are repaired. This was the motivation for several researchers to take a different approach to the problem, namely to make no assumptions about detectability of partitions and even communication failures in general [Davi85], [Sari85]. We call this new failure model the ***dynamic failure model***.

Most of the work that has been done so far on the dynamic failure model suffers from one common drawback: it does not guarantee serializability. Instead, it introduces some weak correctness criteria. In the present paper we suggest a scheme that provides a high degree of availability and, at the same time, preserves serializability as the correctness criterion. In addition, we introduce a new correctness criterion called ***virtual serializability***. With it we can improve availability still further. Virtual

[†] This work has been supported by NSF Grants DMC-8351616 and DMC-8505194, New Jersey Governor's Commission on Science and Technology Contract 85-990660-6, and grants from DEC, IBM, NCR, and Concurrent Computer corporations.

serializability is a little weaker than strict serializability, but substantially stronger than other criteria suggested for high availability databases.

The proposed scheme is characterized by highly autonomous behavior of individual nodes with respect to transaction processing. No attempts at explicit inter-node concurrency control are made. Instead, correctness of execution is enforced by a combination of certain data access restrictions and special update propagation protocols. Thus, on the one hand, individual nodes are endowed with absolute freedom (in scheduling their transactions), and on the other hand, overall harmony (of execution correctness) is nevertheless achieved. Because of this property, we have termed such systems *Bakunin data networks* , after the 19th century Russian political theorist of anarchist convictions.

2. The Database Model.

A distributed database system is viewed as a collection of nodes interconnected by some communication network, which store a set of data items. We assume that the database is fully replicated, *i.e.*, each node has a complete copy of it. Nodes are fail-stop [ScSc83]. Links can fail at any time, and we make no assumptions concerning the detectability of these failures.

In our scheme, the entire database is logically divided into k non-overlapping subsets of data items, called *fragments*, denoted F_1, F_2, ..., F_k. Each node has a copy of every fragment. Each fragment F_i is controlled by a unique node $N(F_i)$ responsible for updating items in the fragment. We call the process at $N(F_i)$ that controls F_i the *agent* of F_i, or $A(F_i)$. For simplicity we assume that each node (and agent) controls only one fragment. (This last assumption can easily be relaxed.)

A transaction is taken to be a sequence of any number of atomic read or write actions [Eswa76]. Each transaction executes at a single node (we say it is *local* to that node) and can only update the fragment whose agent resides at that node, but it can, in general, read copies of other fragments. (Since there are local copies of all fragments, these reads are performed at the updating node.) Subsequently, all resulting updates are propagated throughout the system. Let T_i be an arbitrary transaction, then $U(T_i)$ denotes the list of all updates generated by T_i.

We assume the existence of a reliable multicast mechanism which ensures that all messages are eventually delivered to their destinations and no messages are delivered out of order. Each node continues processing transactions as long as it stays operational, regardless of the status of the communication network. Each node has a local concurrency control mechanism used to schedule transactions within that node. However, there is no (explicit) global concurrency control. In particular, read and write accesses to data never have to be coordinated with other nodes. We will also assume that all nodes control some fragment. (Nodes that do not can be considered as controlling an empty fragment.)

In order to specify how transactions originating at different nodes are restricted in which fragments they are allowed to read, other than the one controlled by their own node, we will use the following graph formalism.

Definition 2.1. The *read-access graph* (*RAG*) is a directed graph $G = (V, E)$, where $V = \{F_1, F_2, \ldots, F_n\}$ and $E = \{(F_i, F_j) : i \neq j$ and a transaction T that is initiated by $A(F_i)$ can read a data object contained in $F_j\}$.

Figure 2.1 shows an example of a read-access graph with three fragments: F_1, F_2 and F_3. Suppose that $A(F_i)$ resides at node i, for $i = 1, 2, 3$. Then transactions that run at node 1 can read items in fragments F_1 and F_2, those running at node 2 can read items in fragments F_2 and F_3, and those running at node 3 can read items in fragments F_1 and F_3.

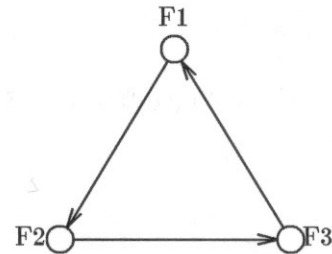


Figure 2.1

Since we assumed a one-to-one correspondence between fragments and computer nodes, we shall sometimes refer to the vertices of *RAG*s as computer nodes rather than fragments, whenever it is more convenient.

2.1. The Generic Update Propagation Protocol.

Let all the nodes of the *RAG* be numbered 1 through n. Let $R(i)$ be the set of all nodes from which node i receives update messages ($R: \{1, \ldots, n\} \rightarrow 2^{\{1, \ldots, n\}}$). Let $S(i)$ be the set of all nodes to which node i sends update messages ($S: \{1, \ldots, n\} \rightarrow 2^{\{1, \ldots, n\}}$). R and S are called *propagation functions*. Functions R and S are mutually redundant since one is the inverse of the other. However, we have chosen to introduce both of them in order to facilitate the forthcoming discussions.

Every node i will maintain two lists, which are initially empty. They are $UPDATES(i)$ and $OUT(i)$. The protocol consists of three concurrent procedures.

(1) *(Permanently in execution at node i, $1 \leq i \leq n$.)* For every local tranaction T_j, append $U(T_j)$ to $UPDATES(i)$. The order in which different transactions' updates are appended to the list is determined by the local serialization order.

(2) *(Executed at node i, $1 \leq i \leq n$, upon receipt of $OUT(j)$, $j \in R(i)$.)* Append $OUT(j)$ to $UPDATES(i)$ and atomically install all updates from $OUT(j)$ in the local copy of the database. Make sure that for every local transaction T_j which is serialized before these updates (and none other) $U(T_j)$ is appended to $UPDATES(i)$ before $OUT(j)$.

(3) *(Iterated with arbitrary intervals at node i, $1 \leq i \leq n$.)* Atomically copy $UPDATES(i)$ to $OUT(i)$. Send $OUT(i)$ to all nodes in $S(i)$. Reinitialize $UPDATES(i)$ to empty.

Note: Since $UPDATES(i)$ is accessed concurrently by several procedures, it is essential that a locking mechanism be used.

Propagation functions establish a *logical* order of update propagation. They have nothing to do with physical routing of messages. Thus, $OUT(i)$ is addressed to nodes in $S(i)$, but how it gets there is not important as long as the nodes not in $S(i)$ through which the message may be routed do not attempt to process it.

Note that, strictly speaking, updates in $U(T_s)$, where T_s is a transaction updating fragment F_i, are needed only at those nodes that control fragments F_j such that (F_j, F_i) is an edge in the *RAG*, for only those nodes will ever read the data updated[†]. We will ignore any redundancies for simplicity's sake. However, it is a straightforward matter to introduce an optimization provision in the protocol that would prune out update lists of those updates that are known not to be needed by any node down the chain.

On the abstract level, a distributed database system can be described by a combination of its *RAG* and propagation functions. In subsequent sections we shall study different such combinations and their properties.

3. Availability and Related Notions.

The main motivation for studying Bakunin data networks is the very high level of availability they can offer. The ability of every individual node to schedule and execute transactions without having to "consult" any other nodes implies that transactions continue being processed at that node as long as the node stays operational. No communication delays or failures can affect this ability.

This property of Bakunin networks does not come for free, of course. In order to insure that the given correctness criteria are not violated by the anarchy in scheduling, we have to curtail somewhat the amount of data available for access at each node. Besides, in some cases it may be necessary to choose the propagation functions in such a way as to restrict the routes of update propagation, which implies delayed delivery of updates at some nodes, or less update availability, so to speak.

The word availability was used three times in the two preceding paragraphs, and each time it had a somewhat different meaning. To clarify the matters, we discuss here briefly the notions involved.

Static Availability.

We use the term static availability to refer to the ability of computer nodes (users) to access certain portions of the entire body of data as determined by the system design (e.g., because of security considerations). The restrictions of static availability do not bear any temporal character. They are set once and for all, for

[†] This is true only for systems in which all the agents are stationary, *i.e.*, fixed at their respective nodes. Such systems are the sole subject of the present paper. However, see [GaKo87] for moving agents.

every node (user). To say that a certain fragment is statically available at a certain node (to a certain user) is to imply that this fragment can be accessed at the node at all times except possibly when accessing it may jeopardize data consistency (e.g., during communication failures). On the other hand, if a fragment is not statically available to a node (user), it cannot be accessed by such even when it is safe from the viewpoint of data consistency. *RAG*s serve as an example of how static availability can be curtailed.

Dynamic Availability.

This term refers to the ability of nodes (users) to access, over time, the portions of data that are statically available to them. A loss of dynamic availability (in replicated systems) can occur to prevent a violation of data consistency that may typically be experienced due to communication failures such as partitions.

Prompt Delivery of Updates (or Update Availability).

The last of the three related notions has to do with the extent to which the system utilizes currently functional communication links in order to propagate updates as quickly as possible. High levels of static and dynamic availability are of no value if updates produced at one node are not delivered to other nodes or delivered with big time delays. Consider, for example, a distributed system with full static and dynamic availability at every node, *i.e.*, all data can be accessed anywhere in the system at any time (barring node failures, of course). Suppose further that updates produce by transactions running at one node never reach other nodes. Clearly, such collection of nodes can hardly even be called a system.

All three notions are closely interrelated. Most of the known techniques for dealing with network partitions do not make any restrictions to static availability (aside from those required by data security considerations). As a result, either dynamic availability (as in [Giff83], [SkWr84]) or correctness (as in [BlKa85], [Davi84], [Park81], [Sari85]) suffers. Bakunin data networks, in contrast to other approaches, compromises neither dynamic availability nor correctness in the least. This is achieved at the price of restricted static availability and, in some cases, delayed delivery of updates. It is not one of the subjects of this paper to discuss to what extent a loss of static availability can be detrimental (except for presenting a brief example in the following section), but in [GaKo87] it is argued that for many systems full static availability is not necessary.

3.1. How to Evaluate Protocols.

In Section 2.1 we presented a general update propagation framework from which an entire family of protocols can be derived by choosing different propagation functions R and S. We are going to use two basic criteria for selecting R and S. First, they must be such as to ensure the required correctness properties. Second, they must provide, as much as possible, prompt delivery of updates. In this section we are going to discuss the second criterion, which will be also referred to as performance of a protocol.

Clearly, the basis for the evaluation of performance should be how fast updates are delivered to the node which needs them. Of course, this largely depends on the topology of the communication network, its current load and status, the bandwidth of its links, etc., but all factors being equal different protocols (different propagation functions) will perform differently.

Consider two nodes in the distributed system, i and j. Suppose, there is an edge from i to j in the *RAG*. That means that transactions executing at node i can read data from the fragment controlled by node j. In that case, it is desirable that updates generated at j be delivered to i as promptly as possible in order for the transactions at i to be able to read up-to-date information. Therefore, a protocol that allows node j to ship its updates directly to i for installation in the local copy is the most favorable to node i. Such a protocol would have $i \in S(j)$ (and $j \in R(i)$).

Definition 3.1. Let $G = (V, E)$ be a *RAG* with corresponding propagation functions R and S. Then $G_p = (V, E_p)$ is called a ***propagation graph*** if $E_p = \{(i, j): i, j \in V \text{ and } j \in S(i)\ (i \in R(j))\}$.

Definition 3.2. Let G and G_p be as above. A path from node i to node j in G_p is called ***characteristic*** if it is a shortest path from i to j in G_p and $(i, j) \in E$.

A propagation graph describes the order in which updates are propagated in the network and installed in the local copies of different

nodes. The length of a characteristic path indicates how far an update has to travel (in the logical sense) before it reaches a node where it is needed. Intuitively, a "good" protocol would result in a propagation graph with as short characteristic paths as possible for as many pairs of nodes as possible. Note that if two nodes, say i and j, are not connected by an edge in the *RAG*, then we are not really interested in the length of the shortest paths from i to j or from j to i, because neither of the nodes processes transactions that read data from the fragment controlled by the other.

Ideally, we would like to have a protocol that results in a propagation graph of which the corresponding *RAG* is a subgraph. Then all characteristic paths would be of length 1. This situation is achieved when $i \in S(j)$, for every $(i, j) \in E$. However, as we shall see in the subsequent sections, it is not always possible to have propagation functions that guarantee both good performance and adherence to the required correctness criteria.

4. Retaining Serializability as Correctness Criterion.

In this section we study systems (*i.e.*, *RAG* — propagation functions combinations) that guarantee serializability of transaction execution. We begin with a fairly simple observation that the *RAG* must be acyclic if we want to guarantee serializability at all times, even during failures. To show the need for this, consider the example of Figure 2.1. Suppose that a partition occurs and leaves each node in complete isolation. If the nodes continue processing transactions — and that is what we are after — the global schedule may not be serializable. For instance, if transaction T_1 runs at node 1, reads $b \in F_2$, and writes $a \in F_1$, transaction T_2 runs at node 2, reading $c \in F_3$ and writing b, and transaction T_3 runs at node 3, reading a and writing c, the results cannot be integrated to match any serial schedule. Note that the above holds true for any possible propagation functions.

Consequently, all the systems that we shall study in this section will be characterized by acyclic *RAG*s only.

Restricting *RAG*s to be acyclic clearly reduces the static availability of the system. But this is the price we pay for maintaining high dynamic availability. A valid question that can arise at this point is whether such a restriction of static availability is justifiable. We argue that in many applications it is.

Consider, for example, an airline reservation application. The database contains information on flight schedules, customer reservations, and seat assignments. Copies are to be placed at several computers, including machines at the airports where this airline operates. High dynamic availability is required for this application. For example, we would like to assign passengers to their seats at the airport even if that machine is cut off from the rest of the system. On the other hand, in this example, it is not necessary to run all types of transactions at any node. For example, it is unlikely that a flight schedule will be changed at an airport. (The actual departure time may be changed, but this is another matter.) Flight schedules are probably handled at a central airline office, and it is this machine that needs to be able to execute schedule changes. Similarly, the central office most likely is not expected to be responsible for seat assignment on any given flight. Thus, one can see that full static availability is not important here.

Each node has a copy of every fragment. There are six fragments: the flight schedules (F), the west coast reservations (R_w), the east coast reservations (R_e), and the seat assignments at airports A, B, and C (S_A, S_B, S_C). (Our airline only flies out of three airports. We also assume that the reservations data are split into two parts).

Each fragment G_i is controlled by a unique node $N(G_i)$ responsible for updating items in the fragment. For simplicity we assume that each node controls only one fragment. Thus, our system will have six nodes: one at the airline headquarters where the schedules are changed, two computers for handling reservations, and one computer at each of the three airports for seat assignments.

The read patterns of the transactions are represented by the *RAG* in Figure 4.1. Transactions that change schedules do not need to read data outside this fragment, so node F has no outgoing arcs. To make a reservation, a transaction must be aware of the schedules, so there are arcs $R_e \rightarrow F$ and $R_w \rightarrow F$. Finally, transactions that give a customer a seat at the airport need to see the schedule and the reservations (a passenger without a reservation does not get a

seat); hence $S_A \rightarrow R_e$, $S_A \rightarrow R_w$, $S_A \rightarrow F$, and so on.

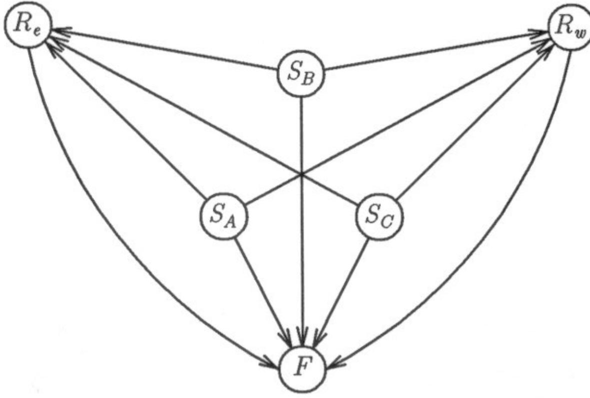


Figure 4.1

Note that the *RAG* is acyclic. Thus in this example we were able to use the inherent structure of update and read operations to painlessly incorporate the required static availability restrictions in the scheme.

4.1. Acyclic *RAG*s: The General Case.

In this subsection we aim at finding the propagation functions for systems characterized by acyclic *RAG*s such that serializability can be guaranteed for all possible transaction schedules. Let $G = (V, E)$ be an acyclic *RAG*. Suppose that $ord\colon V \rightarrow \{1, ..., n\}$ is a topological order on the vertices of G, *i.e.*, for any two vertices i and j, if (i, j) is an edge in G, $ord(i) < ord(j)$. (A topological order must exist if G is acyclic.) Then define functions R and S as follows:

$$R(i) = \begin{cases} \{ord^{-1}(ord(i)+1)\} & \text{if } 1 \le ord(i) < n \\ \varnothing & \text{if } ord(i) = n \end{cases} \qquad (4.1a)$$

$$S(i) = \begin{cases} \{ord^{-1}(ord(i)-1)\} & \text{if } 1 < ord(i) \le n \\ \varnothing & \text{if } ord(i) = 1 \end{cases} \qquad (4.1b)$$

For the purpose of notational convenience, we assume, without loss of generality, that the original numbering of nodes in G corresponds to a topological order. Then we can rewrite the above definitions in a more digestable form:

$$R(i) = \begin{cases} \{i+1\} & \text{if } 1 \le i < n \\ \varnothing & \text{if } i = n \end{cases} \qquad (4.2a)$$

$$S(i) = \begin{cases} \{i-1\} & \text{if } 1 < i \le n \\ \varnothing & \text{if } i = 1 \end{cases} \qquad (4.2b)$$

Definition 4.1. Let $l \in V$ be such that $ord(l) = 1$ (with our simplifying assumption, $l = 1$). Then for a pair of transactions T_i and T_j, $T_i < T_j$ if $U(T_i)$ is installed at node l before $U(T_j)$.

Lemma 4.1. If for every edge (T_i, T_j) in the serialization graph, $T_i < T_j$, then the serialization graph is acyclic.

Proof. Suppose not. Let $C = (T_1, ..., T_k, T_1)$ be a cycle in the serialization graph. It is easy to see that relation $<$ is transitive. Therefore we must have $T_1 < T_1$, which is clearly impossible. □

Lemma 4.2. Let R and S be the propagtion functions defined in (4.2a) and (4.2b). Then for every edge (T_i, T_j) in the serialization graph $T_i < T_j$.

Proof. Consider any pair of transactions T_i and T_j such that (T_i, T_j) is in the edge set of the serialization graph.

Case 1. Transactions T_i and T_j are local to the same node, say node q, $1 \le q \le n$. Since (T_i, T_j) is an edge in the graph, T_i must be serialized (locally) before T_j. Hence, $U(T_i)$ is effectively installed in the local copy of the database before $U(T_j)$. Moreover, they are appended to *UPDATES*(q) in the same order, and thus, will be installed in that order in the copies at nodes $q-1, ..., 1$.

Case 2. Transaction T_i is local to node q, transaction T_j is local to node r, and $1 \le q < r \le n$. This means that T_j overwrites a value read by T_i. Therefore, T_i is serialized at node q before $U(T_j)$. Consequently, $U(T_i)$ is appended to *UPDATES*(q) before $U(T_j)$. This, in turn, implies that nodes $q-1, ..., 1$ install $U(T_i)$ before $U(T_j)$.

Case 3. As in Case 3, except $r < q$. Here, T_j reads a value written by T_i. Therefore, $U(T_i)$ is serialized at node r before T_j, $U(T_i)$ is appended to *UPDATES*(r) before $U(T_j)$, and finally, $U(T_i)$ is installed at nodes $r-1, ..., 1$ before $U(T_j)$.

Cases 1, 2, and 3 exhaust all possibilities. Thus, $T_i < T_j$. □

Lemmas 4.1 and 4.2 together establish the following theorem.

Theorem 4.1. Let R and S be the propagation functions defined in (4.2a) and (4.2b). Then the schedule for any execution characterized by an acyclic *RAG* is serializable.

Note that acyclicity of G was not used explicitly in the proof of the lemmas. This property is essential, however, in order to guarantee the existence of a topological numbering.

4.2. A Special Case: tf-*RAGs*.

The topological protocol of Section 4.1 may be too restrictive in some cases. In particular, if the *RAG* is a tree, possibly with forward edges, then a protocol that propagates updates up the tree can be constructed. It performs better than the topological protocol and still guarantees serializability.

Definition 4.2. Let $G = (V, E)$ be a directed graph. If G has a depth-first search (DFS) tree (forest) that contains tree and forward edges only, then G is called a *tf-graph*.

Let G be the given *RAG*, which is a tf-graph. Let D be a DFS forest of G with no cross or back edges. Without loss of generality, we can assume that D is a tree, for if it is not, it must consist of disconnected trees, in which case each of them can be treated independently. Then:

$$R(i) = \{j: j \text{ is a child of } i \text{ in } D\} \quad (4.3a)$$

$$S(i) = \{j: j \text{ is the parent of } i \text{ in } D\} \quad (4.3b)$$

Definition 4.3. Let r be the root of D. Then $T_i <_{tf} T_j$ if $U(T_i)$ is installed at node r before $U(T_j)$.

Lemma 4.3. If for every edge (T_i, T_j) in the serialization graph $T_i <_{tf} T_j$, then the serialization graph is acyclic.

Proof. Similar to Lemma 4.1. □

Lemma 4.4. Let G be a tf-*RAG*. Let R and S be the propagation functions defined in (4.3a) and (4.3b). Let (T_i, T_j) be an edge in the global serilaization graph. Then $T_i <_{tf} T_j$.

Proof.

Let T_i be local to node v and T_j to node w.

Case 1. Transactions T_i and T_j are local to the same node ($v = w$). Since (T_i, T_j) is an edge in the serialization graph, T_i must be serialized before T_j at v. Therefore, $U(T_i)$ is effectively installed in the local copy before $U(T_j)$. The same order of installation is preserved at all nodes that are ancestors of v in D, including r.

Case 2. (v, w) is an edge in D (either tree or forward). Clearly, at node v, $U(T_i)$ is installed before $U(T_j)$. The same order of installation must hold for any ancestor of v because $U(T_i)$ is appended to $UPDATES(v)$ before $U(T_j)$, and the only way that $U(T_j)$ can get to v's ancestors is through v itself (propagation occurs along tree edges only). Thus at node r $U(T_i)$ is installed before $U(T_j)$.

Case 3. (w, v) is an edge in D. Similar to Case 2.

Cases 1, 2, and 3 exhaust all possibilities. Thus $T_i <_{tf} T_j$. □

Theorem 4.2. Let $G = (V, E)$ be a tf-*RAG*. Let R and S be the propagation functions defined in (4.3a) and (4.3b). Then any execution characterized by G, R and S is serializable.

Proof. Follows trivially from Lemmas 4.4 and 4.3. □

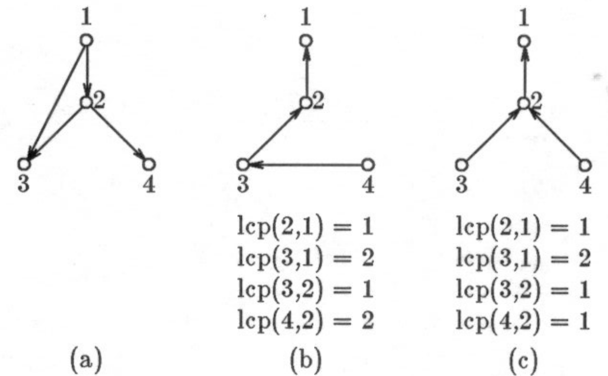


Figure 4.2. (a) *RAG*; (b) propagation graph for topological protocol; (c) propagation graph for tf-protocol.

To illustrate the advantage of the protocol presented in this section over the topological protocol, consider the example of Figure 4.2. We are interested in the length of characteristic paths (*lcp*) between four pairs of nodes (one for very edge in the *RAG* of Figure 4.2(a)). From Figures 4.2(b) and 4.2(c) we see that the tf-propagation results in all but one of these paths being of length 1, while the topological propagation yields two characteristic paths of length greater than 1. It is not hard to see that, in general, the tf-propagation can always do at least as well as the topological propagation (for tf-*RAGs*).

4.3. A Special Case: Loopless *RAG*s.

For some *RAG*s we can devise an even more efficient protocol than the tf-graph one. These *RAG*s are ones without any "loops," i.e., without undirected cycles. In this case we can propagate updates along all RAG edges, obtaining the best possible performance. Although the result can be expressed simply, the proof is substantially harder than the others we have presented.

Definition 4.4. [GaKo87] Let $G = (V, E)$ be our *RAG*. The ***local serialization graph*** of node $v \in V$ is a directed graph whose vertex set contains all transactions local to v or to any node w such that $(v, w) \in E$. Its edges are computed according to the following rules:

(i) For any two transactions local to v, it is determined whether there is a directed edge between them according to the standard dependency rules for centralized databases [Eswa79].

(ii) Let transaction T_i be local to node v, transaction T_j local to some node w, and $(v, w) \in E$. Then if there is a data item d in the fragment controlled by w that is read by T_i and updated by T_j, and the update to d produced by T_j is installed at node v before T_i reads d, put in an edge (T_j, T_i); if the update is installed after T_i reads d, put in an edge (T_i, T_j).

(iii) For a pair of transactions T_i and T_j local to the same node w ($(v, w) \in E$), put in an edge (T_i, T_j) if T_i is installed by v before T_j, an edge (T_j, T_i) otherwise.

(iv) There is no edge between two transactions local to different nodes both of which are distinct from v.

Definition 4.5. A ***legal*** transaction schedule is the one for which all local serialization graphs are acyclic.

Definitions 4.4 and 4.5 formalize the workings of the local concurrency control mechanisms at every computer node. Note that rule (ii) of Definition 4.4 corresponds to the requirement that updates from nonlocal transactions be installed atomically and in the order in which they arrive.

Definition 4.6. Let $G = (V, E)$ be a (global) serialization graph. Let (T_i, T_j) be an edge in it such that T_i and T_j are local to (distinct) nodes v and w of the corresponding *RAG*, respectively. Then we say that edge (T_i, T_j) ***spans*** edge (v, w) or (w, v), whichever is present in the *RAG*.[†]

Definition 4.7. Let $G = (V, E)$ be a (global) serialization graph and let C be a cycle in it. Let G_C be the subgraph of the corresponding *RAG* that contains all the edges that are spanned by edges on C. Then C is said to be ***based on*** subgraph G_C.

Definition 4.8. Let $G = (V, E)$ be a directed graph. A subgraph L of G is called a ***loop*** if, when the directions on its edges are ignored, it forms a simple cycle.

Theorem 4.3. A loopless *RAG* guarantees a serializable transaction execution schedule with any choice of functions R and S in the update propagation protocol.

Proof. As before $G = (V, E)$ is the *RAG*. Let C be a cycle in the (global) serialization graph. Let $G_C = (V_C, E_C)$ be the subgraph of G on which C is based. We will show, by induction on k, the cardinality of V_C, that C cannot exist. For $k = 1$ it is obviously true. Suppose it is also true for all $k < l$. Now let $k = l$. Since G is loopless, so is G_C, and there must be some node $v \in V_C$ which is the head or the tail of only one edge. The general strategy of the proof will be to arrive at a contradiction by transforming C into a new cycle which still corresponds to a legal schedule but is based on a smaller subgraph of the *RAG* (with the cardinality of the vertex set equal to $l - 1$). Consider the following two cases.

Case 1. v is the tail of an edge. Let w be the head of this edge, *i. e.*, $(v, w) \in E_C$. Let u be the number of paths on cycle C that consist exclusively of transactions local to v. (In Figure 4.3, $u = 3$. These paths are represented by solid lines. For our purposes, a single node can be considered a path.) For $j = 1, \ldots, u$, let T_{l_j} be the vertex on C that immediately precedes the j-th path, and T_{m_j}, the vertex that immediately follows it. Since w is the only node in S that shares an edge with v, one can conclude that transactions T_{l_j} and T_{m_j}, for all $j = 1, \ldots, u$, are local to w. Finally let P denote the collection of (disjoint paths) on C that consist of transactions local to nodes other than v.

[†] Note that one of these two edges must be in the *RAG* for edge (T_i, T_j) to be present in G.

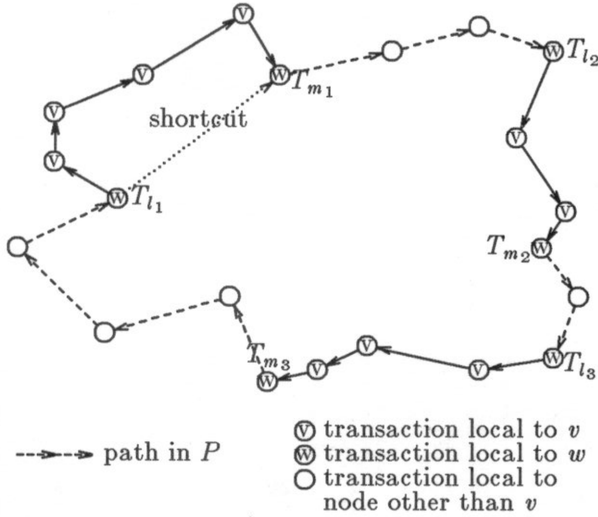


Figure 4.3. Cycle C ($u = 3$).

Let L_w denote the local serialization graph of node w. We will show, by induction on u, that P cannot be totally contained in L_w. First, let $u = 1$. In this case P consists of just one path, a path from T_{m_1} to T_{l_1}. Suppose that P is totally contained in L_w. This implies that T_{m_1} must have been executed at node w before T_{l_1}. But since the order of installation of these transactions' updates at v must be the same, there is an edge (T_{m_1}, T_{l_1}) in L_v, the local serialization graph of node v. There is also a path from T_{l_1} to T_{m_1} in L_v. Hence, L_v contains a cycle, which is impossible for a legal schedule. Thus, for $u = 1$, P cannot be totally contained in L_w.

Proceeding by induction, suppose that for some $u - 1$, P cannot be totally contained in L_w. Assume C has u paths on it consisting exclusively of transactions local to v. Let X be one such path (from T_{l_1} to T_{m_1}). Let us introduce a new data item d into the fragment controlled by w. This new item will be read only by transaction T_{l_1} and overwritten only by T_{m_1} (in that order). This creates an edge (T_{l_1}, T_{m_1}) in the (global) serialization graph as well as in L_w (if it was not there before). Let us call the technique of artificially creating a precedence edge *shortcutting*. Note that when we shortcut from T_{l_1} to T_{m_1}, no cycle is created in L_w (otherwise there would have been a cycle in the (global) serialization graph with P of size 1, which was shown not to be possible). Thus the modified schedule is still legal. However, we end up with a cycle in the (global) serialization graph with P of size $u - 1$, which contradicts the induction hypothesis. Thus, we conclude that C cannot exist.

Now we are forced to assume that if a cycle C does exist, P is not totally contained in L_w. This implies that if we shortcut from T_{l_j} to T_{m_j}, for all j, the result will be a cycle in the serialization graph that is based on a subgraph of G with $l - 1$ nodes. However, L_w will remain acyclic, and hence, the schedule will still be legal. (Note that other local serialization graphs are unaffected by shortcutting since the newly introduced data items can be read and overwritten only by transactions local to w.) This contradicts the induction hypothesis (induction on k). Thus the serialization graph must be acyclic.

Case 2. Assume that v is the head of an edge ($(w, v) \in E$). Let C be a cycle in the serialization graph, and let P be as before. Since in this case L_w contains all transactions local to v, P cannot be totally contained in L_w (otherwise P plus transactions local to v would form a cycle within L_w, rendering the corresponding schedule illegal). Thus, we can shortcut from T_{l_j} to T_{m_j}, for all j, without violating the legality of the schedule and, at the same time, creating a cycle based on a subgraph of G with $l - 1$ nodes, which is in contradiction with the induction hypothesis. □

Since loopless *RAG*s permit the choice of any propagation functions, we can always achieve the perfect propagation graph. Thus, we can, clearly, do better than with either topological or tf-propagation.

4.4. More on the General Case.

In this subsection we combine the results of Sections 4.1 and 4.2 to derive a more powerful propagation protocol for the general case. Let $G = (V, E)$ be an acyclic *RAG*. Let $L = (V_L, E_L)$ be the subgraph of G consisting of all the edges that lie on a loop. Let $L_1 = (V_{L_1}, E_{L_1})$, ..., $L_m = (V_{L_m}, E_{L_m})$ be the weakly connected components of L [Chri75]. For each $i = 1, ..., m$, let $\| V_{L_i} \| = k_i$, and let ord_i: $V_{L_i} \rightarrow \{1, 2, ..., k_i\}$ be a topological order on the vertices of L_i. Then define the propagation functions as follows:

$$R_1(v) = \{ord_i^{-1}(ord_i(v)+1)\} \bigcup \{w\colon w \in V - V_L \text{ and } (v,w) \in E\}$$

$$R_2(v) = \{w\colon w \in V - V_L \text{ and } (v,w) \in E\}$$

$$R_3(v) = \{w\colon w \in V \text{ and } (v,w) \in E\}$$

$$R(v) = \begin{cases} R_1(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v) < k_i \\ R_2(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v) = k_i \\ R_3(v) & \text{if } v \in V - V_L \end{cases} \quad (4.4a)$$

$$S_1(v) = \{ord_i^{-1}(ord_i(v)-1)\} \bigcup \{w\colon w \in V - V_L \text{ and } (v,w) \in E\}$$

$$S_2(v) = \{w\colon w \in V - V_L \text{ and } (w,v) \in E\}$$

$$S_3(v) = \{w\colon w \in V \text{ and } (w,v) \in E\}$$

$$S(v) = \begin{cases} S_1(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v) > 1 \\ S_2(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v) = 1 \\ S_3(v) & \text{if } v \in V - V_L \end{cases} \quad (4.4b)$$

Informally, the above definition of functions R and S means that the nodes of each subgraph L_i propagate updates among themselves strictly in a topological order. In addition, propagation takes place along every edge not in E_L (in reverse direction of the edge).

We redefine relation $<$ of Section 4.1 to apply to a pair of transactions that are local to nodes in the same weakly connected component of L. Thus, $T_i < T_j$ *iff* T_i and T_j are local to nodes v and w respectively, $v,\ w \in V_{L_r}$ for some r, $1 \le r \le m$, and $U(T_i)$ is installed at node $ord_r^{-1}(1)$ before $U(T_j)$.

Lemma 4.5. If functions R and S are defined as in (4.4a) and (4.4b), then for every edge $(T_i,\ T_j)$ in the serialization graph where T_i and T_j are local to nodes v and w respectively and $v,\ w \in V_{L_r}$ for some r, $1 \le r \le m$, we have $T_i < T_j$.

Proof. The proof is very similar to that of Lemma 4.2, except that now we have to consider the possibility of $U(T_j)$ reaching node $ord_r^{-1}(1)$ via nodes not in L_r. However, that would imply the existence of an undirected path between w and $ord_r^{-1}(1)$ with nodes not in L_r, which, in turn, means that those nodes must be in the same weakly connected component of L as w and $ord_r^{-1}(1)$, a contradiction. □

Lemma 4.6. Let G_s be a global serialization graph that contains a cycle C. Let G_r be the corresponding *RAG*. Finally, let C be based on G_C, a subgraph of G_r. Then G_C must contain a loop.

Proof. Suppose G_C does not contain a loop. Consider a new database that consists of just those fragments that are vertices of G_C and whose read access patterns are defined by G_C. Suppose that only those transactions that are vertices on C are executed, and the resulting schedule is the one defined by C. Then C is the global serialization graph for the described execution and, as such, must be acyclic by Theorem 4.3, a contradiction. □

Theorem 4.4. An execution characterized by an acyclic *RAG* $G = (V,\ E)$ and the propagation functions of (4.4a) and (4.4b) is guaranteed to be serializable.

Proof. Suppose to the contrary. Let C be a cycle in the serialization graph. We show that C cannot exist. The proof is by induction on t, the number of loops that are in the subgraph of G on which C is based.

From Lemma 4.6 it follows that for $t = 0$ C cannot exist. So let us assume that no cycle can exist in the serialization graph that is based on a subgraph of G with fewer than t loops, and consider cycle C that is based on a subgraph with t loops, $t > 0$.

Let L_r be one of the weakly connected components of L that contribute loops to the subgraph on which C is based. (At least one such component must exist since $t > 0$.) Let T_r^1, ..., T_r^s be all transactions on C that are local to nodes in L_r and such that if edge $(T_r^i,\ T)$ is on C, then T is local to a node not in L_r, for all i, $1 \le i \le s$. Further, let Q_r^i be a transaction on C local to a node in L_r and such that all transactions on the path from T_r^i to Q_r^i that is part of C are local to nodes outside of L_r, for all i, $1 \le i \le s$. That is, all T_r^i's are points at which C exits L_r and all Q_r^i's are points at which C enters L_r.[†]

If for every i, $T_r^i < Q_r^i$, then we can shortcut from T_r^i to Q_r^i (as in the proof of Theorem 4.3). Therefore we end up with a new cycle in the serialization graph that is based

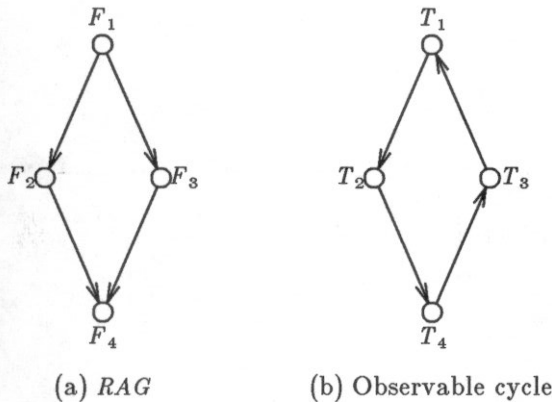
[†] Note that words *enter* and *exit* are used somewhat loosely here because C is a cycle not in G but in the serialization graph.

totally within L_r. It follows trivially from Lemma 4.5, however, that no such cycle can exist. So assume that $Q_r^i < T_r^i$, for some i. Then by shortcutting from Q_r^i to T_r^i we get a new cycle, based on a subgraph of G with fewer than t loops. (It consists of the path from T_r^i to Q_r^i that goes exclusively through the nodes not in L_r, except for its endpoints, and the artificial edge (Q_r^i, T_r^i).) But by the induction hypothesis this is not possible. □

5. Virtual Serializability as Correctness Criterion.

In this section we introduce a new correctness criterion for transaction processing in distributed systems, called *virtual serializability*. It is less strict than serializability, *i.e.*, all serializable schedules are also virtually serializable, but some virtually serializable ones are not serializable in the usual sense.

Before giving a formal definition of the property, we will try to intuitively motivate it. Because of the constraints that we introduced on transaction processing, the following interesting phenomenon is noted. Among all nonserializable schedules, there are some that cannot be identified as such by any transaction. In other words, every transaction in the schedule reads data that could be produced by a serial schedule, even though the global schedule is not serializable.

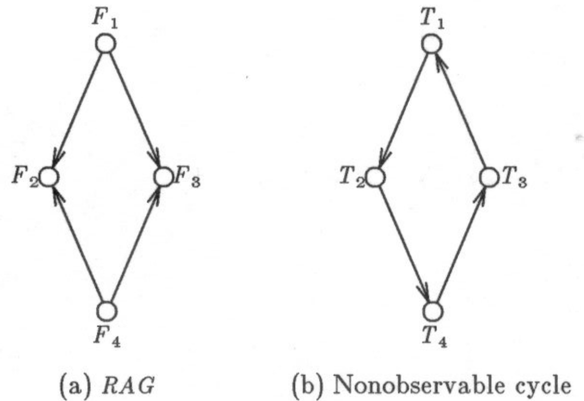


(a) *RAG* (b) Observable cycle

Figure 5.1.

To illustrate let us consider some examples. Figure 5.1 shows a *RAG* and a corresponding serialization graph with a cycle. The fact that the schedule is nonserializable can be detected at node 1 since transaction T_1 reads inconsistent values of data items from fragments F_2 and F_3. These values are inconsistent because they depend on different versions of fragment F_4, before and after transaction T_4 was run.

In the example of Figure 5.2, however, the nonserializable behavior cannot be seen by any of the transactions. Since nodes 2 and 3 do not read data outside of their respective fragments, the fragments cannot become inconsistent (in the sense of the previous paragraph). Thus neither node 1 nor 4 can detect the anomaly. Moreover, the cycle in Figure 5.2(b) was formed because node 1 saw the effects of transaction T_2 before those of T_3 and node 4 saw the reverse. Thus fragment F_1 may now be inconsistent with fragment F_4. However, since there is no legal transaction that can read both of these two fragments, this situation will go undetected.



(a) *RAG* (b) Nonobservable cycle

Figure 5.2.

Definition 5.1. A cycle C in the serialization graph is called *observable* if it can be split into two edge-disjoint paths such that one of them consists solely of dependency edges and the other, solely of precedence edges. (An edge (T_i, T_j) is a dependency edge if T_j reads the value of a data item written by T_i; it is a precedence edge if T_i reads the value of a data item overwritten by T_j [Bern79].)

The cycle in Figure 5.1(b) is observable (edges (T_1, T_2) and (T_2, T_4) are precedence edges; (T_4, T_3) and (T_3, T_1) are dependency edges), whereas the cycle in Figure 5.2 is not observable.

The significance of an observable cycle is that there is a transaction on it, T_1, that may read inconsistent data (see Figure 5.3). T_1 is the first transaction on the precedence path and the

last transaction on the dependency path. When it executes, it reads the new version of fragment F_3 but the old version of fragment F_2. Since the former depends on the state of fragment F_m after the execution of T_m and the latter depends on the state of F_m before the execution of T_m, they are inconsistent. The conclusion that we can draw from this is that observable cycles justify their name.

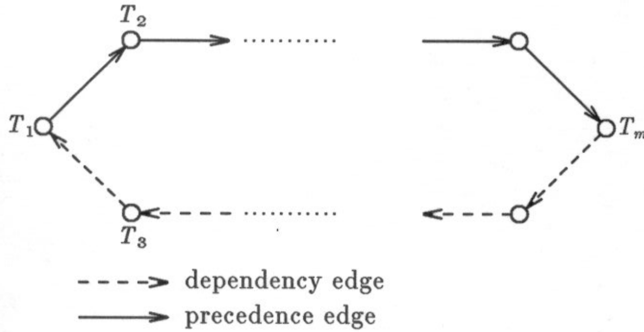


Figure 5.3. Observable cycle. Transaction T_i is local to $N(F_i)$, for all i.

Definition 5.2. A transaction schedule is called *virtually serializable* if the corresponding serialization graph has no observable cycles.

Definition 5.3. Let $L = (V_L, E_L)$ be a *RAG* loop. Let vertex $s \in V_L$ be such that no edge in E_L is incident upon it. Let vertex $t \in V_L$ be such that no edge in E_L is incident from it. Then s is called a *loop source* of L, and t, a *loop sink* of L.

Clearly, if a loop doesn't have loop sources or loop sinks, it is a (directed) cycle. Also, it is easy to see that the number of loop sources in a loop is always equal to the number of loop sinks.

Definition 5.4. L is a *single-source* (*single-sink*) loop if it contains exactly one source (sink). It is a *multi-source* (*multi-sink*) loop otherwise.

Lemma 5.1. (Acyclic) *RAG*s with no single-source loops guarantee virtually serializable schedules regardless of the choice of propagation functions.

Proof. Let G_r be an acyclic *RAG* with no single-source loops. Suppose that C is an observable cycle in the serialization graph. First of all, note that a precedence edge between two transactions that are local to different nodes always repeats the direction of the *RAG* edge between those nodes, whereas a dependency edge always goes in the opposite direction.

Since G_r has no single-source loops (or cycles), C must be based on a subgraph of G_r that includes a multi-source loop. But then C cannot be split in two paths as required by Definition 5.1 because the direction of edges along the loop changes more than twice (see Figure 5.2(a)). Therefore C is not observable. □

Since cycles in *RAG*s can cause the creation of cycles in serialization graphs (including observable cycles) regardless of how propagation functions are defined, we still restrict ourselves to acyclic *RAG*s, just as we did in Section 4.

Replacing serializability with virtual serializability as correctness criterion helps make update propagation less restrictive. For *RAG*s with no single-source loops, for insatnce, update propogation can be totally unrestricted (even in the presence of multi-source loops). Recall for comparison, however, that to acheive serializability, we had to use restricted protocols for update propagation in the presence of any kind of loops.

For general acyclic *RAG*s, we can also make an improvement in the efficiency of the protocols, provided we can be satisfied with virtual serializability. Let $G = (V, E)$ be a *RAG*. Let $G_S = (V_S, E_S)$ be a subgraph that contains all the edges of G that lie on single-source loops. Let $ord: V_S \rightarrow \{1, ..., k\}$, where k is the cardinality of V_S, be a topological order on the vertices of G_S. Define the propagation functions as follows:

$$R_1(i) = \{ord^{-1}(ord(i) + 1)\} \bigcup \{j: j \in V - V_S \text{ and } (i, j) \in E\}$$

$$R_2(i) = \{j: j \in V - V_S \text{ and } (i, j) \in E\}$$

$$R_3(i) = \{j: j \in V \text{ and } (i, j) \in E\}$$

$$R(i) = \begin{cases} R_1(i) & \text{if } i \in V_S \text{ and } ord(i) < k \\ R_2(i) & \text{if } i \in V_S \text{ and } ord(i) = k \\ R_3(i) & \text{if } i \in V - V_S \end{cases} \quad (5.1a)$$

$$S_1(i) = \{ord^{-1}(ord(i) - 1)\} \bigcup \{j: j \in V - V_S \text{ and } (j, i) \in E\}$$

$$S_2(i) = \{j: j \in V - V_S \text{ and } (j, i) \in E\}$$

$$S_3(i) = \{j: j \in V \text{ and } (j, i) \in E\}$$

$$S(i) = \begin{cases} S_1(i) & \text{if } i \in V_S \text{ and } ord(i) > 1 \\ S_2(i) & \text{if } i \in V_S \text{ and } ord(i) = 1 \\ S_3(i) & \text{if } i \in V - V_S \end{cases} \quad (5.1b)$$

In simple terms, this means that the nodes of subgraph G_S enact among themselves a topological propagation protocol, whereas the nodes not in G_S propagate updates along all edges in $E - E_S$ (in reverse direction of the edges). In addition, propagation occurs along every edge connecting a node in G_S and a node not in G_S.

Theorem 5.1. Propagation functions R and S defined in (5.1a) and (5.1b) guarantee virtual serializability for schedules characterized by acyclic *RAG*s.

Proof. Suppose this is not true. Let C be an observable cycle in the global serialization graph. From Definition 5.1 and the observation in the proof of Lemma 5.1, it follows that the subgraph on which C is based must be a single-source loop. Therefore, C is based on a subgraph of G_S. (G_S contains all edges on single-source loops.)

The nodes of G_S, according to (5.1a) and (5.1b), engage in a topological propagation. In the absence of nodes outside G_S, Lemma 4.2 (and hence Theorem 4.1) would be valid and there could be no cycles like C. However, it could be the case that update propagations by nodes outside G_S invalidate Lemma 4.2. We now show that this is not the case.

Consider the edge (T_j, T_i) in C. If T_i and T_j are local to the same node, or if the edge is a dependency edge, then whatever node receives $U(T_i)$ must receive $U(T_j)$ first. ($U(T_j)$ is added to $UPDATES(i)$ before $U(T_i)$; see Section 2.1.) Therefore, $T_j < T_i$. However, if (T_j, T_i) is a precedence edge, $U(T_i)$ is sent out of node i without $U(T_j)$. If $U(T_i)$ were only transmitted along the topological propagation chain, then $U(T_j)$ would arrive first at all nodes, including the last one, l ($l = ord^{-1}(1)$). However, by following a path outside G_S, $U(T_i)$ can conceivably skip ahead of $U(T_j)$ and arrive at l first. For this to happen, there must be a *RAG* path leading to the node where T_i executed, say v, and originating at some other node in G_C, call it w. (Actually, $ord(w) < ord(v)$.) As mentioned earlier, the path (except the endpoints) must be outside G_S, else $U(T_i)$ could not skip ahead.

However, any (unidirectional) path connecting two vertices in a single-source loop creates another single-source loop. (Recall that the RAG is acyclic.) Therefore, the path must also be in G_S, a contradiction. □

6. Conclusions.

We believe that Bakunin data networks offer a valuable alternative to known mechanisms for managing replicated data. Our mechanism yields a high degree of data availability and node autonomy while still ensuring serializability (or virtual serializability). It tolerates arbitrary communications failures, including network partitions that are not consistently detected.

The major drawback is that transactions must be pre-analyzed and the resulting *RAG* must be acyclic. (Although we did not discuss it here, these assumptions can be periodically relaxed. However, to avoid inconsistencies, some type of locking must be performed in these cases. The result is that availability suffers.) We believe that this is not a limitation of our particular approach, but an inherent property of dynamic failure environments. If for a particular application it is essential to run arbitrary transactions, then one must either give up serializability or the ability to operate in dynamic failure environments. If the application demands serializability and operation in the face of unpredictable communication failures, then the transactions must be simple enough so that they can be pre-analyzed. Furthermore, it is necessary to structure the application so that updates to a fragment originate at a single node and the resulting *RAG* is acyclic.

If a Bakunin approach is used, then the major implementation problem is the propagation of updates. If transactions running at a node $N(F_1)$ read data from fragment F_2, it is important that the updates from $N(F_2)$ reach $N(F_1)$ as expediently as possible. For loopless *RAG*s and for no-single-source-loop *RAG*s under virtual serializability, we have obtained the most efficient propagation protocols possible. For other *RAG*s, our protocols yield good performance, although finding the best one is an open question. For instance, the chain protocol operates correctly with any topological ordering on the nodes, but finding the best ordering for a particular communication network may be a difficult problem.

7. Acknowledgment.

The authors wish to thank Daniel Barbara for his help in developing the basic topological propagation protocol for acyclic *RAG*s.

8. Bibliography.

[Bern81] Bernstein, P.A., and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys*, Vol. 13, No. 2, June 1981, pp. 185-221.

[Bern79] Bernstein, P.A., D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. on Software Engineering.*, Vol. SE-5, Num. 3, May 1979, pp. 203-216.

[BlKa85] Blaustein, B.T., and C.W. Kaufman, "Updating Replicated Data During Communications Failures," *Proc. 11th VLDB*, 1985, pp. 1-10.

[Chri75] Christofides, N., *Graph Theory: An Algorithmic Approach.* Academic Press, 1975.

[Davi84] Davidson, S.B., "Optimism and Consistency in Partitioned Database Systems," *ACM Trans. Database Syst.*, Vol. 9, Num. 3, September 1984, pp. 456-481.

[Davi85] Davidson, S.B., H. Garcia-Molina, and D. Skeen, "Consistency in Partitioned Networks," *ACM Computing Surveys*, Vol. 17, Num. 3, September 1985, pp. 341-370.

[Eswa76] Eswaran, et. al., "The Notions of Consistency and Predicate Locks in a Database System," *Communications of the ACM*, Vol. 19, Num. 11, November 1976, pp. 624-633.

[GaKo87] Garcia-Molina, H., and B. Kogan, "Achieving High Availability in Distributed Databases," *Proc. 3rd International Conf. on Data Engineering*, Los Angeles, February 1987.

[Giff83] Gifford, D.K., "Weighted Voting for Replicated Data," *Proc. 7th Symp. on Princ. of Database Syst.*, March 1983, pp. 8-15.

[Mehl84] Mehlhorn, K., *Graph Algorithms and NP-Completeness*, Springer-Verlag, 1984.

[Park81] Parker, R.S., *et. al.*, "Detection of Mutual Inconsistency in Distributed Systems," *Proc. 5th Berkeley Workshop on Distributed Data Management and Computer Networks*, February 1981.

[Sari85] Sarin, S.K., "Robust Application Design in Highly Available Distributed Databases," Computer Corporation of America, Technical Report, May 1985.

[ScSc83] Schlicting, R.D., and F.B. Schneider, "Fail-Stop Processors: An Approach to Designing Fault-Tolerant Computing Systems," *ACM Transactions on Computing Systems*, Vol. 1, pp. 222-238, August 1983.

[SkWr84] Skeen, D., and D. Wright, "Increasing Availability in Partitioned Networks," *Proc. 3rd ACM SIGACT-SIGMOD Symp. on Princ. of Database Systems*, April 1984, pp. 290-299.