BAKUNIN DATA NETWORKS: AN APPROACH TO DESIGNING HIGHLY AVAILABLE REPLICATED DATABASES

Boris Kogan

(Thesis)

CS-TR-173-88

September 1988

BAKUNIN DATA NETWORKS: AN APPROACH TO DESIGNING HIGHLY AVAILABLE REPLICATED DATABASES

Boris Kogan

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

OCTOBER 1988

ACKNOWLEDGMENTS

The word advisor does not adequately describe what Hector Garcia-Molina has been to me. He sparked my interest in database systems and distributed computing. He provided many insights into the problems I was trying to solve. He was always available when I needed to air my ideas. He taught me the skill of technical writing. And above all, he gave me inspiration by his personal example of great efficiency and effectiveness in conducting research as well as by his unwaivering enthusiasm and good spirits. For all this, I am very grateful to him.

I would like to thank my readers, Rafael Alonso and Jeffrey Naughton, who took the time and effort to thoroughly read this dissertation. Their valuable comments and suggestions have undoubtedly improved the quality of it.

I would like to thank Daniel Barbara for the fruitful discussions that we had.

I would like to thank Eugene Davidson, Ruth James, Gerree Pecht, and Sharon Rodgers for being so helpful with administrative matters and, perhaps even more importantly, for being so amazingly nice and cheerful.

I would like to thank all my Princeton friends and fellow graduate students for their large contribution to making my Princeton years both pleasant and productive.

Finally, I am greatly indebted to my family (although I know how totally alien the idea of familial debts is to them) for the solid and unquestioning support they have given me in all my undertakings.

ABSTRACT

Data replication in distributed databases is used to improve data availability and provide faster read access. However, new problems for database management are introduced due to the need for maintaining the consistency of replicated copies. Dealing with communication failures is one of them. For example, when the network partitions, continued update activities may lead to data inconsistency. On the other hand, halting all updates until the network is reconnected is undesirable in many cases.

This dissertation introduces a methodological framework for designing replicated databases that exhibit a high degree of availability (including the ability to make updates) in the face of communications failures that can partition the network. The framework is based on the simple notions of *fragments* and *agents*. It gives rise to a wide scope of options with varying degrees of data consistency and availability. The consistency criteria offered by these options include one-copy serializability as well as two new criteria: *virtual serializability* and *fragmentwise serializability*.

to the memory of my grandparents

Table of Contents

CHAPTER 1

INTRODUCTION

1.1. Replicated Data and Communication Failures.

One of the central motivations for studying and using distributed computer systems is improved availability of computing resources that they offer in the face of machine crashes. In a centralized system, a single site crash brings the entire system down (a trivial fact since there is only one site). In a distributed system, on the other hand, when a single site (or even several sites) goes down, the system may still remain operational.

Unfortunately, this advantage of distributed systems is offset by a whole new type of problems, unknown in centralized systems: communication failures. In distributed databases, the primary resource is data. In many cases data are replicated over several sites to improve availability in case of individual site crashes. However, communication failures can often complicate the management of such systems and, in the absence of any special provisions, even degrade the overall performance in comparison with the centralized case.

This paradoxically adverse effect of data replication is due to the necessity to synchronize updates performed on the replicas. Synchronization is necessary in order to preserve consistency of data. Therefore when a failure in the network halts communications, proceeding with updates entails the risks of creating inconsistent copies. Thus, updates may be blocked even without a site crash, a phenomenon that could never occur in a centralized database.

Consider what may happen if updates are not blocked when communications are disrupted. Let x_A be the copy of data item x stored at site A. Let x_B be the copy of x

at site B. Suppose that A and B cannot communicate during a certain interval of time. If during this interval both A and B update their copies of x, how are the possibly diverging values of x_A and x_B to be reconciled? What should be the "correct" value of x?

Note that, during a disruptive communication failure, an item does not have to be updated at several sites in order to create problems with data consistency. Suppose there were a second data item y with copies y_A and y_B at A and B respectively. During the failure the following (unsynchronized) updates could occur: $x_A \leftarrow f_1(y_A)$ and $y_B \leftarrow f_2(x_B)$ at A and B respectively, where f_1 and f_2 are some functions on appropriate domains. Each data item has been updated at a single site only, and the replicas of each, therefore, can eventually assume the same values after the updates are propagated. Nevertheless, this execution is intuitively incorrect. The problem is that these updates cannot be sequentially ordered since the new value of each of the two items was computed based on the old value of the other.

Thus, it appears that data availability and data consistency are competing goals of distributed database systems that are prone to communication failures: when one is fully satisfied, the other may be compromised.

A failure that results in disruption of communications between sites in a distributed system is called a *network partition.* A network partition divides all the sites into *partition groups* such that any two sites can communicate with each other if and only if they are from the same partition group. Assuming that partitions are easy to detect, a safe and simple, albeit conservative, approach to preserving data consistency is to halt all transactions, i.e., suspend access to all data, for the duration of the partition.

It does not take long to see, however, that such an approach is unacceptable in many real situations. In many applications it is undesirable to halt transaction

processing whenever a partition occurs. In some cases it is because partitions occur at critical times. For instance, in command and control applications a military conflict can make communications unreliable and it is precisely at this time that one wishes to have access to the data. In other cases, halting transaction processing causes a serious inconvenience or economic loss. For example, an unavailable airline reservations system means lost customers, and hence lost revenues.

In yet other cases, partitions are normal rather than extraordinary occurrences. This may be because the network is inherently very unreliable or because communication lines are not dedicated. For example, in the Unix UUCP network computers are normally disconnected from the network. Only occasionally, and for usually short periods of time, do they establish point-to-point connections (by dialing other computers) and exchange data. A similar situation exists in a system that uses satellites for communications among geographically dispersed computer nodes (or clusters of such). Communication between a pair of nodes (clusters) is possible only for a relatively brief period of time when the satellite is in the right position within its orbit allowing it to have radio contact with both nodes (clusters). In such an environment one would like to process transactions even when there is no communication.

Also note that even in conventional networks, partitions may be more common than what one might initially imagine. First, partitions do occur when gateways connecting networks fail or when Ethernet cables are inoperative (e.g., when an Ethernet terminator is removed, say to add a new segment). Second, in most cases it is impossible to distinguish a node failure from the disconnection of a node from the network (e.g., because its ARPANET IMP or its Ethernet transceiver failed). Hence, every node failure must be considered a partition. Finally, "virtual" partitions occur when a computer does not respond promptly to network messages. That is, network protocols usu-

ally have a timeout period. If a computer does not respond within this time limit, it is declared failed, even though it could be operational but slow. The timeout period has to be set relatively short, else it would take an unreasonable amount of time to detect and compensate for real failures. This means that overloaded or slow computers may occasionally cause virtual partitions when they respond late. In summary, even though partitions may not be frequent events, they may occur with enough frequency so that halting transaction processing every time they arise is not acceptable.

1.2. Related Work.

From the discussion in Section 1.1, it becomes apparent that maintaining data availability in the face of communication failures is an important practical problem. It has been recognized as such and studied by many researchers in recent years. We present here a brief overview of some of the approaches to the problem (see [Davi85a] for a thorough survey).

The intuitive notion of data consistency is formalized by introducing a correctness criterion for transaction execution such that when this criterion is satisfied, the data are considered consistent. For example, the widely excepted correctness criterion for transaction execution on replicated data is *one-copy serializability* [Bern86a]. Informally, a one-copy serializable transaction schedule is equivalent (in terms of the output produced by its transactions and the state in which it leaves the database upon completion) to some serial schedule of transactions that have access to only one and the same copy of every data item. Thus one-copy serializable schedules prevent any possible adverse effect that concurrency of transaction execution or replication of data might have on data integrity.

All the known methods for increasing data availability during partitioned operation can be grouped into two categories. The first category is characterized by strict

adherence to one-copy serializability, even during communication failures. This is typically achieved by limiting data access in different partition groups in a manner that excludes the possibility of conflicts among transactions executing in these groups (e.g., a data item will be allowed to be updated in one group only). So even though this strategy is clearly superior, in terms of data availability, to simply halting all transaction execution for the duration of the partition, it does limit availability significantly. This category of approaches is referred to as *restrictive* (this corresponds to what are called pessimistic methods in [Davi85a]).

The second category consists of methods that, instead of one-copy serializability, emphasize unrestricted availability in the face of communication failures. They put no limitations whatsoever on data access during partitions. Special techniques are provided for making data consistent when the network becomes connected again. Note, however, that even when the system is successful in eventually arriving at a consistent state for the database, the transaction execution that took place during the partition cannot be considered correct (in the sense of one-copy serializability) because incorrect outputs may have been produced that cannot be retracted. The second category is designated *free-for-all*† (this corresponds to optimistic methods in [Davi85a]).

1.2.1. Restrictive Methods.

Gifford proposed a technique called *weighted voting*, based on assignment of a non-negative number of votes $v(x_J)$ to each copy x_J of data item x [Giff79a]. Non-negative read and write thresholds, r_x and w_x respectively, are defined for x such that:

$$w_x + r_x > \sum_{all\ J} v(x_J) \tag{1.1}$$

† The main concern of the present dissertation is data availability. Consequently, we would like to give the two categories of techniques the appropriate designations to reflect how they affect availability. That is why we insist on renaming them.

$$2w_x > \sum_{all\ J} v(x_J) \tag{1.2}$$

Version numbers are used for all copies of x. To update x a transaction T must "collect" at least w_x votes. This means that T must initiate write operations on a quorum of copies whose total number of votes is at least w_x, and commit only after all these operations complete. Similarly, to read x a quorum with at least r_x votes must be accessed and the value of the copy with the largest version number taken. Since any read quorum intersects any write quorum (see inequality (1.1)), that value is guaranteed to be the most recent version of x. If due to communication failures a transaction fails to secure enough votes to perform an operation on a data item, this transaction must abort. Inequality (1.2) guarantees that in a partitioned network x can be updated in at most one partition group. Inequality (1.1) guarantees that if x is updated in one group, it cannot be read in any other. Thus no conflicts (either write—write or read—write) can ever arise. Based on this fact it can be shown that the execution is always one-copy serializable.

The described technique is also known as the *quorum consensus* algorithm and is a generalization of the *majority consensus* algorithm (in which exactly one vote is assigned to every copy) introduced by Thomas [Thom79a].

In a related approach proposed by El Abbadi et al. [Abba85a, Abba86a], read and write thresholds are assigned in a similar fashion but used differently. The *view* of a node is defined to be the set of nodes with which this node believes it can communicate. For the method to work correctly, views do not have to be consistent with actual partition groups. For that reason, this method is known as the *virtual partitions* algorithm. A data item can be updated (read) by a transaction only if the view of the transaction's home node has the required quorum of votes (note that this can be determined just from the information at the home node). But to actually write a data item a transac-

tion has to write *all* copies of it within the transaction's view. To read an item *any* copy can be accessed. Correctness of the algorithm is guaranteed by a special *view update transaction.* This transaction is initiated by a node when the node detects a discrepancy between its view and the real communication status of the system (e.g., contact is lost with a node in the view). The main purpose of this transaction is to consistently update the views of all the nodes involved and to bring up-to-date all copies of those items that nodes in the new view will be able to read.

Skeen and Wright developed a different type of restrictive method for maintaining availability [Skee84a, Wrig83a]. In their approach, all transactions are pre-analyzed and divided into classes according to their read and write sets. A *class conflict* graph is constructed that models interactions and possible conflicts among transactions belonging to different classes. When a network partion occurs, the class conflict graph and the partition map are analyzed to determine the classes of transactions that can be executed at a given node safely, *i.e.*, with guarantees of a one-copy serializable schedule. Unlike the previous approaches, the class conflict analysis (as this approach is known) does not exclude the possibility of a data item being read in one partition and updated in another. The class conflict analysis assumes that partitions are correctly detectable.

Some of the other restrictive schemes for availability include the missing writes [Eage83a], the primary copy [Alsb76a], and the token schemes [Mino82a].

1.2.2. Free-For-All Methods.

Davidson proposed a method called the *optimistic protocol* [Davi84a]. It uses the following strategy for making the database consistent after the partition is repaired. Based on transaction logs that each network site kept during the partition, an extended form of the serialization graph [Papa79a] is constructed. This graph models dependencies among transactions not only from the same partition group but from different ones

as well. If the graph turns out to be acyclic, that means that the execution was serializable, and no action needs to be taken. Otherwise, some transactions have to be rolled back to render the graph acyclic. That in turn may necessitate roll-backs of other transactions that depended on the updates produced by the first group of transactions and so on (this is known as cascading roll-backs). The optimistic protocol provides heuristics for keeping the number of roll-backs small (the problem of minimizing this number is NP-complete). Note that even though the resulting state of the database is consistent, some transactions cannot be truly rolled back because their outputs cannot be annulled. For some applications this can be a serious problem.

The *log transformation* technique introduced by Blaustein and Kaufman [Blau85a] (and used in the SHARD system Sari86a) is also based on backing out some of the transactions that executed during the partition. Unlike the optimistic protocol, however, instead of syntactic information (the serialization graph), this technique uses the available semantic information (e.g., commutativity) about transactions to avoid unnecessary backing out and rerunning. Like the optimistic protocol, the log transformation does not guarantee serializability.

Another approach that utilizes semantic knowledge was proposed by Garcia-Molina et al. [Garc83a]. It puts forward *Data-Patch* — a tool based on a collection of rules for integrating possibly diverging copies of data after the network is reconnected. The process of integration is no longer carried out on the basis of transaction logs, as in the previous two approaches, but rather on the basis of the values of different copies of the same data item. The goal is to arrive at a single, "reasonable" value for all copies. One-copy serializability is abandoned altogether.

1.3. The Objectives and Tools of the New Approach.

Restrictive methods are best suited for the kind of database applications in which one-copy serializability is of utmost importance. Free-for-all methods are called for when losing, or even restricting, data availability during network failures is highly costly. On the negative side, however, the former compromise availability during a partition, while the latter lack meaningful correctness criteria.

In this dissertation a new approach to the problem is presented that attempts to remedy the situation to a certain extent. The objectives are high availability in the face of communications failures and, at the same time, adherence to meaningful and fairly strict correctness criteria. High availability will be understood to mean the ability of every node to access the same data during a partition (or other kinds of failures) as during normal operation. The possible correctness criteria will include one-copy serializability. In addition, new criteria will be introduced, which are less strict but quite useful for some applications.

Assume for now that data replication is complete, i.e., every node in the network has a copy of the entire database. The following main tools, or strategies, will be employed to achieve the defined objectives of availability and correctness:

(i) *Operational Node Autonomy.* Nodes operate in total autonomy insofar as transaction scheduling and execution are concerned. No synchronizing, or concurrency-control, messages (e.g., remote lock requests) are exchanged. Transactions are scheduled, run, and commit locally, without ever being blocked because of waiting for a message from another node.

(ii) *Restricted Transactions.* Arbitrary transactions are not allowed. For every node, there is a specified subset of data accessible to local transactions in read mode, and a subset accessible in update mode. These subsets differ from node to node and

can change over time, but they do not depend on the communication status of the network. Thus, during a partition any node is allowed to access the same data it can normally access. Restricted transactions are motivated by the observation that, in real distributed systems, democracy is not a very popular concept, i.e., different users are endowed with different sets of privileges as far as data access is concerned, and the ability of some users to modify (and even access) data is often restricted. For example, in an airline reservations system, it can hardly be considered a loss of availability if a travel agency, operating at its own computer node, is unable to update the flight schedules in the database. Our approach will try to exploit such inherent restrictions that many applications have.

(iii) Restricted Update Propagation. In a replicated database updates produced at some node must be eventually propagated to all other nodes that hold a copy of the relevant data item. Traditionally this is done without any regard to the order in which different nodes receive and install these updates. In the proposed approach, there will be some special restrictions on this order. Sometimes updates might even be delayed on purpose to enforce these restrictions. Note, however, that this has no effect on the operational autonomy of nodes.

It is immediately clear that tool *(i)* — node autonomy — is the guarantee of unrestricted[†] availability of data during partitions and other communication failures. Correctness properties are enforced by the combination of tools *(ii)* and *(iii)*, as will be demonstrated later on in this dissertation.

It is worth noting that node autonomy is a very valuable property to have in a distributed database (or any distributed system for that matter), even apart from its

[†] In the sense of allowing access to the same data during failures as during normal functioning of the communication network.

role in attaining high data availability in the proposed scheme. The reasons for that are as follows.

Many of the previously proposed approaches assume that the nodes belonging to a partition group have complete information about the members of the group, and that the partition borders do not change dynamically. For instance, this implies that all the links whose failure caused the partition are repaired at the same time. We call this type of behavior *clean partitions.* The notion of clean partitions may not represent a realistic scenario. Consider a network where failures of communication links occur fairly often and repairs do not take long. That will result in the rapidly changing status of the communication network. Moreover, as pointed out earlier, significant communication delays are not easily distinguishable from failures. Clearly, under such conditions it will be extremely difficult to detect when partitions occur and when they are repaired. It will also be difficult to determine the exact constituency of every group. This scenario is referred to as the *dynamic failure environment.* Since the dynamic failure environment represents a more realistic model for an asynchronous communication network, it is highly desirable that our approach be compatible with this model. Since node autonomy provides a degree of independence from communication failures, our approach will not require prompt or correct detection of partitions and will be able, therefore, to operate in the dynamic failure environment.

Autonomous operation prevents message delays from contributing to the transaction response time. Message delays can be quite significant for some networks (e.g., long-haul networks) and, in fact, constitute a large portion of the response time when traditional concurrency control and commit mechanisms are used. For that reason autonomous operation implies significantly better response time. In fact, the approach to dealing with communication failures in replicated databases presented here can also

be viewed as a very efficient robust concurrency control mechanism. Thus, it turns out that the tools used by our approach provide us with additional "bonuses," apart from correctness and high availability. Figure 1.1 gives a schematic representation of the relationships among the objectives, tools, and bonuses.

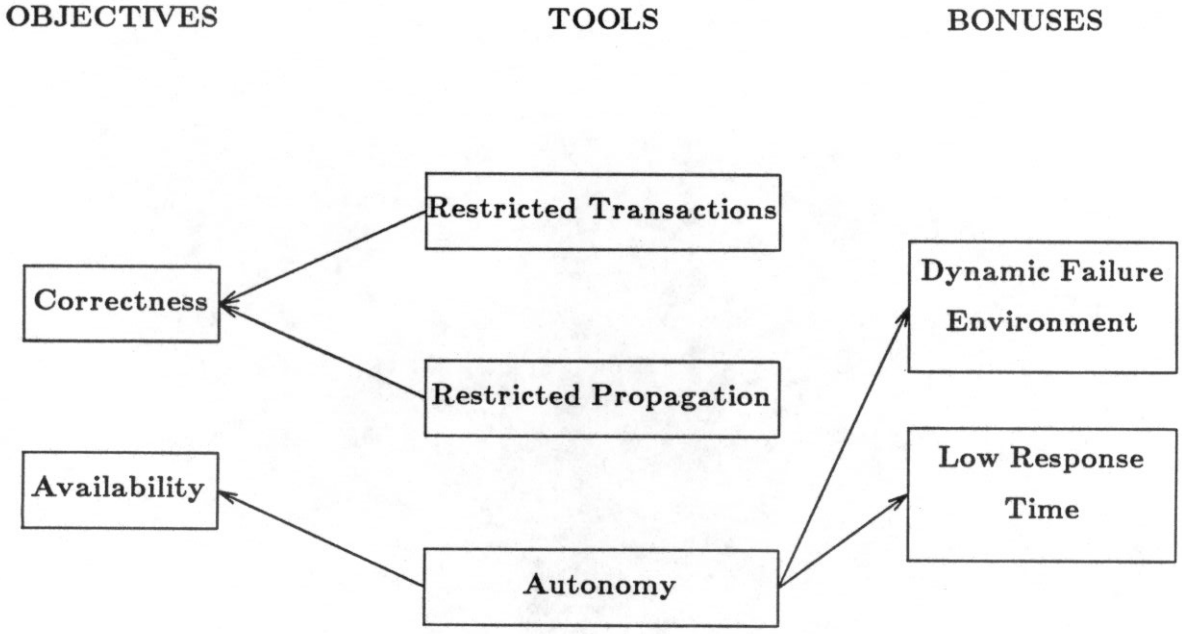


Figure 1.1. Objectives, Tools, and Bonuses.

Finally, note that node autonomy in a distributed system reflects the typical structure of many organizations that might use such a system. Namely, loose administrative links among the constituent departments often imply the need for relative independence among the corresponding computer nodes. Cooperation is still present and important (exemplified, for instance, by information exchange in the form of update propagation), but the ability of every node to function on its own even if it cannot communicate with other nodes is no less essential. Moreover, even during normal communications it may be undesirable to have globally synchronized data access, because it makes the transaction throughput at an individual node dependent on how quickly other nodes can

respond to synchronizing messages.

As a result of the operational autonomy, individual nodes are endowed with absolute freedom (in scheduling their transactions). On the other hand, overall harmony (of execution correctness) is nevertheless achieved. Because of the analogy to an ideal anarchistic society, we have termed our model systems *Bakunin data networks* , after the 19th century Russian political theorist and advocate of anarchism.

1.4. The Outline.

The rest of this dissertation is organized as follows. In Chapter 2 the basics of the proposed model are presented. In Chapter 3 we show how to guarantee one-copy serializability for transaction schedules in Bakunin data networks. Several possible options for doing that are given and proven correct. These options are obtained by varying the types of restrictions on transactions and on update propagation. Chapter 4 introduces the notion of virtual serializability — a new correctness criterion for transaction processing that is less restrictive and cheaper to guarantee than one-copy serializability. As in Chapter 3, several ways of enforcing virtual serializability are presented (complete with proofs of correctness). In Chapter 5 the trend towards less restrictive correctness criteria is continued. We introduce the notion of fragmentwise serializability and show how this criterion can be achieved. In Chapter 6 we present various possible extensions to the basic model that make our approach more general and flexible. Finally, in Chapter 7 the main results of this dissertation are summarized and some conclusions are drawn.

CHAPTER 2

BAKUNIN DATA NETWORKS: A FRAMEWORK

This chapter is devoted to the description of our model. The model is comprised of elements of the standard model for distributed database systems enhanced by special considerations essential to our approach. The goal of the model is to create a comprehensive framework for designing highly available database systems.

We concentrate on the issues of data organization, transaction scheduling, and distribution of updates. Special attention is given to the notion of data control.

2.1. The Network.

The distributed system supporting our database application consists of n computer sites, or nodes, interconnected by a communication network. Throughout this work we assume that the network is point-to-point, however, all the results developed here will be easily adaptable to networks containing broadcast links.

Nodes have stable storage that never fails (although the nodes are allowed to fail). The network is unreliable. A dynamic failure environment is assumed (see Chapter 1), i.e, links of the network can go down and come back up at any time and failures are not automatically detectable. Messages can experience arbitrary delays. There is, however, a facility that guarantees that all messages are eventually delivered in the order sent.

2.2. Replicated Data.

The database is a set of data objects each of which is replicated (has a physical copy) at several sites. The subset of sites which hold a copy of object x is called the *realm of replication* of x. The cardinality of this subset is called the *degree of replication*

of x. The replication is *complete* when the realm of replication for every object in the database is the set of all participating sites.

From the discussion in Chapter 1 it should be clear that increasing the degree of replication for a data object improves its accessibility, but also makes maintaining of data consistency a more difficult task.

We shall assume, for now, that replication is complete, unless otherwise specified. (Later in this dissertation we shall take a more refined view of the subject of replication.) The main motivation for making this assumption is the simplicity of presentation it affords. The results developed in the rest of this dissertation, however, can be applied to the case of arbitrary levels of replication (see Chapter 6).

2.3. Fragments and Agents.

The very simple notions of fragments and agents form the basis of our framework.

The entire database is divided into k non-overlapping subsets called *fragments* and denoted F_1, F_2, ..., F_k. Every data object x belongs to a unique fragment of the database: $x \in F_i$, $1 \leq i \leq k$. (It should be remembered that a data object is a logical entity and is not be confused with its physical replicas. A fragment, then, is a logical entity too.)

Each fragment F_i is controlled by a unique node $N(F_i)$ that is responsible for updating it. The process that realizes the control is called an *agent* and is denoted $A(F_i)$. Thus every fragment has a corresponding unique agent. Note that an agent can be implemented as a separate process or as a component of the local data manager. In either case, updates to a fragment can only be performed at the node where the corresponding agent currently resides.

One way to insure the uniqueness of agents, i.e., to insure that no more than one

node at a time has an active agent for a given fragment, is through the use of *tokens*. These tokens, however, are different from the traditional tokens used in distributed systems (see [Mino82a], for instance). For every fragment, there is exactly one token, and it can be owned by a user (or consortium of users) as well as by a computer node. Thus our tokens have existence outside of the computer system and can be passed by means other than electronic messages, a situation quite different from the traditional use of tokens. This property allows for transfer of tokens from node to node even when the two nodes belong to different partition groups, i.e., are currently unable to communicate with each other. An agent can be created only by the current owner of the corresponding token and is automatically destroyed upon the withdrawal of the token.

As an example of a token, consider the card that a bank customer uses to identify himself to an automatic teller. Whoever owns the card is authorized to perform banking operations on the corresponding account, i.e., to update the fragment containing deposit/withdrawal information for that account. One should not infer, however, from this example that all tokens must have a concrete physical embodiment.

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 $V = \{F_1, F_2, \ldots, F_k\}$ and $E = \{(F_i, F_j) : i \neq j$ and a transaction T that is initiated by $A(F_i)$ is allowed to 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.

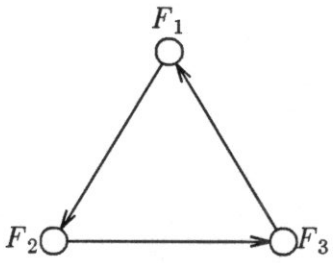


Figure 2.1. RAG.

For the sake of simplicity, we shall assume for now that there is a one-to-one correspondence between fragments and nodes, i.e., not only is every fragment controlled by a unique node, but also every node controlls exactly one fragment. Like the complete replication assumption, the assumption of the one-to-one correspondence is not limiting to our approach and can be relaxed later (as will be shown in Chapter 6). It implies, in particular, that $k = n$. Also, until Chapter 6, we assume that all agents are stationary, i.e., do not move from site to site. These two assumptions together imply that we can alternatively regard the vertices of the RAG as nodes of the network rather than fragments, whenever it is more convenient.

2.4. Transaction Scheduling and Distribution of Updates.

A transaction is taken to be a sequence of any number of atomic read or write actions [Eswa76a]. Each transaction executes and commits 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, as specified by the corresponding RAG. (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.

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 activity at 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. The local concurrency control mechanism schedules transactions local to the given node as well as updates produced by nonlocal transactions that have been received for installation in the local copy of the database. For the purposes of local concurrency control, updates produced by the same nonlocal transaction can be viewed as a special "write-only" transaction to be executed locally. Thus, if transaction T_r is nonlocal to node i, and its updates $U(T_r)$ are received by i, then $U(T_r)$ is treated by i's transaction scheduler just as if it were a local transaction (the only difference is that there are no read actions in $U(T_r)$, which reflects the requirement that updates produced by a transaction be unconditionally installed at other nodes after the transaction commits at the originating site).

To describe the interaction between the local transaction scheduling and propagation of updates, we introduce the *generic* update propagation protocol. This protocol will also provide us with an instrument for restricted propagation when needed. We identify here the nodes of the RAG with the computer sites of our network.

Definition 2.2. 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: $\{1, ..., n\} \rightarrow 2^{\{1, ..., n\}}$). Let $S(i)$ be the set of all nodes to which node i sends update messages (S: $\{1, ..., n\} \rightarrow 2^{\{1, ..., n\}}$). R and S are called *propagation functions.*

Functions R and S are mutually redundant since if one is fully specified then we can also derive the value of the other for any i. 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 that 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, when all the agents are stationary, 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. For the sake of simplicity, we will ignore any redundancies associated with delivering $U(T_s)$ to other nodes. 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 further down on the given propagation path.

By choosing different propagation functions R and S, we can derive an entire family of specific protocols from the general update propagation framework presented above. 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 (see Chapters 3, 4, and 5). Second, they must provide, as much as possible, prompt delivery of updates. Here we are going to discuss the second criterion, which will be also referred to as performance of a protocol or expediency of update propagation.

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 even assuming that these factors are constant different protocols (different propagation functions) will perform differently.

Consider two nodes in a Bakunin data network, 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 2.3. 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 $E_p = \{(i, j)$: $i, j \in V$ and $j \in S(i)$ (or $i \in R(j))\}$.

Definition 2.4. 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. One reason for this is that short characteristic paths imply that there are fewer nodes between the sender and the receiver that have to install the updates in their local copies. Thus the delays will be shorter. In addition, a characteristic path can be the only path between a pair of nodes in the propagation graph (a situation common for the propagation functions considered in subsequent chapters). If that is the case and if some node on the characteristic path gets isolated from both the sender and the receiver, due to a network partition, then the updates will not reach the receiver until the partition is repaired, even if the receiver is in the same partition group as the sender. Thus, a shorter characteristic path (when it is the only path from the sender to the receiver) will have fewer vulnerabilities with respect to communication failures.

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 chapters, it is not always possible to have propagation functions that guarantee both good performance and adherence to the required correctness criteria.

[†] Note that if 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.

2.5. Example.

To illustrate the concepts of fragments, the read-access graph, and propagation functions, consider the following example of a simple Bakunin data network. Suppose we have an airline reservation database (which has typically been used to illustrate high availability replicated data mechanisms). 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 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 cutoff form the rest of the system.

Note that 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.

The database is divided into six fragments. Each node has a copy of every fragment. The fragments are: 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.

Each transaction executes at a single node and can only update the fragment controlled locally. However, a transaction is allowed to read the local copies of other fragments, without requesting any "locks" at other nodes. (A local concurrency control mechanism ensures that local executions are serializable.)

Figure 2.2 presents the RAG for our running example. 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.

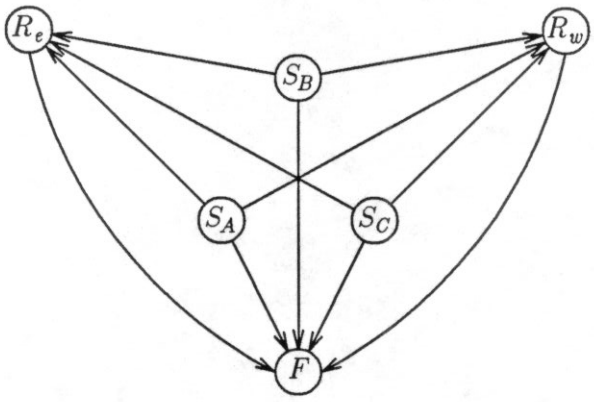


Figure 2.2. Airline Reservation Example.

Recall that our goal is for nodes to be autonomous. Hence, when node $N(R_e)$ wants to make a reservation, it reads R_e and the local copy of F. The copy of R_e is the most up to date since all updates to this fragmant originate at this node; however, the copy of F may be out of date if $N(F)$ has been unable to communicate with $N(R_e)$. This will not stop the reservation transaction from executing. It will read the latest value of F that it has and will proceed to update R_e. The new values generated will

then be propagated to the other nodes.

To illustrate how the propagation functions direct the flow of updates in the network, suppose that

$$S(F) = \{R_e,\ R_w\}$$
$$S(R_w) = \{S_A,\ S_B,\ S_C\}$$
$$S(R_e) = \{S_A,\ S_B,\ S_C\}$$

Then in order to get from $N(F)$ to $N(S_B)$, for example, the updates to fragment F have to go through (and be installed at) either $N(R_e)$ or $N(R_w)$. (We do not show in this chapter how propagation functions are used to guarantee correct global schedules of transactions.)

2.6. Trade-offs for Bakunin Data Networks.

In the previous sections of this chapter we introduced the basic conceptual components of Bakunin data networks — a model for designing highly available replicated databases. Given the assumptions of one-to-one correspondence between fragments and sites, full data replication, and stationary agents, a Bakunin network is fully defined by its RAG and propagation functions. The RAG represents restrictions on access patterns for transactions. The propagation functions represent restrictions on logical propagation paths that updates can take. Together, when properly chosen, the RAG and propagation functions can guarantee the specified correctness criteria for the system without compromising data availability. In the next three chapters, various combinations of RAGs and propagation functions will be studied that guarantee correctness porperties ranging from the traditional one-copy serializability to less strict properties motivated and defined in Chapters 4 and 5.

Maintaining high levels of data availability never comes for free, no matter what approach is applied. The Bakunin networks model is no exception. Unlike other known

approaches, however, Bakunin networks compromise neither availability nor correctness. Instead, we sacrifice the generality of transaction types and, in some cases, promptness of update delivery to remote sites. That immediately points to the kind of applications best suited for this approach. Such applications would have a limited number of well categorized transaction types. Examples include (but are not limited to) airline reservations systems, banking databases, and inventory databases.

It would not be difficult to design a Bakunin data network that can guarantee the desired correctness properties if we were willing to settle for severely restricted RAGs and update propagation functions. The challenge is not to give away more flexibility than necessary in order to achieve the goals of availability and correctness. It is intuitively clear, and will be demonstrated in succeding chapters, that the more restrictive the propagation protocol (functions), the less restrictive the RAG has to be, and vice versa. This is the fundamental trade-off exhibited by Bakunin data networks.

If an application's needs can be satisfied by one of the less strict correctness properties that we shall introduce, then we can even further relax restrictions on the read-access patterns and/or update propagation protocols. For example, let B be a Bakunin data network with a RAG of type X and propagation functions R and S, such that B is known to guarantee serializability. Then it is likely to be the case that if virtual serializability (see Chapter 4) is our goal, it can be satisfied by Bakunin network B' with the same propagation functions S and R, but with a RAG of a more general, and therefore less restrictive, type X'. Thus, the ability to choose from several meaningful correctness criteria will add another dimension to the trade-off space of our model, thereby adding more flexibility.

Bakunin data networks can be viewed as a framework for designing highly available replicated databases, rather than just another approach to coping with communi-

cation failures and network partitions. Such view is justified by the fact that this model provides a large number of specific options, each of which is characterized by a different trade-off instance out of a wide scope of possibilities.

CHAPTER 3

PRESERVING ONE-COPY SERIALIZABILITY

One-copy serializability is an important general correctness criterion for replicated databases. It provides transparency with respect to both concurrency of transaction processing and data replication. In other words, it always appears to the users of the database system that transactions run serially and that there is a single copy of every data item.

The goal of this chapter will be to study Bakunin data networks that guarantee one-copy serializability.

Before we proceed the following simple result is established that will facilitate the presentation of the material in the rest of this chapter. A *missing write* refers to a failure to record an update on one of the copies of a data item (as in the missing writes algorithm [Eage83a]).

Theorem 3.1. A serializable transaction schedule on replicated data in which there are no missing writes is also one-copy serializable.[†]

Proof. Let S be the schedule in question. Since S is serializable, it is equivalent to a serial schedule $S' = T_1 T_2 \cdots T_r$, i.e., S' is a concatenation of r single-transaction schedules each of which includes write actions on every existing copy of every data item being updated. Because of the absence of missing writes, every reference in S' to any x_J (the copy of item x at node J) can be replaced by x (representing a logical operation on item x), which yields a one-copy serial schedule. Hence S is one-copy serializable. □

[†] See [Bern87a] for the definitions of serializable and one-copy serializable schedules on replicated data.

In Bakunin data networks every transaction updating a data item eventually updates every existing copy of it by propagating update messages to remote nodes. Thus, by Theorem 3.1, the notions of serializability and one-copy serializability are equivalent in our model. It will, therefore, be sufficient to concentrate on serializability.

3.1. Cycles in RAGs.

We begin with a fairly simple observation that the RAG must be acyclic if we want to guarantee serializability at all times, even during communication failures. To show the need for this, consider the example of Figure 3.1. Suppose that a partition occurs and leaves each node in complete isolation. If the nodes continue processing transactions — and that is what Bakunin networks are supposed to do — the transaction 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 (the original value of) 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.

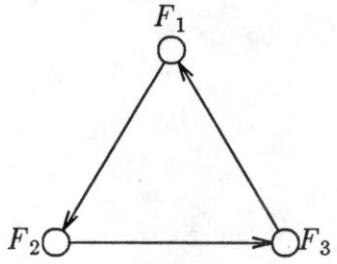


Figure 3.1. RAG.

Consequently, all the systems that we shall study in this chapter will be characterized by acyclic RAGs only. Note that the airline reservations example in the previous chapter had an acyclic RAG (see Figure 2.1).

3.2. Acyclic RAGs: A General Case.

In this section we exhibit a propagation protocol that is sufficient to guarantee serializability for arbitrary acyclic RAGs. Let $G = (V, E)$ be an acyclic RAG. Suppose that ord: $V \rightarrow \{1, ..., n\}$ 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, then $ord(i) < ord(j)$ [Mehl84a]. (A topological order must exist if G is acyclic.) Then define functions R and S as follows:

$$R(i)=\begin{cases}\{ord^{-1}(ord(i)+1)\} & \text{if } 1\leq ord(i)<n \\ \varnothing & \text{if } ord(i)=n\end{cases} \tag{3.1a}$$

$$S(i)=\begin{cases}\{ord^{-1}(ord(i)-1)\} & \text{if } 1<ord(i)\leq n \\ \varnothing & \text{if } ord(i)=1\end{cases} \tag{3.1b}$$

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:

$$R(i) = \begin{cases}\{i + 1\} & \text{if } 1 \leq i < n \\ \varnothing & \text{if } i = n\end{cases} \tag{3.2a}$$

$$S(i) = \begin{cases}\{i - 1\} & \text{if } 1 < i \leq n \\ \varnothing & \text{if } i = 1\end{cases} \tag{3.2b}$$

Definition 3.1. Let $l \in V$ be such that $ord(l) = 1$ (with our simplifying assumption, $l = 1$). We define relation $<$ as follows. 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 3.1. 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 3.2. Let R and S be the propagation functions defined in (3.2a) and (3.2b). 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, $1 \leq q \leq n$. 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 $1 \leq q < r \leq n$. 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 3.1 and 3.2 together establish the following theorem.

Theorem 3.2. Let R and S be the propagation functions defined in (3.2a) and (3.2b). 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 ordering.

3.3. A Special Case: tf-RAGs.

The topological protocol of Section 3.2 may be too restrictive in some cases. In particular, if the RAG is a tree, possibly with forward edges [Aho74a], then a protocol that propagates updates up the tree can be constructed. It performs better than the topological protocol and still guarantees serializability.

Definition 3.2. 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.*

Figure 3.2(a) shows an example of a tf-graph. The nodes are numbered in a DFS order.

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 [Aho74a]. 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 define the propagation functions as follows:

$$R(i) = \{j: j \text{ is a child of } i \text{ in } D\} \tag{3.3a}$$

$$S(i) = \{j: j \text{ is the parent of } i \text{ in } D\} \tag{3.3b}$$

Definition 3.3. 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 3.3. 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 3.1. □

Lemma 3.4. Let G be a tf-RAG. Let R and S be the propagation functions defined in (3.3a) and (3.3b). Let (T_i , T_j) be an edge in the global serialization 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 $U(T_j)$.

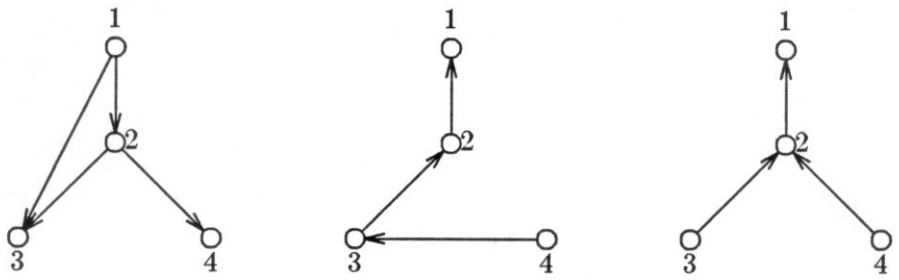
Case 3. (w, v) is an edge in D. Similar to Case 2.

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

Theorem 3.3. Let $G = (V, E)$ be a tf-RAG. Let R and S be the propagation functions defined in (3.3a) and (3.3b). Then any execution characterized by G, R and S is serializable.

Proof. Follows trivially from Lemmas 3.4 and 3.3. □

To illustrate the advantage of the protocol presented in this section over the topological protocol, consider the example of Figure 3.2. We are interested in the length of characteristic paths (*lcp*) between four pairs of nodes (one for very edge in the RAG of Figure 3.2(a)). From Figures 3.2(b) and 3.2(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. Consider, for example, how updates produced by node 4 get to node 2. In Figure 3.2(b) they have to flow through node 3, whereas in Figure 3.2(c) they can be sent directly to 2. If a partition occurs that leaves 3 isolated from both 2 and 4, then the topological protocol cannot enact delivery of the



lcp(2,1) = 1
lcp(3,1) = 2
lcp(3,2) = 1
lcp(4,2) = 2

lcp(2,1) = 1
lcp(3,1) = 2
lcp(3,2) = 1
lcp(4,2) = 1

(a) (b) (c)

Figure 3.2. (a) RAG; (b) propagation graph for topological protocol; (c) propagation graph for tf-protocol.

updates to 2 until 3 is reconnected, even though 2 and 4 might still be able to communicate directly with each other. The tf-propagation, on the other hand, avoids this situation. 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).

3.4. A Special Case: Loopless RAGs.

For some RAGs we can devise an even more efficient protocol than the one in Section 3.3. These RAGs 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, and even may appear intuitive, the proof is substantially harder than the ones presented above.

Definition 3.4. Let $G = (V, E)$ be a *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 [Eswa76a].

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

Definitions 3.4 and 3.5 formalize the workings of the local concurrency control mechanisms at every computer node. Note that rule (ii) of Definition 3.4 corresponds to the requirement that updates from nonlocal transactions be installed atomically and in the order in which they arrive.

Definition 3.6. 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) in G *spans* edge (v, w) or (w, v), whichever is present in the RAG.†

Definition 3.7. Let $G = (V, E)$ be a (global) serialization graph and let C be a

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

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 3.8. 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 3.4. 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 3.3, $u = 3$. 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.

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

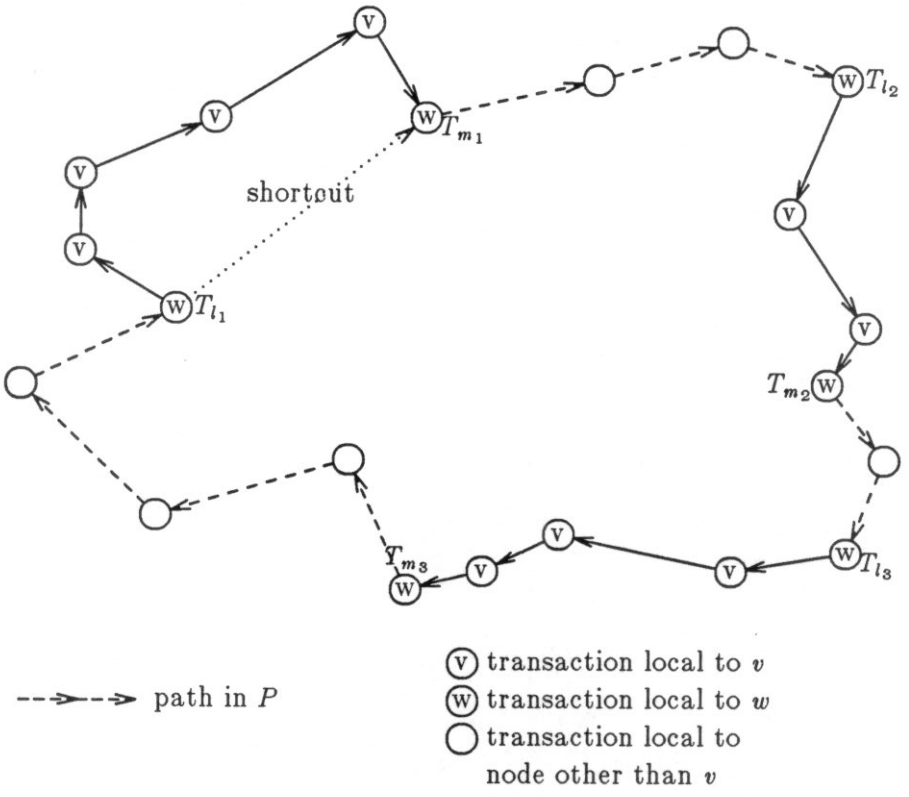


Figure 3.3. Cycle C ($u = 3$).

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 RAGs 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.

3.5. The Extended Topological Protocol.

In this subsection we combine the results of Sections 3.1 and 3.2 to derive a more powerful propagation protocol for the general case of acyclic RAGs. 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 [Chri75a]. 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:

$$R_1(v) = \{ord_i^{-1}(ord_i(v)+1)\} \bigcup \{w: w \in V-V_L \text{ and } (v,w) \in E\}$$
$$R_2(v) = \{w: w \in V-V_L \text{ and } (v,w) \in E\}$$
$$R_3(v) = \{w: w \in V \text{ and } (v,w) \in E\}$$

$$R(v) = \begin{cases} R_1(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v)<k_i \\ R_2(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v)=k_i \\ R_3(v) & \text{if } v \in V-V_L \end{cases} \tag{3.4a}$$

$$S_1(v) = \{ord_i^{-1}(ord_i(v)-1)\} \bigcup \{w: w \in V-V_L \text{ and } (v,w) \in E\}$$
$$S_2(v) = \{w: w \in V-V_L \text{ and } (w,v) \in E\}$$
$$S_3(v) = \{w: w \in V \text{ and } (w,v) \in E\}$$

$$S(v) = \begin{cases} S_1(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v)>1 \\ S_2(v) & \text{if } v \in V_{L_i} \text{ and } ord_i(v)=1 \\ S_3(v) & \text{if } v \in V-V_L \end{cases} \tag{3.4b}$$

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 3.1 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, $1 \leq r \leq m$, and $U(T_i)$ is installed at node $ord_r^{-1}(1)$ before $U(T_j)$.

Lemma 3.5. If functions R and S are defined as in (3.4a) and (3.4b), 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, $1 \leq r \leq m$, we have $T_i < T_j$.

Proof. The proof is very similar to that of Lemma 3.2, 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 3.6. 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 3.4, a contradiction. □

Theorem 3.5. An execution characterized by an acyclic RAG $G = (V, E)$ and the propagation functions of (3.4a) and (3.4b) 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 3.6 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, $1 \leq i \leq s$. 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, $1 \leq i \leq s$. 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 3.4). Therefore we end up with a new cycle in the serialization graph that is based 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. □

While it is obvious that the extended topological protocol is, in general, superior to the topological protocol, the relationship between the former and the tf-protocol is not as obvious. To compare them, consider the RAG in Figure 3.4(a). This is a tf-graph. Therefore, the tf-protocol is applicable. The resulting propagation graph is shown in

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

Figure 3.4(c). If, however, we were to apply the extended topological protocol, we would get the propagation graph of Figure 3.4(b). (Note that in this case there is no difference between the extended topological and topological protocols since every edge of the RAG lies on a loop.) We can see that the tf-protocol does better in this example. In general, the tf-protocol always propagates along every tree edge. That is the best that the extended topological protocol can ever achieve because propagating along a forward edge would violate the reverse topological order (by skipping a number of intermediate nodes). We conclude from the above that for tf-graphs the tf-protocol is superior to the extended topological protocol.

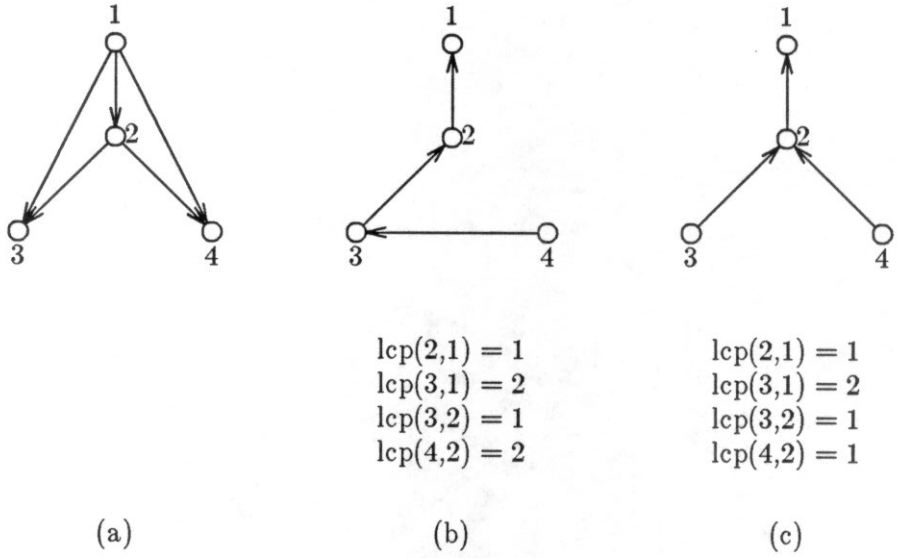


Figure 3.4. (a) RAG; (b) propagation graph for extended topological protocol; (c) propagation graph for tf-protocol.

3.6. Summary.

In this chapter we have exhibited four types of Bakunin data networks that preserve one-copy serializability of transaction schedules. Each one of the four is characterized by a specific trade-off between read-access restrictions and expediency of update propagations. The Bakunin data network types in this chapter are designated

$S1$ trough $S4$, as follows (S stands for serializability):

$S1$: *Acyclic RAG + Topological Protocol.* This type of Bakunin data network is described by an arbitrary acyclic RAG and the propogation functions that define what has been called the topological propagation protocol. A topological order is imposed on the RAG, and the nodes send their updates down along the topological "chain," i.e., from successor to predecessor. The protocol is relatively simple and easy to describe.

$S2$: *Acyclic RAG + Extended Topological Protocol.* The extended topological protocol differs from the protocol of $S1$ in that instead of defining a single topological order for all the nodes of the RAG, a separate topological order is defined for every weakly connected component of the graph. In addition to propagating updates along topological chains in every component, updates can also be propagated between two nodes in different components provided the nodes are connected by an edge. In general this protocol yields a better propagation graph than the topological protocol.

$S3$: *tf-RAG + tf-Protocol.* Bakunin data networks characterized by tf-RAGs, a special form of acyclic RAGs that can be represented as trees with forward edges, can guarantee one-copy serializability if coupled with the protocol that allows propagation of updates along every tree edge in the tree representation of the graph.

$S4$: *Loopless RAG + Unrestricted Protocol.* If RAGs are further restricted not to have any undirected cycles, then an arbitrary choice of propagation functions can guarantee one-copy serializability.

Bakunin data network type $S2$ supersedes type $S1$ in the sense of providing a more expedient propagation protocol for the same type of RAG. Thus $S1$ does not have a practical value, but it was introduced for methodological purposes since it facilitates

the presentation and understanding of $S2$.

Types $S2$, $S3$, and $S4$ are arranged in the order of increasing expediency of update propagation, culminating in the unrestricted protocol of $S4$ that introduces no artificial impediments for propagation paths in the network. At the same time, the allowable read-access patterns for transactions grow progressively restrictive from $S2$ to $S4$, starting with arbitrary acyclic RAGs and ending with loopless RAGs.

CHAPTER 4

VIRTUAL SERIALIZABILITY

4.1 Alternative Correctness Criteria.

One-copy serializability is a very powerful correctness criterion for replicated databases. Just like regular serializability for nonreplicated databases, it is intuitive, general, and semantically independent. Unfortunately, sometimes it may be too expensive to maintain. For it relies either on exchanges of synchronizing messages among nodes, which may imply high communication cost, long delays, and possible loss of availability during failures, or (as was the case in Chapter 3) on restrictions on transaction classes and update propagation paths, which some applications may find too severe. Advantages as well as the shortcomings of serializability (and one-copy serializability) have been well understood for a long time. Some alternative correctness criteria have been suggested. Most notable among these are the notion of degrees of consistency introduced by Gray et al. [Gray76a], weak serializability for read-only transactions by Garcia-Molina and Wiederhold [Garc82a], multilevel atomicity by Lynch [Lync83a], and predicate-oriented criteria by Korth and Speegle [Kort88a].

Alternatives to serializability are usually less stringent and cheaper to maintain. On the other hand, because they are also less general their applicability might be limited.

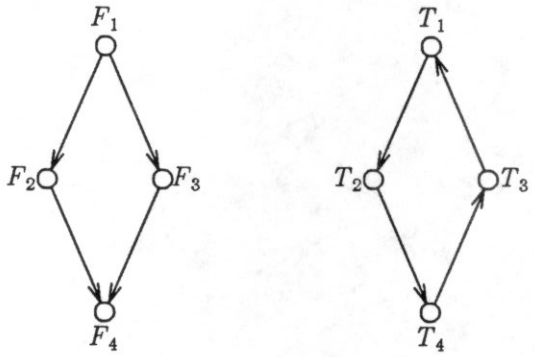
For Bakunin data networks, relaxation of correctness criteria could mean improved trade-offs between access restrictions and update propagation constraints. In particular, a relaxed criterion might allow more expedient propagation for the same type of access restrictions. In the search for alternative criteria in the context of Bakunin data networks, the following two basic guidelines should be used. First, the

alternative criterion must be meaningful at least for some important applications Second, it must have a positive effect on the trade-offs above.

In this chapter we introduce a new correctness criterion for transaction processing in replicated databases, called *virtual serializability.* It is less strict than one-copy serializability, i.e., all one-copy serializable schedules are also virtually serializable, but some virtually serializable ones are not one-copy serializable.

4.2. Motivation and Definition.

Before giving a formal definition of the property, we will try to intuitively motivate it. Note that among all non-serializable schedules, there are some that cannot be identified as such by any legally allowable transaction, i.e., transaction obeying the given RAG. 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 4.1.

To illustrate let us consider some examples. Figure 4.1 shows a RAG and a corresponding serialization graph with a cycle that could be produced in the following way (assuming that the protocols of Chapter 3 were not followed). Transaction T_4 updates fragment F_4 and propagates the new values to nodes 2 and 3. T_2 executes

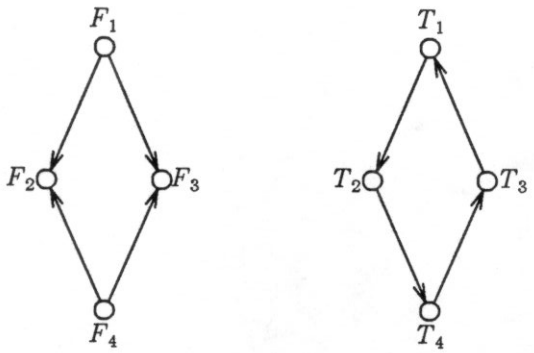
before the updates from T_4 reach node 2. T_2 reads (the old version of) F_4 and udpates F_2. T_3, on the other hand, executes after its home node receives the updates generated by T_4. T_3 reads (the new version of) F_4 and updates F_3. At node 1 the sequence of events is as folllows. Updates generated by T_3 are received; T_1 runs, reading F_2 and F_3 and writing F_1; updates from T_2 are received. Thus T_1 gets to read the new version of F_3 but the old version of F_2.

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 ran.

In the example of Figure 4.2, however, the nonserializable behavior cannot be seen by any of the transactions. The cycle in the serialization graph is produced as follows. Transactions T_2 and T_3 update fragments F_2 and F_3 respectively and propagate the new values to nodes 1 and 4. At node 1, transaction T_1 runs after the arrival of the update from node 3 but before the arrival of the update from node 2. T_2 reads (the new version of) F_3 and (the old version) of F_2 before updating F_1. At node 4, the arrival order of updates from nodes 2 and 3 is reversed, and as a result, transaction T_4 reads the old version of F_3 but the new version of F_2 before updating F_4.

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 example). Thus neither node 1 nor 4 can detect the anomaly. Moreover, the cycle in Figure 4.2(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, at least as far as any transactions are concerned. It might, of

course, be possible for a user to query F_1 and F_4 (one at a time) and recognize the data in them as mutually inconsistent. However, this is quite different from the situation wherein a transaction is allowed to read mutually inconsistent data. For in the latter case the problems could grow if based on inconsistent values it reads the transaction produces inconsistent updates.



(a) *RAG* (b) Nonobservable cycle

Figure 4.2.

Definition 4.1. 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 [Bern79a].)

The cycle in Figure 4.1(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 4.2 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 4.3). 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.

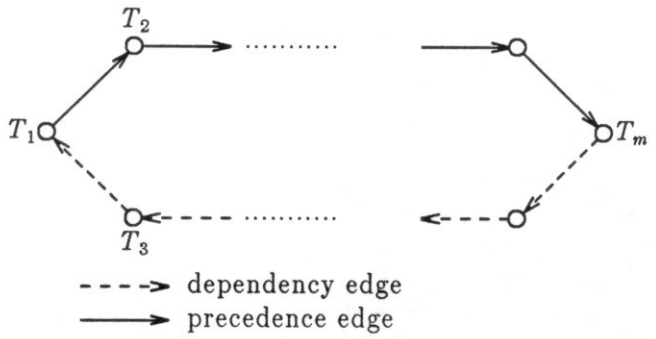


Figure 4.3. Observable cycle. Transaction T_i is local to $N(F_i)$, for all i.

Definition 4.2. A transaction schedule is called *virtually serializable* if the corresponding serialization graph has no observable cycles.

4.3. Applicability.

Before studying Bakunin data networks that guarantee virtual serializability for all legal schedules (Section 4.4 and 4.5), we examine possible use and applicability of this new correctness criterion through an example.

Consider a hypothetical (simplified) banking database. It consists of four fragments. Suppose that there are only two types of accounts that customers can maintain: money market and certificate of deposit. Fragments *MM* and *CD* contain the current money market and CD interest rates that the bank offers, repsectively. Fragments *EC* and *WC* have the information on the balances of the customer accounts (both money market and CD) in the East Coast and West Coast branches of the bank, respectively. As before, each fragment is controlled by a unique node.

The following types of activity can take place in this database. The new money market rate is recorded in fragment *MM* by node $N(MM)$. This is done by transaction T_{MM}. Similarly, the new CD rate is recorded by T_{CD} at $N(CD)$. Account balances in the two branches are periodically updated to reflect accrual of interest based on the appropriate rate. We assume that all accounts in the same branch are updated by the same transaction. Thus, transaction T_{EC} reads the current rates from *MM* and *CD* and updates the balances of the accounts stored in *EC* (money market accounts using the money market rate, and CD accounts using the CD rate). Transaction T_{WC} performs the same activity for fragment *WC*. The RAG in Figure 4.4 defines the read access patterns of the transactions. Of course, there can be other types of activity in our database, such as withdrawing and depositing funds to the customer accounts, reading the balances, reading the rates, etc. These will not be of interest to us in this example, however, because they do not interact with nonlocal transactions (by issuing conflicting read or write operations).

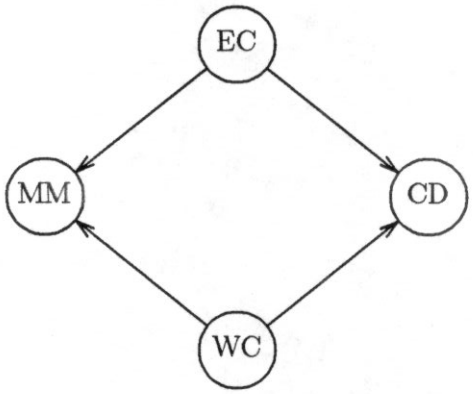


Figure 4.4. RAG for the banking application.

Consider the following sequence of events in this distributed database. Both the money market and CD rates are updated by T_{MM} and T_{CD}, respectively. The updates are propagated throughout the network (assume for now that no restrictions are placed

on propagation paths) and eventually reach $N(MM)$ and $N(CD)$. $U(T_{CD})$ reaches $N(EC)$ in time for transaction T_{EC} to read the new value of the CD rate, but arrives at $N(WC)$ only after T_{WC} has been executed there. The opposite happens to $U(T_{MM})$, i.e., it arrives at $N(WC)$ before the running of T_{WC} but at $N(EC)$, after the running of T_{EC}. Based on this schedule the corresponding serialization graph is shown in Figure 4.5. It has a cycle, therefore the schedule is nonserializable. But the cycle is not observable since the two dependency edges, (T_{CD}, T_{EC}) and (T_{MM}, T_{WC}), are interspersed with the two precedence edges, (T_{EC}, T_{MM}) and (T_{WC}, T_{CD}), and therefore, Definition 4.1 cannot be satisfied. Thus, the schedule described above is virtually serializable.

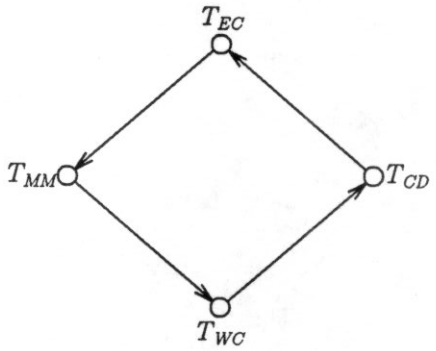


Figure 4.5. Serialization graph for the virtually serializable schedule.

The fact that no transaction can read both fragments EC and WC indicates that there is a degree of independence between them. This may imply that certain kinds of mutual inconsistencies can be tolerated. Let us examine the anomaly in the perceived order of updates above more closely. The anomaly implies that the amount of interest added to the money market accounts in the East Coast branch of the bank is computed on the basis of the old money market rate, while the interest for the CD account is computed on the basis of the new CD rate. The opposite is true for the accounts in the West Coast branch. This may entail a degree of "unfairness" to either the East or West Coast customers, depending on how exactly the rates changed. However, because

of the high fluidity of rates and probable high frequency of interest compounding, it may be too costly (and therefore undesirable) to ensure that total fairness is always maintained, i.e., that the interest compounding transactions (T_{EC} and T_{WC}) always see the same order of changes in the interest rates. Moreover, even serializable schedules do not guarantee that. For example, consider the schedule defined by the serialization graph of Figure 4.6. In this schedule, T_{EC} operates with the old rates for both types of accounts, while T_{WC} operates with the new rates for both. It is true, of course, that this schedule, unlike the one of Figure 4.5, enforces a serial order on the transactions, but since it does not guarantee fairness either (e.g., the new interest rates might be higher), it is not obvious at all what its advantages are over the other. Thus, if the bank deems total fairness absolutely essential, it would require a mechanism separate from serializability for enforcing it. On the other hand, if occasional "unfairness" can be tolerated, than having virtual serializablity instead of serializability does not appear detrimental.

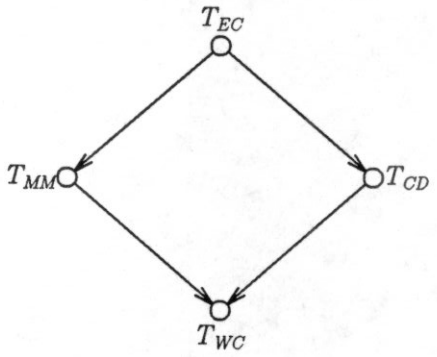


Figure 4.6. Serialization graph for the serializable schedule.

One should bear in mind that the example presented in this section is necessarily simplified and is not intended to convince the reader that virtual serialzability always makes sense. But, while further research is needed to determine the full importance of this correctness criterion, it seems evident that in some cases it can present an interesting and viable alternative to serializability.

4.4. A Protocol for a Restricted Class of Acyclic RAGs.

While the previous section was intended to demonstrate the possible applicability of virtual serializability as a correctness criterion for some database systems, the next two sections study the Bakunin data networks that guarantee this property.

Definition 4.3. Let $L = (V_L, E_L)$ be a RAG loop (see Definition 3.8). 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 4.4. L is a *single-source* (*single-sink*) loop if it contains exactly one source (sink). It is a *multi-source* (*multi-sink*) loop otherwise.

Theorem 4.1. (Acyclic) RAGs 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 4.1 because the direction of edges along the loop changes more than twice (see Figure 4.2(a)). Therefore C is not observable. □

Since cycles in RAGs 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 RAGs, just as we did in Chapter 4.

Replacing serializability with virtual serializability as correctness criterion helps make update propagation less restrictive. For RAGs with no single-source loops, for instance, update propagation 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.

4.5. A Protocol for Arbitrary Acyclic RAGs.

For general acyclic RAGs, 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:

$$R_1(i) = \{ord^{-1}(ord(i) + 1)\} \bigcup \{j: j \in V - V_S \text{ and } (i, j) \in E\}$$
$$R_2(i) = \{j: j \in V - V_S \text{ and } (i, j) \in E\}$$
$$R_3(i) = \{j: j \in V \text{ and } (i, j) \in E\}$$

$$R(i) = \begin{cases} R_1(i) & \text{if } i \in V_S \text{ and } ord(i) < k \\ R_2(i) & \text{if } i \in V_S \text{ and } ord(i) = k \\ R_3(i) & \text{if } i \in V - V_S \end{cases} \tag{4.1a}$$

$$S_1(i) = \{ord^{-1}(ord(i) - 1)\} \bigcup \{j: j \in V - V_S \text{ and } (j, i) \in E\}$$
$$S_2(i) = \{j: j \in V - V_S \text{ and } (j, i) \in E\}$$
$$S_3(i) = \{j: j \in V \text{ and } (j, i) \in E\}$$

$$S(i) = \begin{cases} S_1(i) & \text{if } i \in V_S \text{ and } ord(i) > 1 \\ S_2(i) & \text{if } i \in V_S \text{ and } ord(i) = 1 \\ S_3(i) & \text{if } i \in V - V_S \end{cases} \tag{4.1b}$$

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 4.2. Propagation functions R and S defined in (4.1a) and (4.1b) guarantee virtual serializability for schedules characterized by arbitrary acyclic RAGs.

Proof. Suppose this is not true. Let C be an observable cycle in the global serialization graph. From Definition 4.1 and the observation in the proof of Theorem 4.1, 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 (4.1a) and (4.1b), engage in a topological propagation. In the absence of nodes outside G_S, Lemma 3.2 (and hence Theorem 3.1) 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 3.2. 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 2.4.) 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. □

4.6. Summary.

In this chapter we have introduced a new correctness criterion for replicated databases — virtual serializability. Virtual serializability is less strict than one-copy serializability — the traditional criterion for replicated databases — yet strict enough to be a satisfactory alternative to the latter in some cases.

Relaxation of correctness requirements enables us to introduce Bakunin data networks with better trade-off properties than their counterparts for the case of one-copy serializability. Namely, two types of Bakunin data networks have been studied in this chapter (V stands for virtual serializability):

$V1$: *Acyclic RAGs + Single-Source Topological Protocol.* This type of Bakunin data network is described by an arbitrary acyclic RAG and the propagation functions defined in (4.1a) and (4.1b). The resulting protocol is refered to as the single-source topological protocol because it enacts strict topological propagation for the "dangerous" parts of the RAG — single-source loops; in the rest of the RAG propagation along every edge can be employed.

$V2$: *Acyclic RAGs with no Single-Source Loops + Unrestricted Propagation.* In Bakunin data networks of this type no restrictions on propagation paths are placed. A restricted class of RAGs is required, however.

Compared to type $V1$, type $V2$ exhibits a superior expediency of update propagation but a more restrictive read-access pattern for transactions.

In Chapter 3 we introduced Bakunin networks of type $S2$ (acyclic RAGs +

extended topological protocol), which guaranteed one-copy serializability. With the same class of RAGs but with a less strict correctness criterion, type $V1$ of this chapter provides a better expediency of propagation. This is not hard to see if we compare the corresponding protocols. The two are similar in that they excersize special care only for a "portion" of the corresponding RAG. In the case of $S2$, this is the union of all weakly connected components of the RAG. In the case of $V1$, this is all the edges that lie on a single-source loop. It is easy to see that the latter is a subgraph of the former for any RAG. Therefore, for Bakunin data networks of type $V1$ the special-care portion of the RAG is no greater than for $S2$. Thus, in general, $V1$ has to exhibit less restrain in update propagation than $S2$.

Among the Bakunin data networks that preserve one-copy serializability, the counterpart of type $V2$ is $S4$. Like $V2$, $S4$ allows unrestricted propagation. However, because it has to satisfy a stricter criterion of correctness, $S4$ can only have a RAG with no loops. Clearly this is a subclass of RAGs with no single-source loops allowed by $V2$.

CHAPTER 5

FRAGMENTWISE SERIALIZABILITY

In this chapter the exploration of alternatives to one-copy serializability continues. We motivate and define *fragmentwise serializability.* This correctness criterion is even less strict than virtual serializability of Chapter 4, i.e., the set of all fragmentwise serializable schedules is a proper subset of the set of virtually serializable schedules.

5.1. Unrestricted Bakunin Data Networks.

We will concentrate our attention on the following question. Is there anything useful about the properties of transaction schedules that arise in unrestricted Bakunin data networks? An unrestricted Bakunin network is described by an arbitrary RAG (possibly containing directed cycles) and unrestricted propagation functions, i.e., functions allowing propagation of updates along every edge of the RAG.

First of all, it is not hard to see that any Bakunin network guarantees mutual consistency of replicas of the same data (in the dynamic sense). Let us consider a fragment F_i. Since all updates to F_i originate at $N(F_i)$ and are installed eventually in all remote copies of F_i in the same order, all copies of fragment F_i will be identical after the system becomes quiescent. To be precise, if at time t the processing of new transactions is halted, and it takes time δt for all updates to propagate throughout the network, the copies will become identical at time $t + \delta t$.

Let us now consider the subset of transactions that update the contents of fragment F_i. We denote this subset of transactions $W(F_i)$. Thus, a transaction $T \in W(F_i)$ if and only if T updates F_i.

Definition 5.1. Let A be a set of transactions. Let S be a schedule of transactions

in A. Let B be a subset of A. Let P be the schedule that is obtained from S by deleting all actions pertaining to transactions in $A - B$. P is called the *projection* of S on B.

Definition 5.2. Let S be a transaction schedule. S is said to be *fragmentwise serializable* if and only if (1) the projection of S on $W(F_i)$, for any i, is serializable; and (2) no transaction in S that reads the contents of F_i, for any i, ever sees a partial effect of a transaction in $W(F_i)$.

The following theorem states that all Bakunin data networks guarantee fragmentwise serializability.

Theorem 5.1. An arbitrary acyclic RAG in combination with unrestricted propagation functions guarantee fragmentwise serializable transaction execution.

Proof. First, we prove that condition (1) of Definition 5.2 is satisfied. All transactions in $W(F_i)$ are initiated by the same agent, $A(F_i)$, and, hence, are executed originally at the same node. Therefore, they are subject to the same local concurrency control mechanism, which insures the serializability of their schedule at the originating node. Transactions in $W(F_i)$ generate the lists of updates that are dispatched to other nodes in the network for update propagation. The lists of updates are installed at remote sites in the same serial order in which the transactions that generated them were executed at $N(F_i)$. Thus the schedule is the same at all nodes. (All this is true regardless of the type of RAG and propagation functions used.)

Condition (2) is satisfied in any Bakunin data network due to the fact that every node has a local concurrency control mechanism that ensures a serializable schedule for local transactions and lists of nonlocal updates. □

5.2. The Significance of Fragmentwise Serializability.

To analyze the significance of fragmentwise serializability, let us examine frag-

ment F_i and how it relates to other fragments and the entire database. Fragment F_i enjoys a certain degree of autonomy within the database. This autonomy is manifested on two levels: structural and procedural. On the structural level, fragment F_i represents a semantic unit of information (such as an airline company flight schedules or information on the inventory stock at a local warehouse of a wholesale distributor). On the procedural level, modifications of its content are handled exclusively by the designated agent. Consequently, fragment F_i appears to be a more or less self-contained collection of data. The above notion of autonomy serves as justification for the following conceptual view. Instead of one "large" database divided into several fragments, we have a somewhat different arrangement, namely a group of separate, "small" databases with ongoing communications among them. Update transactions operate on a single small database (a transaction updating fragment F_i is said to be a transaction on database F_i). A read-only transaction that accesses multiple fragments is conceptually replaced by several transactions each of which accesses a corresponding "small" database. As far as a particular database (fragment) F_i is concerned, all other databases (fragments) constitute the "outside world." Thus, when a transaction updating F_i reads from another fragment, the value (values) read can be interpreted as input to this transaction from the outside world. Fragmentwise serializability for the "large" database thus translates into serializability for each constituent database.

To understand the precise nature of inconsistencies that may arise as a result of replacing global serializability with fragmentwise serializability, we turn to the notion of consistency predicates. A predicate $P(v(x_1), ..., v(x_r))$, where x_i, $1 \leq i \leq r$, is a data object and $v(x_i)$ is the value of x_i, is said to be a *single-fragment* predicate if $x_i \in F_j$, for some j and for all $i \in \{1, ..., r\}$; it is a *multi-fragment* predicate otherwise, i.e., a single-fragment predicate spans just one fragment whereas a multi-fragment predicate spans

several fragments. When global serializability is enforced, consistency predicates are never violated. Fragmentwise serializability does not guarantee, in general, that all consistency predicates hold. However, it is an immediate consequence of this correctness criterion that single-fragment predicates are never violated.

Another way to look at fragmentwise serializability is by considering how individual nodes may perceive the same global schedule. Let us consider an example of a fragmentwise serializable schedule that is neither serializable nor virtually serializable. Suppose the database consists of three fragments: F_1, F_2, and F_3. Suppose further that transactions initiated by $A(F_1)$ read from F_1, F_2, and F_3; $A(F_2)$, from F_2 and F_3; and $A(F_3)$, from F_3 only. The corresponding read-access graph is shown in Figure 5.1.

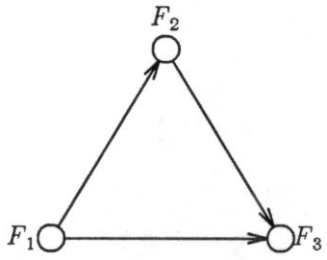


Figure 5.1. RAG.

Let a, b and c be data objects, and let $a \in F_1$, $b \in F_2$, $c \in F_3$. Suppose $A(F_1)$ initiates transaction T_1: $[(T_1, r, c), (T_1, r, b), (T_1, w, a)]$; $A(F_2)$ initiates T_2: $[(T_2, r, c), (T_2, w, b)]$; and $A(F_3)$ initiates T_3: $[(T_3, r, c), (T_3, w, c)]$.[†] Suppose further that update (T_2, w, b) reaches node $N(F_1)$ and is installed in its local copy before action (T_1, r, b) is executed (generating dependency $T_2 \rightarrow T_1$); action (T_1, r, c) is executed before update (T_3, w, c) is installed in the copy of $N(F_1)$ (generating $T_1 \rightarrow T_3$); and finally, (T_3, w, c) is installed at $N(F_2)$ before (T_2, r, c) is executed $(T_3 \rightarrow T_2)$. This

[†] A parenthesized triplet denotes an atomic action, with the first element identifying the transaction it is part of, the second element specifying the type of action (read or write), and the third element identifying the data object on which the action is preformed. The entire expression in brackets denotes the ordered sequence of actions comprising the transaction.

sequence of events can also be presented in a tabular form (time moves down):

T_1	T_2	T_3
READ c		READ c
		WRITE c
	READ c	
	WRITE b	
READ b		
WRITE a		

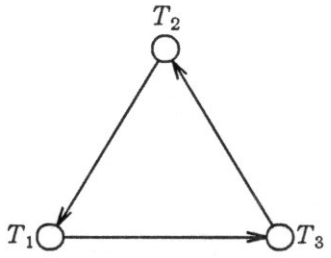


Figure 5.2. Serialization graph.

The above schedule yields a serialization graph with a cycle (Figure 5.2) and, therefore, a non-serializable schedule. Since edges (T_3, T_2) and (T_2, T_1) form a dependency path, and (T_1, T_3) forms a precedence path, the cycle in Figure 5.2 is observable. Hence the schedule is not virtually serializable. But it is fragmentwise serializable, by Theorem 5.1.

In general, some nodes may not be able to observe the entire schedule. In the example above, since $N(F_3)$ is not allowed to read fragments F_1 and F_2, the nonlocal updates received at this node do not conflict with transaction T_3, i.e., T_3's read set and the write sets of transactions T_1 and T_2 do not intersect. Hence, $N(F_3)$ cannot tell what order of execution transactions T_3, T_1, and T_2 followed.

Further, those parts of the schedule that different nodes can observe may not be consistent. Thus, at node $N(F_2)$, the update to F_3 is installed before the local transaction T_2 reads F_3. Therefore, from the point of view of fragment F_2, transaction T_3 preceded T_2. At node $N(F_1)$, however, the serialized order is T_2 followed by T_1 followed by

T_3. So, as far as $N(F_1)$ is concerned, transaction T_2 preceded T_3, and not vice versa.

In general, fragmentwise serializability allows serialized but inconsistent views of the relative order of transactions in the execution schedule, as seen at different nodes in the distributed database system. Note that, unlike virtual serializability, fragmentwise serializability does not mask nonserializable anomalies from the participating transactions. In the example above, when transaction T_1 executes, it reads the old version of fragment F_3 but, at the same time, the new version of F_2, which in turn depends on the new version of F_3.

5.3. Applicability.

As was done in Chapter 4, we resort to an example to illustrate possible applicability of fragmentwise serializability to real database systems. Consider a simplified version of the airline reservations example in Chapter 2, Section 2.5. It has three fragments: F (flight schedules), R (reservations), and S (seat assignments). For simplicity, we assume that flights originate from one airport only (as opposed to three airports in the original example in Chapter 2) and that there is only one travel agency that handles flight reservations (as opposed to two in the original example).

At the node that controls fragment F (the airline headquaters) transactions of type T_F update schedules. These transactions have no need to read the current reservation or seat assignment information.[†] Flight reservations are made by transaction T_R initiated at the node controlled by the travel agency. This transaction will read data from F to verify that the flight for which the reservation has been requested actually

[†] In reality, flight changes are probably made taking into account the demand, which can be measured by the number of reservations. Thus fragment R is relevant to transactions T_F. However, the demand would most likely be gauged by statistical information collected over a relatively long period of time, using data from R. The decision to update schedules would be based on such information. Therefore, updating instructions are coded into T_F unconditionally, so there is still no need for T_F to read outside of fragment F.

exists. Finally, seat assignments are made by the airport node, when the passenger shows up there to board the flight. This is done by transaction T_S, which, apart from the locally controlled fragment (S) has to read fragment F to ascertain the existence of the flight in question, and fragment R to confirm the reservation (presumably a seat assignment is granted only if the passenger has a reservation). The corresponding RAG is shown in Figure 5.3.

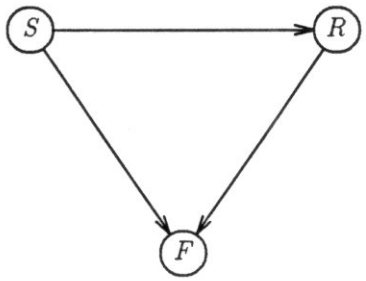


Figure 5.3. RAG for airline reservation example.

Let us consider the following scenario that could occur if we allowed unrestricted update propagation. Suppose a new flight is created, and transaction T_F makes an appropriate entry into fragment F. After receiving and installing this update, transaction T_R reserves a seat on this flight for a customer of the travel agency. This passenger, then, goes to the airport and requests a seat assignment. Transaction T_S is then initiated. Suppose that by this time node S has received the update produced by T_R, but the one produced by T_F still has not arrived (due to some communication problems). The resulting schedule is shown in Figure 5.4. It is easy to see that it is fragmentwise serializable but not virtually serializable.

Transaction T_S gets to observe an inconsistent database state (which could not be the case in any virtually serializable schedule). Since the updates from T_R are reflected in the local copy at $N(S)$, but those from T_F are not, it appears as if a reservation has

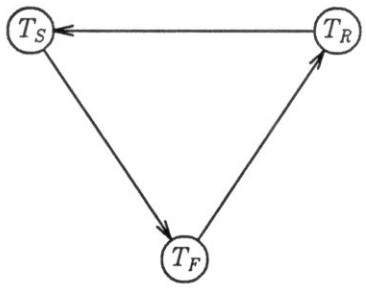


Figure 5.4. Serialization graph for airline reservation example.

been made for a nonexisting flight. Transaction T_S could deal with this type of situation in two different ways. It could either abort and be retried later when, hopefully, the inconsistency has been eliminated; or it could proceed to assign a seat for the passenger on a "nonexisting" flight, under the assumption that the data in the local copy of F is out-of date.

The situation described above might create certain inconveniences for the airline company and, more importantly, for the passengers. However, even one-copy serializability is not a safeguard against this sort of inconveniences. Consider, for example, the above scenario with the creation of a new flight, but assume also that the network partitions and $N(S)$, the airport node, becomes isolated before the updates from either $N(R)$ or $N(F)$ can reach it. When the passenger arrives at the airport and requests a seat on the flight for which he thinks he has a reservation, not only does the local computer not show the new flight, but the reservation itself is not shown either.

5.4. Summary.

A new type of Bakunin data network has been introduced in this chapter. Designated $F1$ (F for fragmentwise serializability), it combines *arbitrary* RAGs (cycles are allowed) and *unrestricted propagation* of updates. Clearly, $F1$ is the most powerful Bakunin network, in the sense of having least restrictions on data access patterns and

propagation paths. $F1$ guarantees fragmentwise serializability, the least strict of the correctness criteria discussed in this dissertation.

CHAPTER 6

EXTENSIONS TO THE BASIC FRAMEWORK

In this chapter we discuss some possible generalizations and extensions to the basic Bakunin data network model outlined in Chapter 2 and studied in Chapters 3, 4, and 5. The goal is to present the wide scope of ideas rather than study any particular extension in great detail.

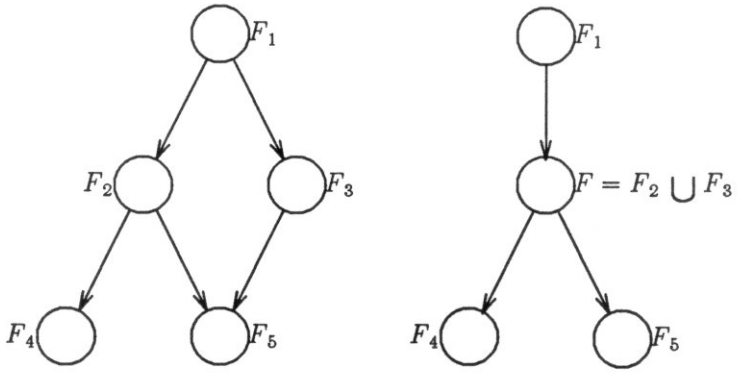
6.1. Nodes Controlling Multiple Fragments.

Until now a one-to-one correspondence between fragments and computer nodes was assumed. However, as was pointed out before, there was no reason for this restriction other than to make the exposition simpler. In this section we consider how to integrate into our model the case of several fragments being controlled by the same node.†

There are two ways to deal with multiple fragments at one node. One way is to replace conceptually the entire collection of such fragments by a new *superfragment* that is the union of them. In particular, this implies that instead of there being several types of transactions, with different access privileges, that could run at the given node, there will be just one type with wider privileges. Namely, suppose that fragments F_1, F_2, ..., F_l are all controlled by node $N = N(F_1) = N(F_2) = \cdots = N(F_l)$. Let G be the RAG of the given Bakunin data network. Then any transaction initiated at N will be allowed to write into superfragment $F = F_1 \bigcup F_2 \bigcup \cdots \bigcup F_l$ and read from any fragment F_j such that (F_i, F_j) is an edge in G for some i, $1 \leq i \leq l$. We could view this as collapsing nodes F_1, F_2, ..., F_l of G into one *supernode* F. Furthermore, agents

† Note that generalization to the case of no fragment at a node is trivial. We can simply assume that the node in question controls an empty fragment.

$A(F_1)$, $A(F_2)$, ..., $A(F_l)$ are replaced by a single agent $A(F)$.



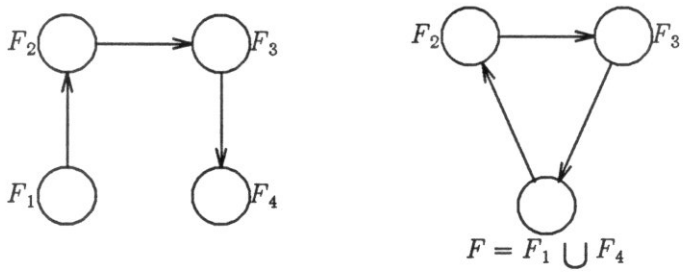
(a) Before (b) After

Figure 6.1. Collapsing two nodes of the RAG into one.

To illustrate consider the RAG shown in Figure 6.1. Suppose that fragments F_2 and F_3 are controlled by the same computer node (Figure 6.1(a)). The result of collapsing the corresponding nodes of the RAG into one is shown in Figure 6.1(b).

Clearly, the technique described above changes transaction access patterns. In the above example, for instance, according to the old RAG (Figure 6.1(a)), a transaction updating fragment F_3 could not read data from F_4. After collapsing nodes F_2 and F_3 into one, however, this becomes possible (see the RAG in Figure 6.1(b)). Changing transaction access patterns may sometimes be undesirable. For example, if the resulting RAG is acyclic, as in Figure 6.2, then one-copy serializability cannot be preserved (see Chapter 3). If that is one of the goals of the system, then such a transformation is clearly unacceptable.

The alternative to the above approach to dealing with multiple fragments at a single node is maintaining logical separation of such fragments (and corresponding agents). Transactions initiated by one agent cannot update fragments controlled by



(a) Before

(b) After

Figure 6.2. Collapsing of nodes introduces a cycle in the RAG.

another. As far as the update propagation protocol is concerned, it should treat the node with multiple fragments as several distinct virtual nodes. Thus, an update sent to the virtual home node of one of the fragments cannot, in general, be shared with other virtual nodes located at the same physical node.

6.2. Distributed Agents.

In Chapter 2 an agent was defined as a process running at the node that controlled a given fragment. The agent was, in fact, a concrete entity for implementing the controlling functions (such as granting a transaction's request to update data in the fragment on its behalf, enforcing the read access restrictions imposed by the RAG, etc.). It was, however, implicit in our scheme that an agent is a local process implemented at a single node.

This notion of agent can be generalized as follows. Suppose that the semantics of the system are such that some fragment cannot be controlled by any single node. This can be the case, for example, if no node in the system can be entrusted with the authority to make unilateral decisions on updating the fragment in question. To enforce such a restriction we have to assign the controlling privileges for this fragment to a consortium of nodes, rather than a single node, as we have normally done throughout this

work. (It is immediately clear that such an arrangement sacrifices a measure of data availability in the face of communication failures since the fragment can be updated only when all the members of the consortium are in the same partition group. Note that this situation is similar to using the technique of weighted voting and assigning the votes in such a way that any majority includes the given subset of nodes.) One way of implementing this is to make the agent — the fragment-controlling process — distributed over the member-nodes of the consortium. Using any of the traditional concurrency control techniques for replicated data (see [Bern81a]) among these nodes, we can make it appear to all the outside nodes as if the control were centralized.

For update propagation purposes this collection of nodes is treated as one, which means that sending updates to the outside nodes should be done in a coordinated fashion (e.g., by electing a unique node and entrusting it with the responsibility for this). Similarly, updates received by one node must be atomically installed at all of the nodes of the group. Hence, commit protocols must be used within the group not only to process transactions that update the locally controlled fragment but also to install updates produced by nodes outside the group. The case of a distributed agent can be viewed as having a number of computer sites simulate one node of the RAG.

6.3. Lifting the Full Replication Assumption.

In Chapter 2 we assumed that data replication in Bakunin networks is complete, i.e., if there are n nodes in the system, then the degree of replication of every data item is n. While full replication is clearly a feasible alternative for relatively small database systems, such as a network routing table database, for instance, it is often prohibitively expensive for large systems. However, the Bakunin network approach can be generalized to deal with replication of arbitrary realms and degrees.

First, let us consider the simplest case of partial replication, namely when the realm of replication of every fragment matches the corresponding RAG exactly. This means that whenever a fragment is allowed to be read by a locally initiated transaction, it turns out that a copy of this fragment is stored at the node in question. Let $G = (V, E)$ be a RAG. Let $RR(F)$ be the realm of replication of fragment F. Then for every $F_i \in V$ and all F_j such that $(F_i, F_j) \in E$, $N(F_i) \in RR(F_j)$.† Let us call this type of partial replication *sufficient* replication.

Clearly, sufficient replication does not necessitate any significant changes in our framework.‡ Since every legal (in the sense of the RAG restrictions) transaction can always find a local copy of a data item it needs to read, node autonomy can be fully preserved. Updates received at a node that does not have a local copy of the data concerned are simply disregarded or forwarded to other nodes specified by the propagation functions.

When the replication is not sufficient, node autonomy is (partially) compromised because a transaction might have to read a nonlocal copy of data if there is no local copy available.†† It is clear that in this type of scenario transactions cannot always run to completion and commit locally. In order to handle nonlocal reads a remote locking mechanism should be implemented. Such a mechanism would request that a lock be set on the copy of the data item located at the node that controls the fragment containing the item. That would insure one-copy (or virtual, or fragmentwise) serializability. We leave out the details of this mechanism, which should be similar to any of the variety of

† This expression reads "node that controls fragment F_i has a copy of fragment F_j."
‡ This is not true if we consider some other extensions discussed below in combination with partial replication. For example, in the case of moving agents (Section 6.4), it may be desirable to have a copy of a fragment at a node even if this fragment is unreadable by any of the agents currently residing at this node. For it is possible that in the future there will be an agent at this node capable of reading the given fragment.
†† Writing data nonlocally will be discussed in Section 6.6.

locking techniques for distributed data (surveyed in [Bern81a]).

Note that curtailed autonomy in the face of insufficient replication is not a drawback of Bakunin data networks but rather an inherent limitation imposed by the absence of full replication.

When full (or sufficient) replication is not feasible or too costly, the logical thing to do is replicate only the most essential and most frequently accessed data. For example, in a database used for air traffic control the information on the current positions of the aircraft in the area might be fully replicated, while the statistical information on the traffic load over a certain period of time could be kept only at one node (or a few nodes). That would lead to relatively infrequent nonlocal reads — clearly a desirable situation. (Compare this policy with the one for page replacement in main memory that tends to keep around the more frequently used pages and flush out to disk those that are not referred to as often.)

6.4. Moving Agents.

Until now we have assumed that agents were stationary, i.e., fixed at their respective nodes. Thus every fragment was always controlled by the same node. In some cases, however, it might be desirable to shift control of a fragment from one node to another. For example, a bank customer would certainly like to be able to use the automatic teller machines at more than just one location. That would require creating an agent (locally) every time the customer correctly identifies himself to the computer by means of a magnetic card, and killing the agent as soon as the card is withdrawn. Note that in this example the card serves as a token.

As another example, consider an airline database where there is a special fragment for seat assignments on a flight. Suppose there is a computer node at every airport and

consider a flight which has stop-overs. (Some passengers can be discharged and others taken at each stop.) Initially the agent for the seat-assignment fragment resides at the airport where the flight originates. It would be desirable, for maximum availability, to make the computer at the airport where the flight is making a stop the current agent for the seat-assignment fragment. That would allow seat assignments to be processed directly at the airport node. Note that in this example the plane can be viewed as a token for the seat-assignment fragment.

Another reason for moving agents is node failure. When the agent's home node goes down, it may be a good policy to regenerate the agent at some other node in the network rather than wait until its home node comes back up.

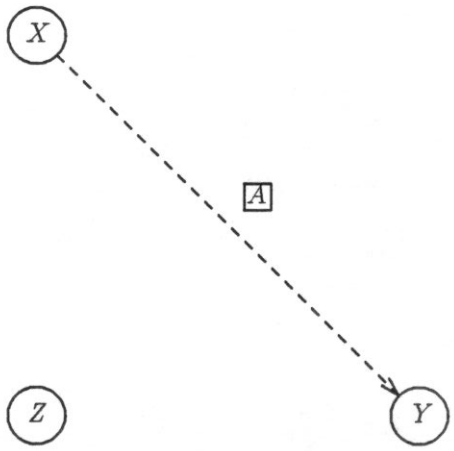


Figure 6.3. Moving agent.

Allowing agents to move, however, may endanger one-copy serializability (or one of the other correctness criteria). Moreover, in some cases, this can even compromise the mutual consistency of replicas. Let A be an agent that decides to move from node X to node Y (Figure 6.3). Let T_1 be the last update transaction initiated by A at X, T_2 the first update transaction initiated by A at Y after the move has been completed. In the absence of any special provisions, it is possible for T_2 to be initiated before $U(T_1)$

has a chance to reach Y (if, for example, there was a break in communications between X and Y). It is also possible for $U(T_2)$ to be received at some other node, say Z, before $U(T_1)$ is received there. It is easy to see that such events may lead to violations of correctness criteria. In both cases, the updates generated by transaction T_1 were delivered late (out of order). Therefore we refer to this as the problem of *missing transactions*. It is impossible, generally speaking, to circumvent this problem without paying a price in availability.

Abandoning the assumption of stationary agents forces us to reaxamine update propagation protocols. They are defined in terms of the corresponding RAGs, which are in turn defined in terms of fragments. But now that the association between fragments and nodes is no longer fixed, the protocols become dynamic in the sense of adjusting the flow of updates in accordance with the current location of relevant agents. To be more specific suppose that $F_i \in S(F_j)$. That means that node $N(F_j)$ is supposed to forward updates to node $X = N(F_i)$. When $A(F_i)$ moves to another node, say Y, $X = N(F_i)$ no longers holds. Instead, we have $Y = N(F_i)$. Therefore, now $N(F_j)$ has to forward updates tp Y instead of X. Thus $N(F_j)$ has to maintain the directory of all agents $A(F_i)$ such that $F_i \in S(F_j)$. We leave out the details of a possible algorithm for performing this task and making sure that the correct sequence of installation of updates is preserved. However, the basic idea of such an algorithm is as follows. Updates that are received at node X and meant for installation at node $N(F_i)$ should be installed at X only if $X = N(F_i)$ currently holds, i.e., if $A(F_i)$ currently resides at X. Otherwise, they should be forwarded to what X believes is the current home of $A(F_i)$. $A(F_i)$ should promptly notify the nodes of the system of its moves.

There are a number of ways to cope with missing transactions. They fall into three main categories. In the first category, certain actions are taken by the system on

a permanent basis, allowing agents to complete their moves "smoothly." The second category provides for some special actions by the system only at the time of the move. Finally, in the third category, agents are allowed to move without any preparatory actions; however, some actions might be taken after the move to rectify possible inconsistencies. There can be a very large number of actual protocols for moving agents. We do not attempt here to be complete, nor do we strive for presenting the most efficient protocols. What follows is just a sample of possibilities, which hopefully gives a flavor of the issues involved.

6.4.1. Permanent Preparatory Actions.

The following method is suitable for those cases when an agent has to leave its home node, for whatever reason, and it is not particularly important exactly which node it is going to move to. This method uses a majority commit protocol. Before a transaction can commit at the agent's home node, the updates it has produced are sent out to the rest of the nodes, and acknowledgments are requested. The transaction commits only after acknowledgments have been received from a majority of the nodes. Then a command is broadcast to commit the updates at remote nodes.† When the agent needs to move, a new home node is selected. The agent must then contact a majority of nodes and request an identifier for all previously executed transactions that updated the fragment. If the new home node had missed any of these, it requests them from the nodes that have them and runs them. This procedure ensures that the home node has seen all transactions previously executed on the fragment. (Each old transaction was seen by a majority of nodes. This guarantees that at least one node in the

† To satisfy the requirments of update propagation we must carry out the broadcast of this command using the propagation protocol. That would insure the proper sequence of update installations. However, the broadcast of updates prior to the transaction commit does not have to be done in strict adherence to the protocol because installation does not occur at that time.

current majority has seen it and gives it to the new home node.) Now the agent is ready to execute new update transactions. The first of these receives the sequence number that follows the last transaction's number, and so on, so that there is a single, uninterrupted sequence of transactions.

It is not difficult to see that any standing correctness criterion (one-copy, virtual, or fragmentwise serializability) is preserved with this method.† However, the communication overhead can be significant, and of course, availability is reduced. Specifically, update transactions can only be processed with the cooperation of a majority group of nodes.

6.4.2. Actions at the Time of the Move.

A. Moving with data.

Let A be the agent that moves from node X to node Y. Let T_1 be the last update transaction initiated by A at X, T_2 the first update transaction at Y. We require that A transport (by any means available) a copy of the fragment stored at X to store it in place of the copy of the fragment at site Y before resuming processing. In addition, all other sites are requested not to install updates from transaction T_2 until those from T_1 have been installed.

The requirement concerning transportation of data by the agent is not as difficult as it may sound at first. Consider the following example. A military command post is to be evacuated to avoid the threat of an enemy attack. The data from the fragment controlled by it can be dumped on tape and transported to a new location. This guarantees that the copy of the fragment at the new command post is now up-to-date. As another example, identification cards with magnetic strips are now becoming common. The

† Provided that the propagation protocol is appropriately adjusted after the move, as discussed in the beginning of Section 6.4.

strips in many cases can carry, not just an identifying code, but also real data (readable *and* writable). For instance, subway cards record the amount of money a user has available; copier cards store the number of copies a person is authorized to make. These cards fit our model exactly. They serve not only as a token for the corresponding fragment but also as a transporter of data contained in that fragment. The procedure of transporting data preserves fragmentwise serializability.

If the transactions that update a fragment must also read it, it is important that the agent has access to the most up-to-date version of the fragment, in order to preserve fragmentwise serializability. However, in those cases when all transactions are write-only, there is no need to transport data. Moreover, when these transactions are commutative (such as incrementing or decrementing some values, or creating new data items), copies of the fragment at different nodes will be mutually consistent regardless of the order in which they receive these updates. Hence, fragmentwise serializability is preserved. See Section 6.7 for examples.

B. Moving with the sequence number.

With this approach, only the sequence number of the last transaction to run at the old home node is given to the new home node. Let A, X, Y, T_1, and T_2 be as above. Before A can execute T_2 at Y, it must receive the sequence number of T_1 from X. This number can either be explicitly requested after the move or can be carried by external means (e.g., in the magnetic strip of a card). Before A executes T_2, it must wait until the updates of all previous transactions performed on the fragment in question are received at Y. New transactions are given sequence numbers that follow that of T_1.

This method, just as moving with data, preserves fragmentwise serializability. It may result, however, in decreased availability due to the waiting for old transactions. On the other hand, it may be easier to implement.

6.4.3. Omitting Preparatory Actions.

In those cases where the previous approaches are infeasible, the only option left is to do the best we can after the move in order to minimize the effects of missing transactions. As before agent A moves from node X to node Y. Let T_1, ..., T_r be the transactions initiated by A at X, and $\mathbf{T}_1$, ..., $\mathbf{T}_s$, the transactions initiated by A at Y, after the move. We assume that it is imperative that A start processing new transactions as soon as it arrives at Y. In such a situation, fragmentwise serializability can be compromised. However, at least we can guarantee that mutual data consistency is preserved, by enacting the following protocol.

Let T_i $(i \leq r)$ be the last transaction from X installed at Y before A initiates $\mathbf{T}_1$. Let Z denote any node in the system different from X and Y. Further, let us assume, that transactions and data items are timestamped.

At node Y:

(1) *Before broadcasting* $\mathbf{T}_1$: Broadcast a special message $\mathbf{M}_0 = (T_1, ..., T_i)$ (containing all transactions from X installed at Y thus far).

(2) *Upon receipt of* $U(T_l)$ *for transaction* T_l $(i < l \leq r)$ *(T_l is a transaction that was missing)*: Remove from $U(T_l)$ those updates that have already been overwritten by more recent transactions. Package the remaining updates into a new transaction $\mathbf{T}_k$, where k is the next sequence number. Install the updates of $\mathbf{T}_k$ locally and send $U(\mathbf{T}_k)$ out. If this missing transaction causes an anomaly that can be detected, then issue the necessary corrective actions. (For example, if after $\mathbf{T}_k$ runs, a flight is overbooked, then cancel one or more reservations.)

At node Z:

(1) *Upon receipt of* $\mathbf{M}_0$: (Let T_j $(j \leq r)$ be the last transaction from X installed at Y). If $j < i$, install transactions T_{j+1}, ..., T_i.

(2) *Upon receipt of* $U(T_q)$ $(max(i, j) < q \leq r)$ *after* $\mathbf{M}_0$ *was received (*T_q *is a missing transaction)*: Do not install $U(T_q)$. Instead, forward it to Y, so it can take corrective actions.

(3) *Upon receipt of* $U(\mathbf{T}_q)$ $(q \geq 1)$: Process it as usual, after installing $U(\mathbf{T}_{q-1})$.

It is not hard to verify that all copies of data will eventually converge. This method for moving agents is somewhat similar to the free-for-all systems, in that it gives high availability (the agent can start processing transactions as soon as it arrives at the new home node), and the correctness criterion is weak. The advantage of this method over free-for-all systems is that all decisions concerning corrective actions for a fragment are centralized (at the current home node of the fragment concerned).

6.5. Read-Only Transactions.

In some cases Bakunin data networks can treat read-only transactions differently from other transactions. In particular read-only transactions might not be required to obey the access pattern restrictions that are imposed by RAGs. These transactions may, then, see inconsistent data, but the database itself cannot become inconsistent.

The general rule for deciding whether to allow a read-only transaction to read a fragment inaccessible to transactions of the general type is to determine whether the transaction is sensitive to reading possibly inconsistent data. For example, a transaction that reads the balances of two bank accounts between which a transfer of funds is taking place is sensitive to this type of inconsistencies. It should read the values either before or after the transfer has taken place, but not while the funds are "in transit." Otherwise, it may appear that the transfered funds have dissappeared. On the other

hand, consider a transaction that tries to estimate the average salary of the company's employees. Suppose that an employee is transfered from department A to department B. Further, suppose that the above transaction runs after the old record (in the fragment listing the employees of A and their related data, including salaries) is deleted but before the new record (for department B) is inserted. This inconsistency of view will result in omitting one employee from the statistical sample. But assuming that the number of employees at the company is large, the outcome will not be affected very significantly.

Garcia-Molina and Wiederhold [Garc82a] showed that, in general, holding read-only transactions to lower standards of correctness provides better concurrency in distributed databases. In the case of Bakunin data networks, the benefit of such a relaxation is wider access privileges for read-only transactions than for transactions of the general type.

6.6. Updates to Multiple Fragments and Violations of the RAG.

One of the underlying assumptions in our scheme was that every transaction can update at most one fragment — the fragment controlled by the agent initiating the transaction. Although we argued (with the help of examples) that in many cases this restriction is not as limiting as it might seem at first, there still are some cases where it will be necessary to execute a transaction that updates more than one fragment. To that end, the following exception-handling provision can be incorporated into the Bakunin data networks, albeit at some cost in availability whenever it is used.

To update a fragment which it is normally not allowed to write into, a transaction must obtain exclusive locks on the copy of this fragment (or the part of it to be modified) at every node that can read the fragment. Then the transaction proceeds to install the update atomically at all those nodes. After that the locks can be released.

Note that this mechanism is very similar to a number of commonly used concurrency control algorithms for distributed systems.

The described provision guarantees that the correctness criteria are satisfied. However, it entails partial loss of autonomy and, therefore, data availability.

Just as it may be necessary to run transactions that update more than one fragment, sometimes it may also be necessary to run transactions that would violate the restrictions imposed by the RAG. Again, the solution here is to use conventional locking to run these special transactions. Failures during the execution of such transactions could cause blocking and decrease availability.

It is important to be able to handle within out framework the special transactions of the type described in this section. However, such transactions should be relatively rare, otherwise the benefits of the Bakunin networks approach will be minimal.

6.7. Localizing Consistency Predicates.

As was established in Chapter 5, fragmentwise serializability implies that single-fragment consistency predicates, i.e., consistency predicates involving data items from the same fragment, are never violated. The only type of consistency predicates that can be violated by fragmentwise serializability are multi-fragment predicates. Consequently, a good database design using the Bakunin network model would keep the number of such predicates as small as possible, and preferably zero.

In this section we are going to examine a technique that allows us to avoid (at least in some cases) the use of multi-fragment consistency predicates. The main idea is to transform a multi-fragment predicate into a single-fragment one by means of introducing auxiliary fragments and modifying the semantics of some update operations, but without restructuring the existing fragments.

Let $P(v(x_1), v(x_2), ..., v(x_m))$ be a multi-fragment consistency predicate, i.e., $x_1 \in F_1$, $x_2 \in F_2$, ..., $x_m \in F_m$, and for some i and j, $1 \leq i, j \leq m$, $F_i \neq F_j$. We redesign the database in the following manner. Let us create a new fragment, F, with an arbitrary non-empty realm of replication, and assign one of the existing nodes to control it. F will consist of duplicates of data objects $x_1, x_2, ..., x_m$. Let us rename the original objects $x'_1, x'_2, ..., x'_m$.

An update to object x_i (authorized at $N(F_i)$) under the old database design will be translated into the following procedure under the new design. The update will first be performed on x'_i (at node $N(F_i)$). When the update is propagated and installed in the copy of the fragment at node $N(F)$, a monitoring mechanism (another necessary component of the new design) will detect the change and alert agent $A(F)$. $A(F)$ will try to reproduce the update on x_i (in F). The attempt will succeed if this does not cause a violation of the consistency predicate P. It will be aborted, otherwise. Note that the predicate has become single-fragment — it involves objects $x_1, x_2, ..., x_m$, all of them in fragment F, rather than $x'_1, x'_2, ..., x'_m$.

Objects $x'_1, x'_2, ..., x'_m$ are called *request objects*. An update performed on one of them is understood as an expression of intent to update the corresponding "real" object, rather than an actual update. This change in the semantics of update permits us not to enforce a consistency predicate on request objects — after all intentions do not have to be consistent, only the real updates do.

To illustrate the notion of localizing consistency predicates, we turn to the already familiar airline reservation example. Suppose that fragments $F_1, F_2, ..., F_m$ above are reservations fragments controlled by different travel agencies. Let x_i ($x_i \in F_i$, as before) be the total number of seats reserved on a certain flight by the i-th travel agency. Our predicate P will take the following form:

$$P(x_1, x_2, ..., x_m): \sum_{i=1}^{m} x_i \leq MAX$$

where *MAX* is the maximum number of seats allowed by the airline to be booked on this flight (*MAX* might be the seat capacity of the plane plus a certain margin to account for an expected number of cancelations by customers). P is a multi-fragment predicate. Suppose that the correctness criterion chosen for our system is fragmentwise serializability. Therefore there is no guarantee that P will not be vioalted. By introducing a new fragment F consisting of $x_1, x_2, ..., x_m$, and replacing the old $x_1, x_2, ..., x_m$ by request objects $x_1', x_2', ..., x_m'$, we make P single-fragment. Now P is guaranteed to be safeguarded by fragmentwise serializability. Incrementing x_i will imply that a new reservation request has been recorded. This is not an assured reservation, however. Only $N(F)$ (e.g., this could be the computer node at the airline headquarters) will make the final decision whether to grant the request. Thus there is bound to be a delay between the time a reservation is requested and the time it is confirmed, which, incidentally, is no different from the way airline reservation systems operate today, i.e., even when the customer thinks that he has a reservation, it may turn out later that there was an overbooking and the reservation had to be cancelled. In a Bakunin data network with fragmentwise serializability and localized consistency predicates this type of phenomenon is incorporated into the semantics of the database, and is, therefore, dealt with automatically in a systematic way.

It should be emphasized that updates to request objects and updates to "real" objects are performed by separate transactions (the former at node $N(F_i)$, for some i, the latter at $N(F)$). Thus all transactions are still able to run and commit locally, and the property of node autonomy is not compromised.

CHAPTER 7

CONCLUDING REMARKS

7.1. Summary and Conclusions.

The main contribution of this thesis is a new methodology for designing highly available replicated databases. As has been pointed out, communication failures, especially network partitions, may cause a conflict between the goals of high data availability and correctness in replicated databases. Because of this inherent conflict, there can be no general solution that satisfies both goals without a price. The Bakunin network model is no exception. While it provides a number of viable alternatives for many possible database applications, it is by no means a panacea. It is in this light that our approach should be evaluated.

Bakunin data networks are characterized by the ability to guarantee the same amount of data availability during communication failures, including partitions, as during normal operations. At the same time meaningful correctness criteria are also guaranteed. The novelty of the approach lies precisely in this duality. The price for preserving both availability and correctness (however the latter might be defined) is that Bakunin data networks do not support arbitrary transactions. Only certain types of transactions are permitted, defined by the their write and read sets. In addition to that, special update propagation protocols are employed, which may sometimes inhibit the rate of propagation and result in reading of stale data. Nevertheless, this price may be acceptable for applications which require high availability and some meaningful correctness guarantees.

The notion of node autonomy is central to our approach. A node can always initiate transactions unilaterally and then schedule, execute, and commit them locally

without exchanging messages with other nodes. (The only type of message exchange that takes place in a Bakunin data network is update propagation.) Autonomy helps improve efficiency and robustness of the system to a significant extent. Moreover, just as distributed computer systems of the general kind reflect the decentralized structure of organizations that own and operate them, distributed systems with highly autonomous nodes reflect the loose administrative links among the constituent departments (or individuals) that are in control of the nodes, a situation typical of many such organizations.

The framework gives rise to a number of specific Bakunin data networks by varying the three main characteristics: read-access patterns, update propagation protocols, and correctness criteria. We have identified several types of Bakunin data networks each of which possesses a particular mix of these characteristics. Thus a database designer can make a choice that is best suited to the given application.

We have introduced two new correctness criteria that can serve as substitutes for one-copy serializability in some cases. These are virtual serializability and fragmentwise serializability. While allowing more freedom in either transaction read-access patterns or update propagation protocols (or both), they provide some formal guarantees of correctness that, albeit are not as strict as one-copy serializability, still can be useful in analyzing the behavior of the given system. For the sake of comparison, it can be noted that both virtual serializability and fragmentwise serializability are stricter than level 2 consistency on the scale of Gray et al. [Gray76a] (level 3 being traditional serializability).

Table 7.1 summarizes all the options studied in Chapters 3 through 5. The columns in this table correspond to different types of RAGs, while the rows correspond to different update propagation protocols. The columns are arranged from left to right

in the increasing order of restrictions on RAGs. The exception is the tf-RAGs and RAGs with no single-source cycles. These two categories cannot be ordered with respect to each other, i.e., neither set of graphs is a subset of the other. The rows are arranged from top to bottom in the increasing order of expediency of update propagation protocols.

	Arbitrary RAG	**Acyclic RAG**	**tf-RAG**	**RAG with no single-source cycles**	**Loopless RAG**
Topological protocol	fs	S_1	1cs	1cs	1cs
Extended topological protocol	fs	S_1	1cs	1cs	1cs
tf-protocol	fs	fs	S_2	1cs	1cs
Single-source topological protocol	fs	V_1	vs	vs	vs
Unrestricted protocol	F_1	fs	fs	V_2	S_4

1cs: one-copy serializability
vs: virtual serializability
fs: fragmentwise serializability

Table 7.1.

Entries in Table 7.1 denoted by the subscripted letters S, V, and F correspond to the specific Bakunin data networks studied in Chapters 3 through 5. For example, V_2 is a network characterized by a RAG with no single-source cycles and by the unrestricted update propagation protocol. V_2 guarantees virtual serializability for all possible schedules. Other entries correspond to other possible options and are labeled "1cs"

(for one-copy serializability), "vs" (for virtual serializability), or "fs" (for fragmentwise serializability) to indicate which correctness criterion they enforce. These options are not of any significant interest, however, since each of them is superseded by one of the investigated Bakunin data networks. For example, entries in the Extended Topological Protocol row to the right of entry S_1 trivially guarantee one-copy serializability. That is because they correspond to restricted types of acyclic RAGs, and we know that S_1 guarantees the same criterion for any acyclic RAG.

Flexibility of the Bakunin network model is manifested in how easily it can be extended and generalized (see Chapter 6). In fact, this can be pursued to the point where general types of transactions are supported and the synchronization mechanisms used resemble the traditional ones. Needless to say, it does not make much sense to use our approach as a substitute for any of these traditional schemes. However, the property of extensibility may be quite useful for designing "hybrid" systems, i.e., systems that consist of parts that have different characteristics. For example, we could have a database that guaranteed virtual serializability for transactions accessing a certain subset of fragments and one-copy serializability for transactions accessing another subset. Moreover, movement of agents could be permitted among some nodes but not others. Finally, in the same database, there could be a subset of fragments that allowed arbitrary types of transactions (in terms of their read and write sets) as long as these transactions accessed fragments only in this subset. All this could be achieved using the same model — Bakunin data networks — thereby significantly simplifying the design.

7.2. Directions for Future Work.

One of the most interesting issues that have not been fully explored in this dissertation is the integration of Bakunin data networks with more traditional mechanisms for managing replicated data. Examples of such integration are provisions that allow

transactions (as an exception rather than a rule) to update more than one fragment and violate read-access restrictions imposed on them by the RAG. In Chapter 6 some ideas for doing this were suggested. However, many unanswered questions still remain. For instance, it was suggested that a transaction that wants to write a fragment that is not controlled by its home node obtain locks on all copies of that fragment stored at nodes that can read them. An interesting open question is whether it is possible, using the properties of RAGs and update propagation protocols, to lock only some of these copies and still preserve correctness. More generally, is there a more efficient integrated mechanism (not necessarily based on locking) for handling exceptions? These and similar questions present an interesting possibility for future research.

An obvious extension of the work presented in this dissertation would be an implementation of a simple system (probably on a network of workstations) for experimenting with different types of Bakunin data networks. Building a physical Bakunin data network has not been of high priority in this research until now because the fundamental questions of correctness, applicability, and performance of our model (at least in its basic form, i.e., as presented in Chapters 2 through 5) can be addressed on the purely conceptual level. However, implementation can still be useful because it is likely to provide some additional insight into the problem of high data availability as well as into the specific solution proposed here. This point is particularly relevant to extensions discussed in Chapter 6.

The work on Bakunin data networks also gives rise to some interesting theoretical problems. One of them is finding an optimal topological order on the nodes of an acyclic directed graph. Recall that many of our update propagation protocols were based on topologically ordering the nodes (of a subgraph) of a RAG. While the choice of a specific order does not affect the correctness of the protocol, it can affect its expediency, as

determined by characteristic paths of the propagation graph (see Chapter 2). Thus, it is desirable to have an efficient algorithm for finding a topological order that is optimal in some expediency metric. For example, we might choose to minimize the maximum (or the average) length of a characteristic path.

References

Abba85a.
Abbadi, A. El, D. Skeen, and F. Christian, "An Efficient Fault-Tolerant Protocol for Replicated Data Management," *Proc. 4th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 215-228, Portland, Oregon, March 1985.

Abba86a.
Abbadi, A. El and S. Toueg, "Availability in Partitioned Replicated Databases," *Proc. 5th ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 240-251, Cambridge, MA, March 1986.

Aho74a.
Aho, A.V., J.E. Hopcroft, and J.D. Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, Massachusetts, 1974.

Alsb76a.
Alsberg, P.A. and J.D. Day, "A Principle for Resilient Sharing of Distributed Resources," *Proc. 2nd International Conf. on Software Engineering*, 1976.

Bern79a.
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, no. 3, pp. 203-216, May 1979.

Bern81a.
Bernstein, P.A. and N. Goodman, "Concurrency Control in Distributed Database Systems," *Computing Surveys*, vol. 13, pp. 185-221, June 1981.

Bern86a.
Bernstein, P.A. and N. Goodman, "Serializability Theory for Replicated Databases," *Journal of Computer and System Sciences*, vol. 31, no. 3, pp. 355-374, December 1986.

Bern87a.
Bernstein, P.A., V. Hadzilacos, and N. Goodman, *Concurrency Control and Recovery in Database Systems*, Addison Wesley, 1987.

Blau85a.
Blaustein, B.T. and C. Kaufman, "Updating Replicated Data During Communications Failures," *Proc. 11th Conf. on Very Large Data Bases*, pp. 1-10, 1985.

Chri75a.
Christofides, N., *Graph Theory: An Algorithmic Approach*, Academic Press, 1975.

Davi84a.
Davidson, S.B., "Optimism and Consistency in Partitioned Distributed Database Systems," *ACM Transactions on Database Systems*, vol. 9, no. 3, pp. 456-481, September 1984.

Davi85a.
Davidson, S.B., H. Garcia-Molina, and D. Skeen, "Consistency in Partitioned Networks," *ACM Computing Surveys*, vol. 17, no. 3, pp. 341-370, September 1985.

Eage83a.
Eager, D.L and K.C. Sevcik, "Achieving Robustness in Distributed Database Systems," *ACM Trans. on Database Systems*, vol. 8, no. 3, pp. 354-381, September 1983.

Eswa76a.
Eswaran, K.P., J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM*, vol. 19, no. 11, pp. 624-633, November 1976.

Garc82a.
Garcia-Molina, H. and G. Wiederhold, "Read-Only Transactions in a Distributed Database," *ACM Trans. on Database Systems*, vol. 7, no. 2, pp. 209-234, June 1982.

Garc83a.
Garcia-Molina, H., T. Allen, B. Blaustein, R.M. Chilenskas, and D.R. Ries, "Data-Patch: Integrating Inconsistent Copies of a Database after a Partition," *Proc. 3rd IEEE Symp. on Reliability in Distributed Software and Database Systems*, pp. 38-48, Clearwater Beach, FL, October 1983.

Giff79a.
Gifford, D.K., "Weighted Voting for Replicated Data," *Proc. 7th ACM SIGOPS Symp. on Operating Systems Principles*, pp. 150-159, Pacific Grove, CA, December 1979.

Gray76a.
Gray, J.N., R.A. Lorie, G.R. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Data Base," in *Modelling in Database Management Systems*, ed. G.M. Nijssen, pp. 365-394, North Holland Publishing Company, 1976.

Kort88a.
Korth, H.K. and G. Speegle, "Formal Model of Correctness without Serializability," *Proc. ACM SIGMOD International Conference on Management of Data*, pp. 379-386, Chicago, June 1988.

Lync83a.
Lynch, N., "Multilevel Atomicity -- A New Correctness Criterion for Distributed Databases," *ACM Transactions on Database Systems*, vol. 8, no. 4, pp. 484-502, December 1983.

Mehl84a.
Mehlhorn, K., *Data Structures and Algorithms 2: Graph Algorithms and NP-Completeness,* Springer-Verlag, Berlin, 1984.

Mino82a.
Minoura, T. and G. Wiederhold, "Resilient Extended True-Copy Token Scheme for a Distributed Database System," *IEEE Transactions on Software Engineering*, vol. SE-8, no. 3, pp. 173-189, May 1982.

Papa79a.
Papadimitriou, C.H., "The Serializability of Concurrent Database Updates," *JACM*, vol. 26, no. 4, pp. 631-653, October 1979.

Sari86a.
Sarin, S.K., "Robust Application Design in Highly Available Distributed Databases," *Proc. Fifth Symp. Reliability in Distributed Software and Database Systems*, pp. 87-94, January 1986.

Skee84a.
Skeen, D. and D. Wright, "Increasing Availability in Partitioned Database Systems," *Proc. 3rd ACM SIGACT-SIGMOD Symp. on Principles of Database Systems*, pp. 290-296, Waterloo, Ontario, April 1984.

Thom79a.
Thomas, R.H., "A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases," *ACM Transactions on Database Systems*, vol. 4, no. 2, pp. 180-209, June 1979.

Wrig83a.
Wright, D.D., "Managing Distributed Databases in Partitioned Networks," Ph.D. Thesis, Department of Computer Science, Cornell University, September 1983.