Distributed Systems with Data Replication
A Non-Technical Survey
Technical Report IFI-TR-90.11

CLEMENS H. CAP
Dept. of Computer Science
University of Zurich
Winterthurerstr. 190
CH – 8057 ZURICH
SWITZERLAND

e-mail `cap@ifi.unizh.ch`
voice +1–257 4326
fax +1–257 4004
telex `unii ch 817 251`

16. November 1990

Abstract

Distributed systems allow – a literal interpretation of the word – the distribution of a set of computational objects to a number of sites. A special case of such a distribution is replication, i.e. the placement of physical copies of a single logical object to several different sites. It is necessary to provide a coordination service when using these systems – and in the case of data replication this is of a particular importance. In this survey paper we first will give some examples of the problems that may arise with data handling in a replicating system. Then we will isolate the main requirements for a coordination service. The first and naive approach seems to pose some problems. A more detailed analysis will show the reason, namely the impossibility to define a *generic global time* in a distributed system. Inspired by the rather obvious concept of *serializable transactions* we will introduce *one-copy-serialization* which provides a satisfactory notion of consistency. The most popular and interesting techniques for maintaining consistency are sketched. They include an application of LAMPORT'S famous construction of a non-generic global time. The high overhead causes us to consider more and more sophisticated tricks leading to *virtual views* of site and link failures and the problem of snapshots of a distributed system. We then will follow a different approach not trying to maintain consistency but to detect and repair inconsistencies. However, practical applications may put a severe limit to this optimistic idea.

The presentation is given in plain English to avoid technicalities of a specific school of formal methods. The need of formal techniques for the study of distributed systems is motivated. The paper also mentions closely related distributed phenomena and sketches the beauty and complexity of distributed systems which can be nicely illustrated at the example of replicating systems.

Contents

1 Introduction

Of all features of distributed systems,the possibility to duplicate logical resources such as data or processes at different physical locations is one of the most interesting for the user. Replicating data near the point of its use makes communication both cheaper and faster. Especially duplication of server processes and read-only data, tremendously increases the performance of large systems with many users. In case of site failures the user still has access to the requested service or to a copy of the needed data at a different location, at least he can expect a graceful degradation of the functionality of a distributed system in contrast to its full loss in the case of a centralized one. However in the design of such systems it turns out, that there is a high price for obtaining these attracting features. In this paper we will point out several problems that arise with data replication and we will sketch some of the possible solutions. On studying the established techniques we will find out, that they are closely related to most of the other classical problems of distributed systems. Whereas subjects such as reliable broadcast, event ordering or snapshots may be treated separately, data replication is directly connected with most of these distributed phenomena. Therefore it is also suitable as a demonstration of the aesthetic beauty and complexity of distributed systems.

2 Statement of the Problem

The *data model* we have in mind consists of a set of data items, which we can operate on with read and write requests and arbitrary combinations of them (i.e. transactions with read- and writesets). The *distributed system* we want to consider is composed of several sites, each having one physical copy of *all*[1] our items. Every copy allows read and write accesses made by a controlling process running on the copies site. The individual sites are capable of multiprocessing.

At a moment where no site is operational every copy has a certain well-determined value. If all those values are equal we call the system *statically consistent.* Now we start processing. Human users or client processes contact those sites and issue requests to read and write our logical data items. The replication controlling algorithm of those sites then have to start exchanging messages and finally will translate these requests into accesses to copies throughout the system. Having receivedpossibly different results of a set of read accesses the algorithm then has to decide on the value to be returned to the read request. In case of failures the algorithm may abort requests.

How is this translation going to look like? It *seems* that there is a straightforward solution. Why not translate every read request into a read access of the local copy – and every write request into a write access of the local copy followed by a broadcast of that change to all the other copies? At a closer look one realizes that this approach will have rather unpleasant properties. Imagine that at the same time two users contact two different sites for changing a data item to two different values. In this case the broadcast of those two changes may reach the other sites in two different orderings, due to the finite speed of the communication media and different distances of the individual sites. In this case we might end up with several copies showing the first value and others showing the second. Following the proposed strategy for reading we would receive different answers when attempting to read that item, depending on the site we choose.

In order to understand similar associated problems, let us review some well-known examples.

[1]We will not consider replication schemes where not every site holds a copy of every data item, as they would require a technical apparatus beyond the intention of this paper – though in principle quite similar to those discussed here. Such techniques are described in [2]

3 Examples

3.1 Savings Account without Replication

The data item stored is the current standing of a families savings account (fsa) shared by a husband and his wife. Let us first deal with a single multiprocessing site and one copy only. The present value of the account is say \$800. Nearly at the same time the husband and his wife both want to retract \$100 each, at two different public teller machines connected to this site. This means that the transaction $T = \{x := read(fsa); write(x - 100, fsa);\}$ has to be executed for the husband and the wife process. An algorithm translating their requests into the individual read and write accesses might produce several sequences of executions, e.g.

Execution 1	Execution 2
$x_{husband} := read_{husband}(fsa);$	$x_{husband} := read_{husband}(fsa);$
$x_{wife} := read_{wife}(fsa);$	$write_{husband}(x_{husband} - 100, fsa);$
$write_{wife}(x_{wife} - 100, fsa);$	$x_{wife} := read_{wife}(fsa);$
$write_{husband}(x_{husband} - 100, fsa);$	$write_{wife}(x_{wife} - 100, fsa);$

Execution sequence 1 will end in the *wrong* account value of \$700 whereas number 2, which is by the way the *serial execution* of first the husband's then the wife's transaction, produces the correct result \$600. Solutions to this problem are not very difficult, however, we can expect to find similar problems also in connection with replication, where we have to cope not only with an interleaving of different interaction threads (as in this example), but also with their fully parallel execution at different sites.

3.2 Passwords

The data item stored is a network-wide superuser password. Two administrators agree to change the password from `TOM` to `SUSAN`[2]. Due to lack of coordination both issue a request to change the password approximately at the same time at their own sites. Both administrators first are prompted for the old password, which they enter correctly as `TOM`. Then – just at the same time, but at different sites – they enter the new password `SUSAN`. A certain protocol might realize, that there was a change request issued at about the same time at different sites. However as the new value is the same for both change requests, the protocol might decide, that the value to be propagated as the new correct value should be just that common value `SUSAN`, which is okay for the intended semantics of passwords.

[2]Actually something like `s!24_aCQz` would be a more secure password !

3.3 Savings Account with Replication

Let us have another look at above savings account example. Again the husband and his wife both retract \$100 each at two different teller machines, however now every teller machine is equipped with a CPU managing its own local copy of the logical data item. The protocol of our second example would detect the two change requests and will not hesitate to propagate \$700 as new and *wrong* value to all the remaining local copies – basing this decision on the fact that the new values requested by both teller machines are equal.

These 3 examples illustrate the problems we might run into when trying to control data replication. We now will formulate conditions for a replication controlling algorithm, that avoids such problems.

4 Consistency

4.1 Semantic Consistency

Consistency means for the *user* that the system *behaves reasonably*, e.g. that the sum of all changes to an account is equal to the present value of that account. If the intended semantics of an application is given as a description of its transactions and a set of interdependency rules, implementing this application means to reproduce this semantics in hardware – whether that is a single site or a distributed system with a complicated replication scheme. Of course, this approach is intimately connected with the specific semantics of the application and will not produce a general replication controlling mechanism. This technique – though promising – has not yet led to broadly usable and efficient programs and so we will not discuss it in greater detail (see [8] for further references).

4.2 Syntactic Consistency

In every application a transaction will consist of reading several items, doing some computation and finally writing some items. We could weaken above approach, neglecting the semantics of the transactions, and concentrate solely on the fact, that transactions read and write items. Our controlling algorithms will have to translate the read and write requests of the transactions into suitable sequences of accesses to copies throughout the system. Not depending on the semantics of an application such an algorithm will be applicable to arbitrary transactions.

Every transaction is constructed – at the responsibility of its programmer – to produce the intended behavior in a single-user, single-process, single-copy environment. When moving on to a system supporting multi-user (multi-processing), distribution and replication, the only thing we have to guarantee is, that *the behavior of the application is not changed, independent of its special semantics.*

4.3 Ideas for Syntactic Consistency

It is quite easy to imagine a technique that allows us to control the multi-user (multi-processing) situation of a single-copy system. We have to guarantee, that a transaction – once started – has to be finished, before another transaction may be started. Such an approach however would dramatically slow down processing: During times where one transaction is blocked by e.g. user interaction, there may be *no* other active transaction – how independent it may be from the blocked one. A review of the above condition shows that it is sufficient to require that the outcome of several interleaved transactions is the same as the outcome of *any strict sequentialization* of those transactions. It does not matter, whether this sequentialization is equal to the sequence in which these transactional requests were sent to the system – as the order in which our controlling algorithm receives these requests might anyway be a totally different one, depending on the specific process switching of the operating system. Only after an explicit acknowledgement of the system – and this must be implemented as an additional feature – we may be sure that our transaction has been completed. So our controlling algorithm receives the requests and has to translate them into a sequence of accesses in a manner that the resulting state of our data items could have been reached by *any strict sequential order* of these transactions. Such a *serialization algorithm* is usually constructed using shared and exclusive locks on the data items, tree oriented locking protocols or time stamps.

In the case of example 1, execution 2 produces a state that can be reached by either one of the strict sequential orderings $T_{husband}$; T_{wife} or T_{wife}; $T_{husband}$. In fact execution 2 is even equal to such a strict sequential ordering (which need not be true in the general case). A serialization algorithm using locks might proceed as follows: First a transaction must secure exclusive locks on all data items it ever wants to write and shared locks on all data items it ever wants to read. Hence either one of the two transactions $T_{husband}$ or T_{wife} would be the first to obtain locks and would delay the execution of the other transaction. The locking protocol must guarantee, that not both transactions are granted an exclusive lock at the same time. Furthermore, care must be taken to avoid deadlocks and several details such as lock granularity etc. have to be dealt with ([12],[2]).

The first idea for *replication* would be to require that a controlling algorithm has the following property for every possible sequence of requests: Every successful read request returns the value that was written by the last successful write request that returned before this read request returned – and if there was no such successful write, then the value read is the value of the statically consistent state we started in. Unfortunately this approach suffers several problems. In order to define what it means, that a write request returned successfully *before* a read request returned successfully, we must fix moments in time having the meaning *"a write request returned successfully"* or *"a read request returned successfully"*, and we must explain the meaning of *before*.

A detailed analysis shows, that there is no way for solving the first problem in a natural way: We cannot use the moment where the user receives the message "read gave ..." resp. "write completed". It is possible that an algorithm finishes all bookkeeping, commits, removes all locks and then, immediately before the user is notified of the completed write, there is a long delay. During this delay another site could successfully complete a read request, that would return the just written value – due to the completed bookkeeping such a request will not be blocked or delayed in any way. However this would violate above condition. On the other hand we cannot use the event "all our bookkeeping is done, let's inform the user", as this would require a detailed knowledge of our algorithm – and such details should not be visible in the definition of the functionality resp. the interface of just that algorithm.

4.4 There is no Natural Linear Ordering

The second problem is even worse as it is generally not possible to decide for two events of a distributed system which had happened *before* the other, i.e. there exists no reasonable linear ordering on the events of a distributed system. Let us assume that all our events are represented by atomic moments in time. The execution at each individual site is a sequence of such atomic events, furnishing immediately a (linear) ordering of the events at that site. Among the possible atomic events are the sending and receiving of messages, for which we adopt the *causality principle*: The event of sending a message should happen *before* the event of receiving this message. Transitive closure furnishes the rest of that order relation. Hence an event A *happened before* an event B [3] iff there is a sequence $A = A_{0,0}, A_{0,1}, \ldots, A_{0,n_0}, A_{1,0}, A_{1,1}, \ldots, A_{1,n_1}, A_{2,0}, \ldots, A_{k,n_k} = B$ of events, where the pair $(A_{i,j}, A_{i,j+1})$ denotes two events at site i, with event $A_{i,j+1}$ immediately following event $A_{i,j}$ and $(A_{i,n_i}, A_{i+1,0})$ denotes first the sending of a message at site i to site $i+1$ and then the receiving of that message at its destination. Unfortunately, it is very easy to give examples where two events A and B of two different sites cannot be compared with this (partial, non-linear) ordering. Of course, every possible execution in a distributed systems respects this ordering and in a specific execution one of two uncomparable events might actually *seem* to happen before the other – at least for a certain observer. Above construction, the local sequence of events at a single site and the causality principle, would be the only candidate for a generic linear ordering, but the thereby produced ordering generally is not linear. We will see below, that time stamps can be used to generate several different linear orderings of our events, but the obtained orderings will depend on several arbitrary design decisions in the time stamp algorithm. Furthermore the time stamp ordering may still be different from any artificially introduced [4] global physical clock. In the absence of a reasonable and generic linear ordering we have the possibility to either

[3]... possibly executed on a different machine than A

[4]Modern physics by the way shows that there is no such thing as a global physical time.

choose and settle with one of those arbitrarily generated linear orderings and to obtain numerous different notions of consistency – or to adopt a different approach.

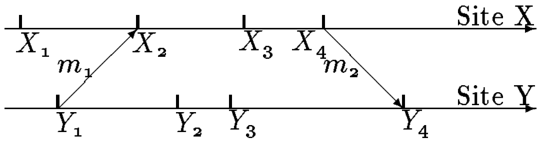

Fig. 1: Ordering of Events

Although X_2 seems to happen before Y_2 and Y_3 before X_3, this is only a consequence of the chosen scale at which the two clocks at sites X and Y are drawn. A small change of the scale or of the speed of the clock of site X near event X_3 could produce a picture suggesting that X_3 is before Y_3. The sites therefore are not able to decide from the executional behavior alone, which of those two events happened earlier. This can only be done in a real-time system with physical clocks and a clock synchronization algorithm and will depend on these additional concepts. However we can clearly decide that X_2 happened before X_3 and Y_1 before X_2. Hence Y_1 also happened before X_3.

4.5 One-Copy Serializability

Let us assume that the replication algorithms at our n copies end up translating the n sequences of transactions $T_1, \ldots, T_n$ issued at the n sites into n sequences $A_1, \ldots, A_n$ of read and write accesses to the copies local to the respective sites. Then it is possible that there is a merge of the n sequences of transactions to a single sequence T with the following special property: Let us assume that the transactions in T act strictly sequential (as in 4.3) on only one copy, then we may immediately identify the value every read of a specific transaction did access. On the other hand our replication algorithm will translate this read into several accesses. They might not return the same value, but the algorithm will have to decide[5] on a final value to be used in the further calculation or to be displayed etc. If there exists such a merge T where for every transaction in T all the values read are the same as all the corresponding values which the replication algorithm declared to be the final values, and if this is the case for all possible T_i, then we will call this replication algorithm to be *one-copy serializing.* Hence the effects of our transactions on the distributed and replicated data base – seen through this algorithm – are the same as the effects of a suitable serialization operating on one copy only.

Note however, that nothing is said about the specific serialization – the algorithm

[5] e.g. using version numbers

is free to produce an effect on the data base that corresponds to a large rescheduling of the transactions. So if a user enters first T_1 then T_2 at his terminal the algorithm is free to first execute T_2 and then, ten days later, T_1. However one might add the requirement, that the algorithm notifies the user when a specific transaction has been completed. Then a user is able to enforce a specific ordering simply by submitting T_2 only after he got the notice that T_1 was carried out. Such a feature can easily be added to most replication algorithms. Furthermore these algorithms are tuned to execute transactions as soon as possible, so it is most likely that T_1 is finished when the user enters T_2 – however this property does not follow from one-copy-serializability.

5 Failure Modes

When discussing fault tolerance properties of a system one usually does not consider arbitrary failures but well-defined subsets of all possible failures. For individual computing sites the *fail-stop* mode is useful, which assumes that the only way a site can fail is by simply not functioning at all – in contrast to partial functionality or wrong behavior. This makes sense as it is rather unlikely that a single site fails in a mode producing some seemingly reasonable functionality that however does not meet its specification in every detail. Furthermore, todays soft- and hardware reduce most failure modes to fail-stop with the help of checksums, plausibility and functionality checks. A similar approach is made for the communication channels, where error correcting codes and other techniques guarantee that the only possible failure mode is the *non-delivery* of a message. Channel and site failures may present themselves as a *network partition*, i.e. where a previously connected network splits into two or more disjoint subsets in a fashion, that two sites can communicate with each other if and only if they belong to the same subset. Whenever a failure gets repaired, recovery algorithms must be started to obtain the new values of the data items that were changed in the mean time and to resynchronize the replication controlling algorithm.

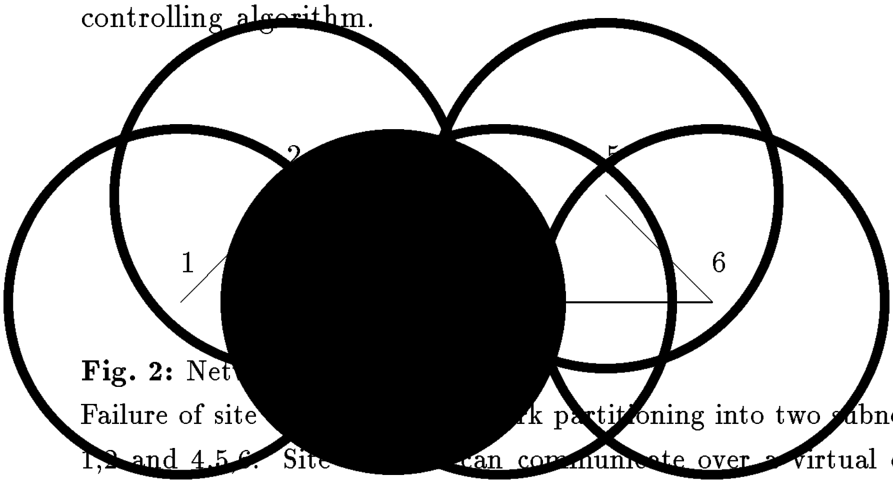

Fig. 2: Net[illegible]
Failure of site [illegible]k partitioning into two subnets consisting of the sites 1,2 and 4,5,6. Site[illegible] can communicate over a virtual channel furnished by the communications software.

It is also possible to consider more complex partition failures in contrast to the mentioned *clean* partitions. For these failure modes we retain the existence of several disjoint subsets with two sites out of two different subsets being unable to communicate with each other, but two sites of the same subset are not necessarily able to directly communicate with each other. We can admit this failure mode to our analysis or hide this situation using suitable communication layers. An even more complicated approach might cope with *partial site-failures*, where a site still is able to run the low level communication layers but the rest of the site, especially the access to the local copy, has failed.

We cannot immediately distinguish a site that *crashed and consequently restarted again* from a site that is *overloaded and thus very slow*. Here we might use time-out techniques in order to decide when a site should be declared unreachable and further processing with this site should be suspended. A site that crashed initiates recovery mechanisms by itself. If a site did not crash but was declared unreachable by time-out, it might not know, that its data is old. Here suitable techniques must initiate some recovery steps before processing can be resumed again. Other approaches tolerate even arbitrarily long delays and thereby take care of crashed and slow sites at the same time.

6 Maintaining Consistency

In the following we will present algorithms that – with the exception of the YP Algorithm – all are one-copy-serializing and thus maintain consistency.

6.1 The Primary Site Algorithm

Whenever a site wants to write a data item, it contacts the primary site (of this item) and transmits its request. The primary site locks its local copy and propagates the updates to all other sites. For a read request first a shared lock for this item must be acquired at the primary site, then any local copy may be read. A transaction on several items will need several locks and if different items have different masters the cooperation of the respective sites is necessary.

This algorithm is resilient to site failures for sites different from the primary site. In case we lose contact with the primary site, there are two possibilities. If, through other properties of our system, we are able to assert that the reason is a failure of the primary site, we may elect another site to become the new master. If we are not able to determine primary site failure, installing a new primary site runs the risk of having two primary sites, separated by a partition – which may lead to inconsistencies. In this case we therefore have to halt the processing in all sites not able to contact the primary site. After a partition the primary site could find itself in a very small subnet, making accesses impossible for the vast majority of sites.

There exist several modifications of the primary site algorithm. A weakening of the protocol leads to the Yellow Page Algorithm. Allowing different sites to play the role of the master, of course only one at a time, leads to the Token Algorithm.

6.2 The Yellow Page Algorithm

The Yellow Pages (YP) constitute a well-known distributed data base for mostly-read and seldom-changed system data in local area networks, such as user passwords, ethernet addresses or service port numbers. As these data are seldom changed the implementors decided not to maintain consistency at all times. The data items are read directly from the local copy without need for a permission from the master. Write accesses first modify the primary site, then these changes are propagated to all the other copies. There is no locking mechanism and inconsistencies are possible during the propagation of a change. Usually the applications that use YP do allow some inconsistencies, provided they do not last too long.

6.3 The Token Algorithm

In this generalization of the Primary Site Algorithm the coordinating master is not chosen once and for ever (resp. until its clearly determined failure), but may be dynamically identified as the unique holder of the `exclusive` token. Other sites, several at the same time, may be holders of `shared` tokens. A write access is granted only to the holder of the `exclusive` token. Therefore, in order to write an item we either must be the primary site, must request to become master in the near future or transmit our request to the present primary site. A read access is granted to sites holding `shared` tokens after having secured a shared lock on the copy of the present master. The read access then may proceed reading the local copy.

Consistency is guaranteed by invalidating all `shared` tokens after an update and by generating new shared tokens for those sites that already have receivedthe newest version. Since a read access requires a shared lock on the copy of the present master, the invalidation of the `shared` tokens can be effectuated e.g. by a lock refusal to holders of out-of-date tokens, so no explicit notification "invalidate you token" at the time of the update is necessary. The lock refusal may be accompanied with a message informing the site that it should wait for a new value to be propagated instead of retrying to obtain that shared lock.

This approach allows a dynamically optimizing choice of the master by passing around the `exclusive` token. It also allows to return from the write request before all the local copies have been updated. It is resilient to our failure modes, but we run into problems if we lose the exclusive token. There are techniques for regenerating lost tokens ([11]), but in case of a partition we still face the same problems as the Primary Site Algorithm. If we are unable to contact the holder of that token this may be due to a partition or to a lost token. Also the token might find itself in a very small subnet.

6.4 The Coteries Algorithm

The methods considered up to now suffered problems connected with a *single* fixed or floating master. This can be remedied by distributing the masters responsibilities to several sites.

The first step in this approach is to construct a *coterie*, i.e. a set of sets of sites, where every two of these have at least one site in common, e.g. $\mathcal{C} = \{\{A, B\}, \{A, C\}, \{A, D\}, \{B, C, D\}\}$. Every transaction on an item requires the consent of all sites of one such set of the coterie. A read request may then proceed securing shared locks on all consenting sites and reading the version numbers of their copies. The copy finally to be read must be up to date, which can be determined by the highest version number. A write request may proceed securing exclusive locks on all those sites and writing the data item and a new version number (the latter being generated as 1 plus the maximum of all the old version numbers of the consenting sites). An arbitrary transaction may obtain different consent sets for the items of its read and write sets.

Consistency is guaranteed by the fact that every two accesses have – by construction of the coteries – at least one site in common. In case of a read access this site furnishes the value of the latest update, for write accesses it delivers the system wide highest version number (which is necessary for determining a correct new version number for just that write). Compared to the previous methods we have an increased overhead for gaining the consent of several sites. However we no longer depend on a single site and therefore fault tolerance is improved. By choosing suitable coteries we can adapt the system e.g. to known failure rates of sites or to communication delays that favour the consent of a specific site. In above example $\mathcal{C}$ stresses the importance of site A, reflecting presumably high reliability and performance. In the case of a partition it is evident, that there will be at most one subnet containing a consent set, and hence consistency will be maintained. However it is more likely that a larger or more important subnet will contain such a consent set. Again we are able to influence this by choosing suitable coteries. Several related algorithms rely on similar principles for guaranteeing consistency.

6.5 The Quorum Algorithm

This algorithm is characterized by two positive numbers r and w called *read- write-quorums.* They must satisfy the following two conditions (where n denotes the total number of copies in the system):

$$\begin{aligned} r + w &> n \\ 2w &> n \end{aligned}$$

For a read access we need the consent of at least r sites whereas for writing we need w consenting sites. Once a consent has been reached, processing continues using locks and version numbers just as above. Arbitrary transactions need a quorum for every affected item.

The first condition guarantees non-empty intersections between a read and a write consent set, whereas the second guarantees non-empty intersections between two write consents. We thus have a similar situation as in the Coteries Algorithm: A read access will always comprise a site having the latest update available, the same is true for a write access and thus we get the highest version number of that item. This algorithm tolerates site and communication failures. By changing the parameters r and w it can be tuned, e.g. choosing a small r speeds up read requests. The choice of r and w also influences the behavior in case of a partition: The second condition guarantees, that write requests take place in at most one subnet, namely that with the largest number of sites. If $r = 1$ we may read any copy but we must write to every copy, a technique which we will call *Write All Algorithm* and which is quite attracting as it allows read requests to immediately access any copy without having to gather consent from several sites. r and w may be different for the individual items of a data base, hence in case of a partition it could happen that one item may be written in one subnet and another item may be written in a different subnet, but transactions having both items in their write set must await repair of that failure.

6.6 Variants of the Quorum and Coteries Algorithms

In the Quorum Algorithm every copy was given a single vote. The *Majority Voting Algorithm* allows the local copies an arbitrary (non negative) number of votes which may also depend on the specific items. Again there are two parameters r and w as above, however in the inequalities n now represents the total number of votes of all copies in the system. The access mechanism stays the same, but a single copy now may contribute to the total quorum with its assigned number of votes. By choosing the parameters r and w and by suitably assigning votes to the copies this algorithm may be tuned in a quite flexible way. We already discussed the influence of the parameters r and w. Giving a single copy a high number of votes stresses its importance. In the special case where all votes are assigned to a single site we get the Primary Site Algorithm. The case where every copy gets a single vote and $r = w = \lfloor \frac{n}{2} \rfloor + 1$ is known as *Majority Consensus Algorithm.* Another variant uses consent sets based on read and write coteries instead of vote based consents. Similar to the Token Algorithm we may design techniques that dynamically rearrange the vote assignments and the parameters r and w. This is particularly interesting for the Write All Algorithm where the failure of even one site would make make writing impossible everywhere. Dynamic parameter rearrangement, depending on the specific failure mode, can remedy these problems ([8]).

More sophisticated replication schemes (where not every site has a copy of every item) allow further variants. In this case it might be possible that even during a partition failure all copies of one item reside in one subnet and all copies of a different item reside in another subnet. If we positively know that a failure is of this type and if we use the Write All Algorithm no rearrangement of parameters has to be made. However such a rearrangement will become necessary if copies of one item are distributed to different subnets. Such an algorithm is described in [7] and a similar approach can be found in [1].

There is an interesting problem associated with both of these approaches: The view a site S has of the state of a distributed system generally is wrong – during the time a state change or an event at a distant site came to the notice of S further state changes have happened, due to the finite speed of message transfer. Even worse, the times it takes for two neighboring sites to inform S of nearly simultaneous events a and b might differ very much, so that S might believe e.g. that a happened long before b. The difficulties are again connected with the problems we had, trying to define a global time. Above mentioned view would be a *snapshot*, i.e. a consistent global state of all sites of our distributed system. Here consistency is again connected with the causality principle – the state of site A may not report a message receivedfrom site B that B in his state does not report as sent. See [4] for a full discussion of this problem.

6.7 The Time Stamp Algorithm

This approach makes use of the fact that we can generate linear orderings on the events of a distributed system ([10]). We already discussed why such orderings cannot be used for defining syntactic consistency – but they can still be employed for constructing a special controlling algorithm.

At every individual site we may imagine a (logical) clock providing every event at that site with a time stamp, i.e. a natural number out of a strictly increasing sequence. This gives us a (trivial) linear ordering of all events at that site. However we cannot use this time stamp immediately for ordering events, as we might end up with a situation like the following: Message A was sent from CPU3 to CPU2 at a (CPU3)–time stamp of 28 and message A was receivedat CPU2 at a (CPU2)–time stamp of 21. Therefore these time stamps would suggest that message A was receivedbefore it had been sent. To remedy this problem every message that is sent is accompanied by the (sender-site-generated) time stamp of the message-sending event. When receiving that message the receiver CPU advances its own clock to a value larger than the stamp that accompanied the message (and, of course, larger than its own present value). This advancing does not ruin the linear ordering at the receiving site, but it guarantees causality of message transmission. When comparing two events on the basis of this modified time stamp mechanism we might run into a tie of two events having the same time stamp. Of course this can only happen for events of two different sites and we will arbitrarily resolve that tie, e.g. by declaring the event of the site with the smaller CPU number as the earlier event. This tie resolution and the different, possibly even varying speeds of the local clocks are responsible for many different linear orderings.

We may use any of these time stamps to develop a replication control algorithm. Let us assume that our communication protocols deliver messages in the order they were sent and let us furnish every site with FIFO-buffers for all the other sites and one for the user requests. All receivedmessages are placed into the corresponding FIFO the oldest message receivedfrom a certain site is the one at the top of the resp. FIFO.

If a site is contacted for a write it places the request including its time stamp into its user FIFO and broadcasts it to all the other sites. If a site is contacted for a read it just places the request and a time stamp into the user FIFO. By inspecting the heads of all its FIFOs every site is able to execute accesses in the order[6] they originated and hence maintains consistency. In case one of the site FIFOs is empty, it is possible that a transactional request is just on its way. In order to guarantee a correct ordering in this case the site sends out an `inquiry` message and waits for a **`response`** message. As messages are delivered in the order they were sent, this will guarantee that we do not miss such a travelling request.

This interesting approach maintains consistency, but produces notable overhead from the inquiry – response messages and blocks when waiting for response messages. It tolerates failures, but we might have to await repair.

[6] Based on our very special time stamps.

7 Regaining Consistency

7.1 Main Idea

Algorithms trying to maintain consistency often behave bad in the presence of partition failures, as they either halt processing in all but one subnet, produce much overhead or even require additional information as in the case of the Primary Site Algorithm. However in many applications the formation of inconsistencies would still be very unlikely even if we neglected the problems associated with partitioning. So we could use any replication controlling algorithm that maintains consistency within one partition – e.g. any of the above algorithms, dropping all effort that is made to maintain consistency across the partitions (in the case of the Primary Site Algorithm this would mean, that a new master may be appointed any time we are not able to contact the present master). Later, when partitions join again, we have to check for inconsistencies. If there are none, we are lucky, otherwise it might be necessary to undo quite a number of already committed transactions and to roll back to a consistent state. Then we could either redo the undone transactions in the repaired system or notify the user of the roll back.

Whereas this seems possible in an idealized system we might run into interesting troubles with real applications where it might no always be possible to undo transactions which are connected with some real world transactions. If the problem is the overbooking of an airplane seat, we can simply inform our passenger. However when the transaction consists in the handing out of several billion dollars or if it was a decision to fire an atomic missile an undo might not be possible. Let us illustrate this approach in the following example.

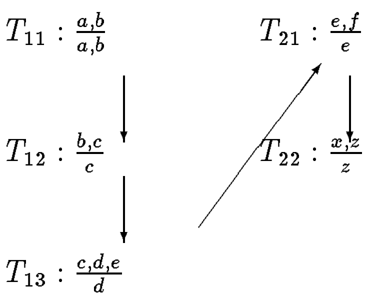

Fig. 3: Serializable across partitions

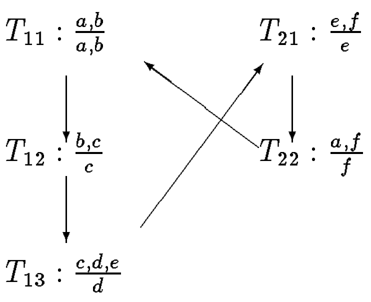

Fig. 4: Not serializable across partitions

Figures 3 and 4 show 5 transactions[7] which were made by an algorithm that guarantees proper coordination of the local copies (i.e. serializability) within the two existing partitions. This algorithm was responsible for a coordination of the transactions in the two partitions to obtain the same effect as the one-copy-serializations T_{11}, T_{12}, T_{13} resp. T_{21}, T_{22}. The transactions in figure 3 are still serializable across the two partitions, but an equivalent serialization must schedule T_{21} after T_{13}, as T_{21} writes the item e whereas transaction T_{13} read the original unmodified e. However in figure 4 also T_{22} must be scheduled before T_{11} for a similar situation with item a. The resulting graph is cyclic, hence no compatible serialization is possible. Similar serialization graphs which are constructed on the accesses to the local copies instead of the transactions in the subnets are used in [2] and as a general framework for analyzing distributed data handling techniques.

7.2 The Optimistic Algorithm

The Optimistic Algorithm closely follows the idea outlined above. First an algorithm is used, that maintains consistency within a fully connected network. If a partition failure occurs, no effort is made to maintain consistency across the partitions – e.g. in the case of the Primary Site Algorithm this means, that a new primary site may be appointed any time we are not able to contact the present master.

Furthermore, in every partition a log of the successfully completed transactions is written[8], either by appointing a master site for this logging or by another distributed protocol. When certain sites can be reached again, the two logs are compared. Every single log consists of a one-copy-serialized sequential graph of transactions. Now we add *interference edges* to this graph by connecting transactions in one partition that read an item to transactions in another partition that wrote this item. If the resulting graph is acyclic then the executed transactions may still be one-copy-serialized even in the union of the two joining partitions and data in one partition may be updated using data in the other partition according to the necessities of the algorithm used in the individual partitions. However if there is a cycle, then several transactions have to be undone until we obtain an acyclic graph. The transactions that were undone then can be rerun automatically or the user can be informed accordingly.

---

[7]Transactions are denoted in *bar notation*, i.e. items read occur above the bar, items written below the bar.

[8]As similar logs must be written anyway for a commit-recovery protocol, this does not constitute much additional overhead.

7.3 The Class Conflict Analysis

Class Conflict Analysis is a consistency maintaining technique using similar ideas as the Optimistic Algorithm. In case of a partition failure every partition is assigned a predetermined set of transactions which it may execute. These sets are determined in a fashion that the resulting serialization graph is acyclic. Upon a repair of the failure several recovery steps might be necessary, depending on the specific algorithm used in the partitions.

8 Formal Techniques

The algorithms we outlined in this paper usually are explained in straight forward plain English and they are often "proved" in a similar way. However any human language is highly inadequate for the description of distributed algorithms. This can be seen immediately, if one tries to actually implement any non-trivial distributed algorithm of which one only has a verbal description. In [9] a total number of 8 deadlock situations occuring in the current version of the ISO File Transfer, Access and Management Protocol FTAM is found using formal techniques. The alarming fact that deadlocks exist in internationally standardized and normed distributed algorithms, which in the FTAM case could even be found with the help of an automated analysis tool, shows the necessity to develop such formalisms and to encourage their use.

Formal specifications and verifications even of simple protocols often tend to be complex enumerations of large state spaces and need many obvious but often tedious technical details. Nevertheless it is necessary to set up formal definitions and proofs in different frameworks such as assertion logic, temporal logic or process algebra.

9 Related Problems and Future Research

There are several problems we did not mention explicitly but that have to be solved in order to be able to get replication running. First, it is necessary to have a protocol for granting shared and exclusive locks on remote machines. For the case of failures we have to develop some ideas. Every access to a physical copy takes some time. A transaction, whose complete and successful termination alone guarantees the correct behavior of our application, may consist even of several such accesses. A site crashing in the middle of a transaction or an incompleted write access would leave its local copy in an inconsistent state. Therefore it is necessary to design algorithms allowing us to treat transactions as *atomic*: They complete immediately, in no time – if they complete at all. If for some reason they cannot be completed, no remainder of any partial execution remains in the copy. More precisely, every transaction ends with a decision to either commit or abort. If it decides to commit, then the copy will show all effects of this access, else it shows none. This act of decision is an atomic one. A single transaction might affect the data items of several sites – either as a consequence of replication or it simply modifies several items, stored at different locations. In this case we even need a distributed commit protocol with the additional property that all sites have to reach the same decision for a specific transaction. Additionally all affected sites must abort a transaction even if a single site crashes during the execution of this transaction. The usual technique to solve these problems is to write a transactional log for each modification carried out. This log then may be used for a reconstruction of a consistent state upon a crash. It might be bad for performance to actually execute every disc access immediately on the disc – often intermediate caching in memory proves to be much more efficient. Here we face a similar problem, a site might crash before the cache manager got a chance to write back its buffers.

Another interesting problem is the choice of the replicating sites. In a large network not all data will be replicated at every single small site. So we first need an algorithm to decide where we want to place which data. It could be necessary to change this replication scheme from time to time to adapt to changed load situations and it would be fine if this adaption could be done *on the fly*, i.e. without the need to halt processing. Then it will be necessary to design a technique for finding the nearest location of a requested data item, maybe not the nearest but the location where access is very fast – or maybe where it is very cheap.

We observed that the various algorithms optimize different aspects and allow some individual tuning via parameters. A changing environment might suggest a parameter change or switching from one algorithm to another. Once again it would be appealing to be able to do this tuning without halting the processing.

Most replication algorithms have been designed in thought and there is not much experience on their actual performance nor on practical issues. Also a theoretical analysis of their performance would be interesting.

Touching many other problems of distributed systems this field promises to be an interesting one.

References

[1] A. E. Abbadi and S. Toueg. Availability in partitioned replicated databases. In *Proceedings 5th ACM Symposium on the Principles of Database Systems*, pages 240 – 251. ACM, March 1986.

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

[3] W. Cellary, E. Gelenbe, and T. Morzy. *Concurrency Control in Distributed Database Systems.* Number 3 in Studies in Computer Science and Artificial Intelligence. North Holland, 1988.

[4] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63 – 75, February 1985.

[5] S. B. Davidson. Optimism andconsistency in partitioned distributed database systems. *ACM Trans. on Database Systems*, 9(3):456 – 481, September 1984.

[6] S. B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *Computing Surveys*, 17(3):341 – 370, September 1985.

[7] D. L. Eager and K. C. Sevcik. Achieving robustness in distributed database systems. *ACM Trans. Database Syst.*, 8(3):354 – 381, September 1983.

[8] M. Herlihy. A quorum-consensus replication method for abstract data types. *ACM Trans. on Comput. Syst.*, 4(1):32 – 53, February 1986.

[9] R. Lai, T. S. Dillon, and K. R. Parker. Verification results for iso ftam basic protocol. In C. A. E Brinksma, G Scollo and Vissers, editors, *Protocol Specification, Testing and Verification*, volume IX of *Proceedings on the Ninth Symposium on Protocol Specification, Testing and Verification, Enschede, 1989*, pages 223 – 234. Noth Holland, 1990.

[10] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Comm. of the ACM*, 21(7):558 – 565, July 1978.

[11] J. Misra. Detecting termination of distributed computations using markers. In *Proceedings 2nd ACM Conference on Principles of Distributed Computing, Montreal*, pages 290 – 294. ACM, August 1983.

[12] M. Singhal. Deadlock detection in distributed systems. *IEEE Computer*, 22(11):37 – 48, November 1989.

ACKNOWLEDGEMENTS
I want to thank my colleagues PETER BAUMANN and EDGAR LEDERER for carefully reviewing the manuscript.