The Maintenance of Common Data in a Distributed System

Baruch Awerbuch *
M.I.T.

Leonard J. Schulman †
M.I.T.

Abstract

A basic task in distributed computation is the maintenance at each processor of the network, of a current and accurate copy of a common database. A primary example is the maintenance, for routing and other purposes, of a record of the current topology of the system.

Such a database must be updated in the wake of locally generated changes to its contents. Due to previous disconnections of parts of the network, a maintenance protocol may need to update processors holding widely varying versions of the database.

We provide a deterministic protocol for this problem, which has only polylogarithmic overhead in its time and communication complexities. Previous deterministic solutions required polynomial overhead in at least one of these measures.

1 Introduction

1.1 Motivation

Many tasks in distributed computing deal with concurrently maintaining the "view" of a common object in the separate sites of a distributed system. This object may be the topology of a communication network (in which case the view is a description of the underlying network graph), or certain resources held at the system sites (in which case the view is an inventory listing the resources held at each site), or a general database. These objects are subject to occasional changes (e.g., a link fails, a resource unit is consumed or released, a database record is modified). It is thus necessary to have an efficient mechanism for maintaining consistent and updated views of the object at the different sites.

The most well known example of a task falling into this category is that of *Topology Update*, whose essence is updating at each node a local view of network topology, which needs to be updated in response to topological changes (failures and recoveries of network links) that occur in the network. Topology update protocols constitute a key component in handling various control and management tasks (e.g. routing, bandwidth reservations) in the major existing networks such as the ARPANET [MRR80], DECNET [Wec80], and SNA [BGJ$^+$85], as well as in new network prototypes [CG88, ACG$^+$90]. Further note that topology-dependent tasks such as Shortest Paths and Minimum Spanning Tree can be solved locally in a network in which current replicas of the topology are maintained at each node. (Of course individual problems may have more efficient problem-specific protocols).

Due to possible partitions of the network, it may be impossible to maintain identical views of the common database at all the nodes, and some "disconnected" nodes or subnetworks will have somewhat outdated views. However, once the connectivity is restored, the differences in views must be eliminated.

The most naive way of achieving this goal is simply to broadcast the information to all the nodes. This method, referred to as *Full Broadcast*, might be very wasteful in communication since it fails to take advantage of prior partial knowledge available in the system. For, processors need be informed of relatively few changes if they already hold nearly correct pictures of the database. The *Incremental Update* strategy in [ACK90, ACG$^+$90] exploits this fact to the extreme, sending each processor only one message per error.

While *Incremental Update* is clearly superior to *Full Broadcast* in terms of communication complexity, it can be significantly inferior in terms of time complexity. This is due to the fact that the *Full Broadcast* method takes advantage of message pipelining, while

*Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. Supported by Air Force Contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171, NSF contract CCR8611442, and a special grant from IBM.

†Dept. of Mathematics and Lab. for Computer Science, M.I.T., Cambridge, MA 02139. Supported by an ONR Graduate Fellowship.

the *Incremental Update* method does not [ACK$^+$91]. The problem of updating the views of all the processors, while taking advantage of prior knowledge and of the possibility of pipelining, is formulated as *Broadcast with Partial Knowledge* in [ACK$^+$91].

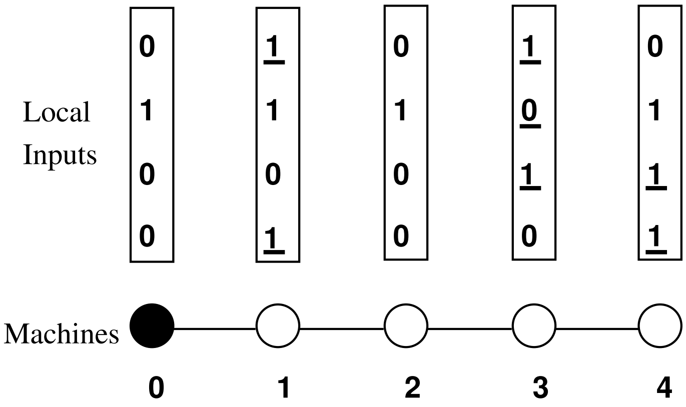

Figure 1: Example illustrating the *Broadcast with Partial Knowledge* problem. In this example, we have $n+1=5$ processors maintaining a joint view with $m=4$ items. The source is processor 0. The incorrect entries of each processor are underlined. The *local discrepancy* is the number of underlined entries at a processor. The *total discrepancy* is $\Delta = 7$.

1.2 The Problem

In the *Broadcast with Partial Knowledge* problem we consider an asynchronous communication network consisting of $n+1$ processors, each holding an m-bit *local input*. One processor is distinguished as the *source*. The problem requires all the processors to write in their local output the value of the input at the source. Note that we allow the inputs of the various processors to differ in arbitrary ways. We make the *neighbor-knowledge* assumption (justified in [ACK90]), namely that each processor knows the inputs of each of its network neighbors.

Throughout our discussion we will consider the case in which the topology of the network is a tree, with the source node being its root. This fits into the framework of [ACK90, ACG$^+$90] in which topology changes are broadcast over a spanning tree. Furthermore it will be sufficient to solve the problem on a chain, with the source at one of its ends. (Reduce by using a depth-first tour). For simplicity (although this could be dispensed with) we will assume that that each processor knows its own position in the chain.

We will picture the global input configuration as a two-dimensional array of bits, with the source at the left, machine index increasing to the right, and each column vector representing a local input of a particular processor (see figure 1). In section 3 this picture will be the setting for a geometric analysis of the time complexity of our protocol.

The input stored at each processor is the local representation of the database at this processor. The correct description of the database is held by the source; the local descriptions may differ from the correct one, and from each other, as a result of changes in the database and in the network topology. Our goal is to spread the source's view of the database throughout the network.

1.3 Complexity

The communication complexity of a protocol is defined as the number of one-bit messages which it transmits.

We use the standard notion of time for asynchronous systems, in which each link can at any time carry a single one-bit message, and convey it from one end of the link to the other in at most one unit of real time.

Another common model in the literature is the "word" model, in a which each message in the above definitions is allowed to contain a logarithmic number of bits (instead of one). We will present the time and communication complexities of our protocol in both models: the differences between the models will of course be only logarithmic, a relatively minor effect since the differences among the complexities of the various protocols discussed in this paper, are polynomial.

In order to quantify the possibility of exploiting local knowledge, we use a measure that captures the level of "correctness" of the knowledge held by each processor [ACK$^+$91]. Let the *local discrepancy* of a given processor be the number of bits in which the local input at this processor differs from the source's input. Define also the *total discrepancy* Δ (which is *not* assumed to be known to the processors) as the sum of the local discrepancies.

We will express the communication and time complexities of our solution as functions of m, n and Δ.

1.4 Existing Solutions

We have already mentioned the *Full Broadcast* solution to the Broadcast with Partial Knowledge problem, which requires $O(nm)$ messages without regard to Δ; but which on the other hand terminates in the optimal $O(n+m)$ time.

In the *Incremental Update* [ACK90] solution, machines are brought one at a time into complete agreement with the source's database. Thanks to the neighbor-knowledge assumption, a machine holding the correct database can correct the errors in its neighbor's database with just one message per error. Thus the communication complexity of this solution, $O(\Delta \log m + n)$, is optimal. However since the correc-

Reference	Commun.	Time
Full Broadcast	mn	$m+n$
Incr. Upd. [ACK90]	$\Delta \log m + n$	$\Delta \log m + n$
This paper	$(\Delta + n) \log m$	$(m+n)\log^3 m$
Lower Bd. [ACK+91]	$\Delta \log \frac{nm}{\Delta} + n$	$m+n$

Figure 2: Complexity of deterministic protocols (bit model).

tions are made sequentially, the time complexity is also $O(\Delta \log m + n)$, which can be as bad as $O(mn \log m)$.

An efficient *randomized* solution was given in [ACK+91]. It achieves essentially optimal complexities in both time and communication, at the expense of a small error probability.

1.5 Our Solution

In this paper we provide a deterministic solution to the Broadcast with Partial Knowledge problem, that uses $O((\Delta+n)\log m)$ communication and $O((m+n)\log^3 m)$ time. In other words, the multiplicative overheads of our solution in communication and time are both polylogarithmic, in comparison with previous deterministic solutions for which at least one of these overheads was polynomial. See figure 2.

The intuition behind our work is as follows. We place one "process", a program that runs on the host processors, in charge of the entire protocol — that is, in charge of "cleaning up" the entire array of bits. Then, in order to strive for good time complexity, we adopt a very weak version of pipelining, by allowing this process to work recursively and create "child" processes. Each of the child processes is in charge of cleaning up a smaller subarray, and they are staggered relative to one another in their work. In order to clean their subarrays the children run the same protocol as the main process (with smaller parameters). In particular, they can create children which are in charge of subarrays of their subarray, and so forth, down to descendents which are in charge of constant-size subarrays.

Now, if each process were to immediately delegate all of its work to its children, the message complexity of the protocol would be very poor. We avoid this problem by allowing our processes to be lazy. For a while they will try, themselves, to clean up the subarray that they are in charge of; only if this starts to take a long time, will they create children, and delegate the work. In this manner the communication costs associated with the creation of children, will not actually be incurred by a process unless enough errors are available to pay for these costs.

The time analysis is more involved. A straightforward induction on the size of a process will not suffice, and instead a constructive insight will be to make a correspondence between the length of a longest chain of causally dependent messages in the protocol (a measure of its time complexity), and the "fractal" dimension of that chain viewed as a subset of the real unit square. (Let m and n tend to infinity, and consider the array of bits as an asymptotic object).

We will find that the dimension of this chain will equal the exponent of the polynomial term in the runtime of our protocol. Further, there is a correspondence between the runtime of the protocol and the limiting process that is used to define the fractal dimension: namely, that the rate of convergence of the limiting sequence to the dimension, will dictate the subpolynomial term in the runtime. (This term will simply be the overhead, since the protocol is optimal up to polynomial factors).

2 Algorithm

2.1 Preliminaries

For simplicity we will assume that m is a power of 2. We will also suppose that n is a multiple of m. (If not then let $a = m(\lceil n/m \rceil - n/m)$, and label the machines $M_a, ..., M_{n+a}$; thus the source is M_a, and the index of the last machine is divisible by m).

Messages in our protocol will only be sent in the direction away from the source (which we have taken to be M_0), to the right in our diagrams. We will refer to the right-hand neighbor of a machine as its successor. The incorrect bits at each machine will be corrected one-by-one in order of increasing bit index, by messages from the machine's left-hand neighbor. No machine will issue such a "correction" message for bit j, until it knows that all its own bits $1, ..., j$ are correct; thus the "clean" area of certified bits will have the monotone appearance of figure 3.

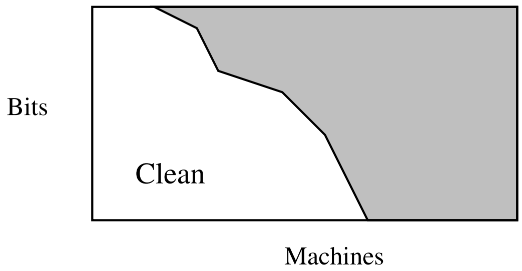

Figure 3: A snapshot of the operation of the algorithm.

Our algorithm will be administered by the *processes*, which "run" on their current host machine, and progress from machine to machine. Each process Q is responsible for cleaning some rectangle in the diagram, which we will refer to as the Q-rectangle: for example, the entire protocol will be started and maintained by the *main process*, which is responsible for cleaning the whole table.

Each process will have two parameters (see figure 4): (a) Its altitude y_Q, and (b) Its height m_Q. The rectangle that it is responsible for consists of bits $y_Q - m_Q + 1$ through y_Q, on the machines from its host's successor up through the next machine whose index is divisible by m_Q. (In our diagrams a vertical line through such a machine will be called an m_Q-column).

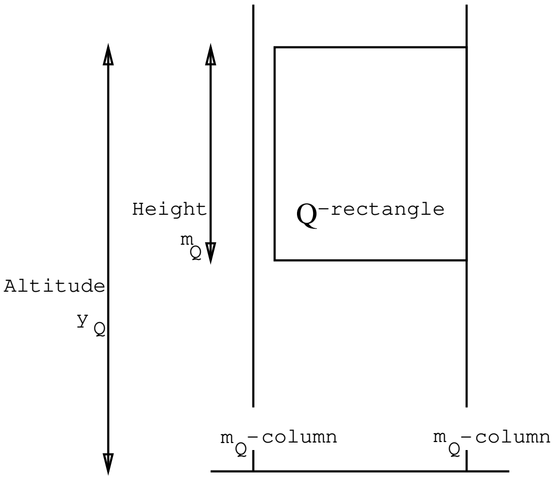

Figure 4: The rectangle of a process Q.

2.2 Outline

A key observation to achieving a balance between the two extreme strategies is this: suppose a process starts correcting the errors in its Q-rectangle itself, thus (as in the incremental update algorithm) taking no risks with the message cost. If it progresses quickly — all is well. Conversely: slow progress is a sign that it is encountering many errors — enough, so that it can "pipeline" part of its task: it can generate a pair of child processes, delegate to each the cleanup of a subrectangle of Q, and pay for their travel costs. In this case, essentially by induction, the process's quick progress will be guaranteed until the children terminate.

By the time it is done, a process will have created either 0, 2 or 4 children (see figure 5).

In addition to its height and altitude, which are fixed from its creation, a process also maintains an "error counter" variable η (in the range $0, ..., m_Q$) with which it counts the errors that it corrects itself.

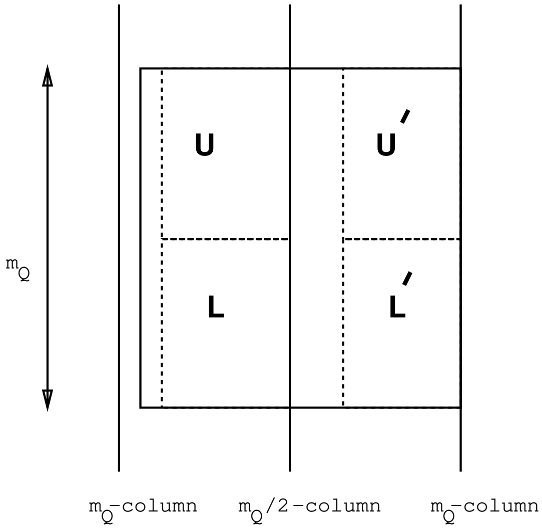

Figure 5: The children of a process.

Every message we use will be labelled with an index between 1 and m, corresponding to its "altitude" in the diagrams. The messages will satisfy a "local monotonicity" condition:

- The messages sent by any machine to its successor over the course of the algorithm, will occur in order of non-decreasing altitude.

We will use two kinds of messages in the algorithm: *corrections*, and *process-carriers*. In a correction message, a machine tells its successor to "flip bit j", for some $1 \leq j \leq m$. There will be exactly Δ such messages during the algorithm. The "altitude" of such a message is j. Note that it uses $O(\lg m)$ bits.

The other class of messages, the process-carriers, are simply the way in which a process moves from its current host machine, to that machine's successor. Such a message uses $O(\lg m)$ bits to encode the altitude, height, and error counter of the process.

At any moment, a process will be in one of two modes: *open mode* or *split mode*. In open mode, the process issues corrections itself; in split mode, it delegates the cleanup to its children and tags along behind them. (See figure 6).

The first thing a process does when it is born is to start an open mode.

The entire protocol begins when the main process, with height m and altitude m, starts an open mode at the source machine M_0.

2.3 Open Mode

When a process Q starts an open mode, it initializes its "error counter" η to 0, and then starts correcting

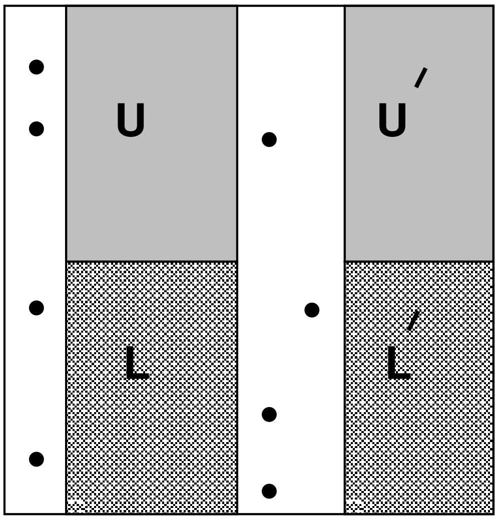

Figure 6: The operation of a process (correction messages marked).

every error it sees in its host's successor, in order of increasing altitude. Once the successor's vector is clean throughout the segment that the process is responsible for, the process moves on to the successor, and continues operating in open mode there. With each error that it corrects, it increments η.

The process (of height m_Q) continues in this manner until either: it reaches an m_Q-column, and terminates; or it reaches an $m_Q/2$-column which is not an m_Q-column, in which case it resets η to 0, effectively beginning a new open mode; or $\eta = m_Q$, in which case it immediately starts a split mode (to continue until the nearest $m_Q/2$-column).

2.4 Split Mode

In split mode, the process itself does not correct any errors. Rather, it creates two child processes, each responsible, in each machine they encounter until the end of the split mode, for cleaning half of the vector-segment that their parent process is responsible for. After creating them, the parent process just tags along behind them.

If the parent is of height m_Q and altitude y_Q, then the height of each child is $m_Q/2$, the altitude of the upper one is y_Q, and the altitude of the lower one is $y_Q - m_Q/2$. Both children terminate at the next $m_Q/2$-column. Thus each is responsible for a rectangle of width at most $m_Q/2$.

In accordance with the local monotonicity condition (section 2.2) on the order of messages, all operations of the upper child will be staggered behind the lower child. All that remains to be specified is that the parent process itself tags along just behind the upper child. (Note that they are at the same altitude, so their order is not dictated by local monotonicity. It is convenient to think of a parent as having "slightly" higher altitude than its upper child.) Observe that the parent proceeds from M_x to M_{x+1} as soon as it sees, by virtue of the fact that both its lower and upper children have proceeded, that the successor's vector is clean up to altitude y_Q.

When the child processes terminate at an $m_Q/2$-column then, if the parent process does not itself terminate (i.e. if this is not an m_Q-column), it simply resets η to 0 and starts a new open mode.

2.5 Base Case

A process with $m = 1$ will simply proceed from machine to machine, correcting the bit that it is in charge of, as it goes. Thus it traverses x links in time x. (Note that $x \leq m = 1$ unless this is the main process.)

3 Analysis

It will be easiest to make both the message analysis and the time analysis in the "word model", where each message consists of $O(\lg m)$ bits, and requires up to a unit of time. Our main theorem summarizes the results of the analysis:

Theorem 3.1 *In the word model our algorithm has message complexity $O(\Delta + n)$ and time complexity $O((m+n)\lg^2 m)$.*

Conversion to the bit model is straightforward:

Corollary 3.2 *In the bit model our algorithm has message complexity $O((\Delta+n)\lg m)$ and time complexity $O((m+n)\lg^3 m)$.*

3.1 Message Analysis

First consider the correction messages: there is exactly one of these per error.

Second, consider the process-carrier messages. The *main* process travels across the entire chain, thus costing n messages.

Each other process, of height $m' \leq m$, uses at most m' process-carriers, and is created as either the upper or lower child of a process of height $2m'$, after that process encountered $2m'$ errors in open mode. We use these errors to pay for both of the children. The cost per error is thus 1.

The total number of messages is therefore $2\Delta + n = O(\Delta + n)$.

In the bit model this comes to $O((\Delta + n)\lg m)$.

3.2 Time Analysis

The standard model of time complexity in an asynchronous setting involves the following assumptions: (a) We ignore the computation time at each node. (b) We assume that each message traverses its link in at most one unit of real time.

The quiescence time T^* of a protocol is defined as the maximum over all combinations of link-traversal times, of the real time required for the protocol to terminate. We will bound the quiescence time of our protocol with another quantity that we shortly define.

The Dependency Graph

First we describe the *dependency graph* $\mathcal{D}$ among messages in a protocol. This is a directed acyclic graph, whose vertices are the *messages* sent from the various machines to their successors; and in which arc (v,w) indicates that the transmission of message w is contingent upon that of message v. ($\mathcal{D}$ may also be thought of as a Hasse diagram on the set of messages).

There are two types of dependencies in $\mathcal{D}$:

1. Machine-internal dependence, implied by the relative priorities among the messages exiting a particular machine.

2. The dependence of a message exiting a particular machine, on messages reaching that machine.

These categories can arise in various distributed protocols. Here is how they occur in ours (see figure 7 for some examples):

1. Upward: v and w are both messages from M_x to M_{x+1}, and v has higher priority than w.

(a) A process in open mode will need to wait for either: a process at a lower altitude; or the last correction it has to issue at the current machine.
(b) A process in split mode will follow its upper child.
(c) A correction issued by process P will need to wait for either: the previous correction issued by P at the same machine; or the last process of lesser altitude than P.

2. Forward: v is a message from M_{x-1} to M_x, and w is from M_x to M_{x+1}.

(a) The very first process-carrier for a new process, can only be created after its parent has arrived at the originating machine.
(b) A carrier for a *continuing* process is dependent on the previous carrier: i.e. the carrier from M_x to M_{x+1} can only leave after the carrier from M_{x-1} to M_x has arrived.
(c) A correction message is dependent on the process that issues it.

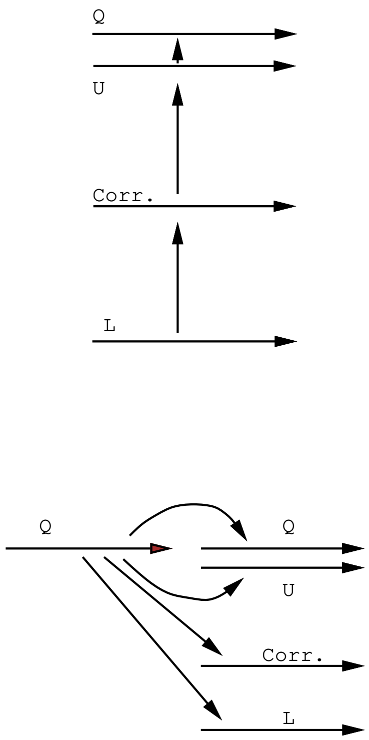

Figure 7: Some dependencies among messages.

Upward arcs always point straight up, while forward arcs point either across, or down and across. Note that every message has at most one dependence of each of these categories, i.e. every vertex of $\mathcal{D}$ has at most one incoming edge of each kind.

Properties of the Dependency Graph

Now: let T be the length of a longest path in $\mathcal{D}$.

Claim 3.3 *The time complexity T^* of our algorithm is less than or equal to T.*

Proof Outline: One must observe that a processor can indeed transmit a message as soon as that message's predecessors in $\mathcal{D}$ have been transmitted. □

This will allow us to discuss our time bound entirely in terms of path lengths in $\mathcal{D}$.

Although it does not, strictly speaking, matter for our result, we point out that in fact there is no loss suffered in the bounding of T^* by T.

Claim 3.4 *In our algorithm, T^* is greater than or equal to T.*

Proof Outline: Our algorithm has the property that its "life history" — the sequence of messages received and transmitted at each machine — does not depend on the link traversal times. □

If Q is a process, denote the process-carrier which takes it from machine M_x to M_{x+1} by $Q(x)$.

Claim 3.5 *A path through $\mathcal{D}$ can cross the time-line of any process at most once, and that is while going up.*

Proof: Observe the following fact regarding $\mathcal{D}$. Let Q be a process which splits at some machine M_x, creating a lower child L and and an upper child U. Then the forward arc from Q (i.e. from the process-carrier $Q(x-1)$) extends to the first message generated by L, which will be either $L(x)$, or the first correction in the L-rectangle. Thus the forward arc into $L(x)$ comes from $Q(x-1)$. Next observe that for $x' > x$, the forward arc into $L(x')$ comes from $L(x'-1)$. These two facts, along with our earlier note that there is only one forward arc into any message, lead to the following conclusion: *No event in the U-rectangle has any bearing on events in the L-rectangle.* Let us say precisely, that a message is in the Q-rectangle if it is generated by Q or by a process that is a descendent of Q. (Thus, its altitude will be between $y_Q - m_Q + 1$ and y_Q, where y_Q and m_Q are the altitude and height of Q). Then, more formally: *No message in the L-rectangle is a descendant of a message in the U-rectangle.* Pictorially this can be viewed as saying that no path crosses the time-line of a process, going down. The claim follows. □

On the other hand note, that events in U are dependent on those in L. In particular, no message in the U-rectangle can depart a machine $M_{x'}$ until after $L(x')$ does so.

In figure 8 we have illustrated a path through a dependency graph, and the time-lines of three generations of processes.

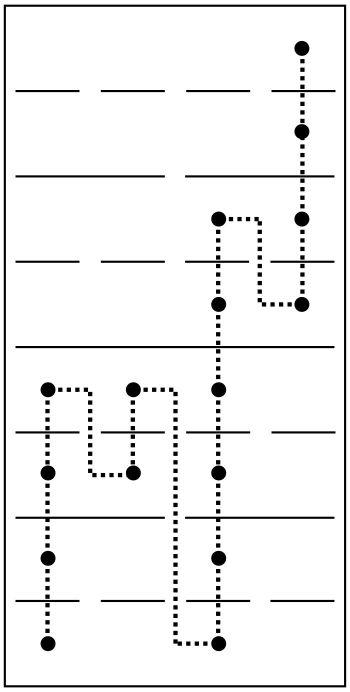

Figure 8: Time-lines of processes, and a path through $\mathcal{D}$.

Proof of the Main Theorem

First we make an observation which will allow us to make our time analysis separately for the stages during which the main process is in open and in split mode.

Observation 3.6 *(See figure 9):*

1. *Suppose Q is a process which starts a split mode at machine M_i. Suppose v is a message in the open subrectangle preceding M_i, and w is a message in the following split subrectangle. Then all paths in $\mathcal{D}$ from v to w, pass through the message $Q(i-1)$.*

2. *Suppose Q is a process which finishes a split mode at machine M_j. Suppose w is a message in the split subrectangle preceding M_j, and z is a message in the following open subrectangle. Then all paths in $\mathcal{D}$ from w to z, pass through the message $Q(j-1)$.*

An examination of the algorithm shows that the main process undergoes $\lceil 2n/m \rceil$ iterations of the open mode – split mode sequence, each iteration ending at an $m/2$-column. (Recall that even if no split is created, the error counter is reinitialized at the $m/2$-column anyway). Thus in view of observation 3.6, we can multiply a bound on the algorithm in the special case $n = m/2$ by the factor $\lceil 2n/m \rceil$ to obtain a bound for the general case. This is how our main result, theorem 3.1, follows from theorem 3.8 (the analysis for $n = m/2$).

Thanks to this simplification we will now work with the dependency graph $\mathcal{D}$ among the messages in an $(m/2) \times m$ rectangle (i.e. involving machines $M_0, ..., M_{m/2}$). For each m', a power of 2 between 1 and m, let us introduce a system of rectangles, which we will call the m'-rectangles, partitioning the array of bits. Each m'-rectangle will be of height m', and (for some k) will include bit indices $km' + 1, ..., (k+1)m'$; it will be of width $m'/2$ and extend between adjacent $m'/2$-columns. Observe that the rectangle of a process of height m' intersects at most two m'-rectangles.

The central idea of the analysis will now be to consider any path γ through $\mathcal{D}$, and to assign the messages

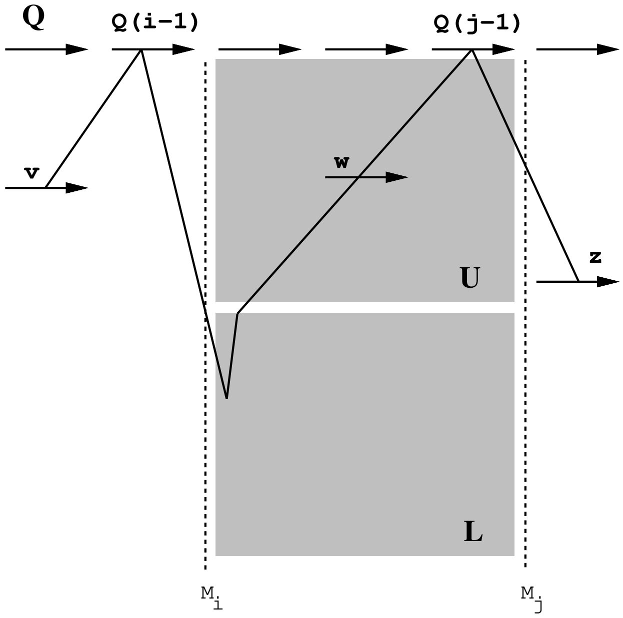

Figure 9: The constraints on paths between consecutive open and split modes.

in γ to m'-rectangles of various sizes, that γ passes through. Then we will bound the number of messages that might be assigned to any m'-rectangle. In this way the total length of γ will be related to the number of m'-rectangles of each size that it encounters. We will bound these numbers with a recurrence relation derived from the properties that γ must satisfy. The combination of these elements will give us a bound on the length of γ.

We proceed with this program. First we describe how to assign messages (vertices of $\mathcal{D}$) to m'-rectangles. Let w be a message issued by a process Q of height m'. We assign w to the m'-rectangle that contains it (which is one of those that the Q-rectangle intersects). This assignment actually has two cases: w is either a a correction message issued by Q in an open mode, or a process-carrier for Q (in either mode).

Now, there is at most one process of height m' intersecting any given m'-rectangle. This process will use at most $m'/2$ process-carriers in the m'-rectangle, and issue at most m' corrections. Hence an m'-rectangle is assigned at most $3m'/2$ messages.

Now we consider just the vertices of γ. We will say precisely that γ encounters an m'-rectangle, if it contains a w that is assigned to that rectangle by the above rule.

Since $\mathcal{D}$ is acyclic, no vertex appears more than once in γ. Therefore, with $m' = m/2^\mu$, we find that the number of edges of γ is at most:

$$\sum_{\mu=0}^{\lg m} a_\mu \frac{3}{2}\frac{m}{2^\mu} \tag{1}$$

Our definition of "encountering" vacuously demonstrates that no edges of γ are charged to m'-rectangles it does not encounter. It also implies that γ cannot encounter an m'-rectangle without also encountering the enveloping $2m'$-rectangle. For, let w be a vertex of γ in the m'-rectangle. Then above w lies the time-line of the process P of height $2m'$, which is the parent of the process that issued w, and which is currently in split mode. If γ exits the P-rectangle upward during this split mode, then it passes through a carrier for P; while observation 3.6.2 implies that this is also the case even if γ exits the split mode on forward edge.

Let a_μ denote the number of $m/2^\mu$-rectangles that γ encounters. In accordance with our program we now study the $\{a_\mu\}$. We will view γ as if it were some limiting object, a subset of the unit square in the Euclidean plane; and then consider both its fractal dimension (which is defined via a limiting process), and the rate at which the finite approximations approach the fractal dimension. Essentially, the fractal dimension will dictate the polynomial exponent of our time analysis — e.g. whether our runtime is linear or quadratic in m; while the rate of convergence toward the dimension will dictate the size of the subpolynomial term multiplying the polynomial term.

In the continuous case, m is infinite, and a measure of the rate of growth of the a_μ, $\lim_{\mu\to\infty} \frac{\lg a_\mu}{\mu}$, is known as the Pontrjagin-Schnirelman dimension of γ [KT59, Man82, PS32].

If we use the last term of (1) (with $\mu = \lg m$) as a very rough representation of the entire summation, then we obtain an estimate of $m^{dim(\gamma)+o(1)}$ on the length of γ. In particular a length of $m^{1+o(1)}$ corresponds to a dimension of 1. Furthermore: establishing that the length of γ is $m^{1+O(\lg\lg m/\lg m)} = m(\lg m)^{O(1)}$ corresponds to bounding the convergence to the dimension by $1 + O(\frac{\lg \mu}{\mu})$.

From now on we will think of γ as a worst-case path, and speak of exact values for the a_μ, instead of bounding them.

Proposition 3.7 $a_\mu = 2^\mu(\frac{\mu}{2}+1)$.

Before proving this result, let us apply it to obtain a bound on the running time of our algorithm:

Theorem 3.8 $T = O(m(\lg m)^2)$.

Proof: $T \le \sum_{\mu=0}^{\lg m} a_\mu \frac{3}{2}\frac{m}{2^\mu} = \frac{3}{2}m\sum_0^{\lg m}(\frac{\mu}{2}+1) = 3m(1+\lg m)(4+\lg m)/8$. □

Proof of proposition 3.7: The main step is to establish the following recurrence relation among the a_μ.

Lemma 3.9 *For every $\mu \geq 1$, $a_\mu = 2a_{\mu-1} + 2^{\mu-1}$.*

Proof: Throughout the proof of this lemma let $m' = m/2^{\mu-1}$.

Recall that γ can encounter a $m'/2$-rectangle only if it encounters the enveloping m'-rectangle.

Now consider any m'-rectangle that γ encounters. We claim that γ can encounter only three of the four $m'/2$-rectangles it contains: for, to encounter the upper-left one means that the process of height m' has split before the $m'/4$-column running down the middle of the m'-rectangle. Now, γ never proceeds to the left, so the lower-right subrectangle can only be encountered after the upper-left one. However as observed earlier, no arcs descend from an upper child's rectangle into the time-line of a lower one. Hence γ cannot encounter both the upper-left and lower-right subsquares.

Next, examine any of the $2^{\mu-1}$ vertical swaths of m'-rectangles, into which the array is partitioned by the $m'/2$-columns. γ enters this swath from the left, and leaves it to the right.

γ encounters some subset of the m'-rectangles of this swath, and we claim that it visits them in order of increasing altitude. That is, if v and w are messages issued by two different processes of height m' in this swath, and v is issued by that of lower altitude, then γ cannot visit w before v. For, let Q be the process which is the first common ancestor of the two processes. Then, since Q itself issues neither v nor w, it must be in split mode at least as early (i.e. as far to the left) as the earliest of v and w. But then for γ to go from w to v, it would have to descend past the time-line of Q's lower child L: this it cannot do. See figure 10.

Finally, we claim that in all but one of the m'-rectangles of the swath, γ can visit only two (rather than three) of the subrectangles. For, since γ never travels leftwards, it has only one edge which crosses the $m'/4$-column running down the middle of this swath. If this edge occurs within some m'-rectangle, then γ can encounter three of its subrectangles; but in lower m'-rectangles γ can encounter only the lefthand subrectangles, and in higher m'-rectangles it can encounter only the righthand subrectangles. See figure 11.

Therefore if γ encountered s m'-rectangles in this swath, it can encounter within it at most $2s+1$ $m'/2$-rectangles. The lemma follows by summing over all $2^{\mu-1}$ swaths. □

Now we rewrite the recurrence we obtained in the lemma, in matrix form:

$$\begin{pmatrix} a_\mu \\ 2^\mu \end{pmatrix} = \begin{pmatrix} 2 & 1 \\ 0 & 2 \end{pmatrix} \begin{pmatrix} a_{\mu-1} \\ 2^{\mu-1} \end{pmatrix}$$

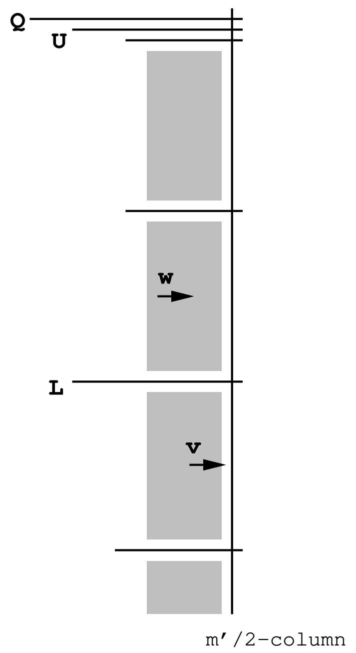

Figure 10: The m'-rectangles in a swath are visited in ascending order.

Both eigenvalues of the transition matrix are 2, therefore a_μ has a closed form expression as 2^μ times a polynomial in μ. The exact solution, taking into account the boundary condition $a_0 = 1$, is given in the statement of the proposition. □

Acknowledgments

We thank Rainer Gawlick for his helpful comments.

References

[ACG+90] Baruch Awerbuch, Israel Cidon, Inder Gopal, Marc Kaplan, and Shay Kutten. Distributed control for paris. In *Proc. 9th ACM Symp. on Principles of Distributed Computing*, 1990.

[ACK90] Baruch Awerbuch, Israel Cidon, and Shay Kutten. Optimal maintenance of replicated information. In *Proc. 31st IEEE Symp. on Foundations of Computer Science*, 1990.

[ACK+91] Baruch Awerbuch, Israel Cidon, Shay Kutten, Yishay Mansour, and David Peleg. Broadcast with partial knowledge. In *Proc. 11th ACM Symp. on Principles of Distributed Computing*, 1991. to appear.

[BGJ+85] A. E. Baratz, J. P. Gray, P. E. Green Jr., J. M. Jaffe, and D.P. Pozefski. SNA networks of small systems. *IEEE Journal on Selected*

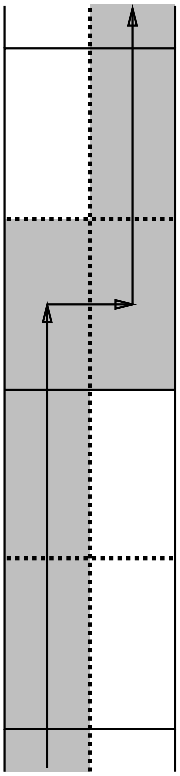

Figure 11: The subrectangles encountered in a swath.

Areas in Communications, SAC-3(3):416–426, May 1985.

[CG88] I. Cidon and I. S. Gopal. Paris: An approach to integrated high-speed private networks. *International Journal of Digital & Analog Cabled Systems*, 1(2):77–86, April-June 1988.

[KT59] A. N. Kolmogorov and V. M. Tihomirov. Epsilon-entropy and epsilon-capacity of sets in functional spaces. *Uspekhi Matematicheskikh Nauk (N. S.)*, 14:3–86, 1959. Translated in American Mathematical Society Translations (Series 2) 17:277-364 (1961).

[Man82] B. Mandelbrot. *The Fractal Geometry of Nature*. W. H. Freeman and Company, San Francisco, 1982.

[MRR80] John McQuillan, Ira Richer, and Eric Rosen. The new routing algorithm for the arpanet. *IEEE Trans. on Commun.*, 28(5):711–719, May 1980.

[PS32] L. Pontrjagin and L. Schnirelman. Sur une propriété métrique de la dimension. *Annals of Mathematics*, 33:156–162, 1932.

[Wec80] S. Wecker. DNA: the digital network architecture. *IEEE Transactions on Communication*, COM-28:510–526, April 1980.