MIT/LCS/TM-326

CONTROLLING WORST-CASE PERFORMANCE OF A COMMUNICATION PROTOCOL AND DYNAMIC RESOURCE MANAGEMENT

BARUCH AWERBUCH

MAY 1987

Controlling worst-case performance of a communication protocol and dynamic resource management

Baruch Awerbuch [†]

Department of Mathematics and
Laboratory for Computer Science,
MIT, Cambridge, MA 02139
(baruch@theory.lcs.mit.edu.arpa)

Abstract

This paper raises a fundamental question, neglected so far in the literature: how to make a distributed algorithm robust against input errors and wrong probabilistic assumptions about the distribution of the inputs or of the link delays. We introduce a notion of *complexity-preserving protocol controller* : this is an automatic procedure that controls worst-case execution of any distributed algorithm. We then suggest a controller with poly-logarithmic overhead. We show that the problem of designing controllers is a special case of another problem, referred to as *dynamic resource management.* We generalize our solution to solve the latter problem.

We believe that the techniques used are basic ones, and will be used to solve a variety of unrelated network problems. Our solution seems to be very practical, since the formal code of the protocol is very simple and thus easy to implement. The technique used in the solution appears to be interesting because a *global* resource is manipulated *locally.* This somewhat resembles the "parallel prefix" technique used extensively in parallel computing.

Key words and phrases

communication protocols, worst-case complexity, admissible inputs, non-admissible inputs, controller , dynamic resource management.

† Supported by Air Force contract TNDGAFOSR-86-0078, ARO contract DAAL03-86-K-0171 and NSF contract CCR8611442.

1. Introduction

1.1. Motivation

Almost any piece of software, either in context of distributed or sequential algorithms, has certain assumptions about its input. That is, if the input belongs to a certain *admissible* domain, then it is guaranteed that the output will be computed correctly and the complexity of the algorithm will be small. Also, we may have certain assumptions about the probability distribution of the inputs. (We view the coins tosses used in the algorithm as a part of its input.)

There are two things that could go wrong with the input to the algorithm. First, the input could be outside of the admissible domain. Second, it may as well be that, for one reason or another, our assumption about the probability distribution of the input simply happens to be wrong, i.e. input distribution is *not* uniform, and coin tosses are neither random nor independent, etc.

Clearly, if something goes wrong, then we do not expect the algorithm to produce the correct output; however we still expect that the algorithm will not do anything destructive, i.e. consume infinite amount of some valuable resources. That is, we want an algorithm whose worst-case performance is almost as good as its expected (or average) performance. Such algorithm always terminates fast. In most cases, it outputs the right value. However, if something unexpected happens, we allow the algorithm to terminate the computation without any output but with a declaration of error. If our probabilistic assumptions are right, the latter would happen very rarely; otherwise we should not have used that algorithm anyway. Such a algorithm is said to be *controlled.*

In case of sequential computing, it is very easy to control the worst-case execution of an algorithm, because time and space complexity of an execution are *known* at any time during the execution. Thus, once an execution runs out of allowable time or space, it is stopped with the declaration of a failure, i.e. the execution is *timed-out.*

This paper raises a fundamental question, neglected so far in the literature: *how to control worst-case performance of a distributed algorithm* ? Unfortunately, while timing out is easy in case of sequential algorithms, it turns out to be a non-trivial task in case of distributed computing. In distributed computing, some of the relevant complexity measures, e.g. number of messages sent, or amount of buffer space used, are hard to evaluate on-line during an execution, since, for example, a node does not know how many messages have been been sent by other nodes.

While sequential algorithms are only sensitive to the input (including coin-tosses), asynchronous network protocols are also sensitive to the behavior of the link delays; some protocols make probabilistic assumptions about the distribution of the link delays. Those link delays are controlled by data link protocols like HDLC [BS-83], which ensure that messages come in the right order and no message gets lost. Those protocols employ lower-level physical-layer protocols, which use coding schemes to detect and correct the noise of the communication channel.

Distributed network protocols are not, by any means, more robust than sequential algorithms since there are much more things that could potentially go wrong. In fact, malfunctioning of one protocol could cause malfunctioning of many other protocols, since network protocols are usually constructed in a modular way, the output of one protocol being the input to another. Thus, once one protocol starts producing wrong output, other protocols which receive their input from the former protocol will get wrong input, and may behave in an unpredictable way. This means that the latter protocol not only will produce incorrect output, but also will consume more network resources than expected, i.e. sending too many messages, occupying too much memory, etc. As a result the network can get congested and may in fact collapse; [P-83] shows how little it took to bring down the whole ARPANET in 1980.

There are various network algorithms, which happen to be very sensitive to their input. For example, the Minimum Spanning Tree (MST) algorithm of [GHS-83] assumes that edge weights are unique; under this assumption the unique MST is found with $O(E+V\log V)$ messages. However, if some edge weights *happen to be identical*, then not only does the algorithm of [GHS-83] fail to build a MST, but its communication complexity is unbounded for such input. Another example is the Bellman-Ford shortest-path algorithm. This algorithm is specified to work on connected graphs which do not contain negative cycles. However, if the graph happens to contain negative cycles, or is not connected, then a straight-forward distributed implementation of this algorithm in the style of the ARPANET routing protocol [JFR-78] will send an unbounded number of messages. Yet another example is the routing protocol, which attempts to forward messages along edges of a rooted tree towards the root. The protocol is very simple: each message is forwarded from a node to its parent in the tree until it reaches the root. The pointers to parents are produced by some other protocol. The assumption is that those pointers indeed form a tree. Observe that if the set of pointers contains a directed cycle, the messages may loop infinite number of times in that cycle, making complexity of the protocol unbounded. Controlling protocols is essential for adapting them to changing input and network topology [A-87-C]. since the transient inputs are not necessarily coherent.

There are also many examples of problem for which link delay distribution is relevant. One of them is the average case performance of Breadth-First-Search (BFS) protocols in a network. In most networks, the delay distribution of the links has the property that its variance is much less than its expected value. Under this assumption, the Bellman-Ford shortest-path algorithm is the simplest and the most practical solution; the expected number of messages sent by it is upper-bounded by $O(E\log E)$. However, if the variance of the delay distribution happens to be large, then the number of messages sent will reach $O(E\cdot V)$; in this case it is advantageous to use, say, more sophisticated algorithms [AG-85], which require $O(E^{1+\epsilon})$ messages in the *worst case*. Another and even more striking example is Spinelli's topology update protocol [S-86],[BG-87]. Assuming link delays have small variance, the average number of update messages sent is E per topological change. However, if link's delays behave "badly", then the number of messages grows as $E\cdot 2^k$, where k is the number of changes. Still, Spinelli's algorithm is believed to be very practical [BG-87].

It appears that controlling protocols is a fundamental problem. All the problems with specific protocols can usually be patched with ad hoc methods, using some special properties of those protocols. However, so far there has been no general methodology that will enable us to treat *different* protocols in a *homogeneous* way. Such a methodology seems to be vital for developing further the theory of communication networking. Thus, it is natural to state the problem of finding a general methodology that will enable to control worst case execution of an *arbitrary* protocol.

1.2. Contributions of this paper

One contribution of this paper is *introducing* the following of complexity-preserving *protocol controller*. Another contribution is an efficient controllers, referred to as *Controller 1*, which transforms arbitrary non-controlled protocol π with complexity c_π with v_π participating nodes into an equivalent controlled protocol ϕ with bit communication complexity $C_\phi = O(c_\pi + v_\pi \log^2 v_\pi \log\log v_\pi)$.

We believe that, besides being a theoretical result, Controller 1 has also a practical value since it is very simple and easy to program; its code can be specified formally on 1 page. Additional attractive features of Controller 1 are: it uses only those edges traversed by the messages of the original protocol, it does not use unbounded counters, it does not assume existence of node identities, it does not assume FIFO regime on communication channels, and its memory requirements are $O(\log\log v_\pi)$ bits per edge.

We demonstrate that complexity-preserving protocol control is a special case of a more general problem of *resource management*. The essence of the latter problem is to devise protocols for each of several communicating asynchronous parallel processes to control access to a shared pool of resources. A weaker version of the resource management problem was stated in [LGFG-86], where a probabilistic solution is given. Under certain probabilistic assumption, [LGFG-86] show that the *average* cost of accessing a resource is only constant, independently of the size of the network. However, without those probabilistic assumptions, the cost of accessing a resource appears to be linear in the size of the network.

We show a simple and efficient solution to the resource management problem, whose worst-case amortized bit complexity per share of the resource is only $O(\log^2 n \log\log n)$, where n is the number of participating processes. The technique used in the solution appears to be interesting because a *global* resource is manipulated *locally*. This somewhat resembles the "parallel prefix" technique used extensively in parallel computing.

The above technique seems to be a fundamental one in distributed computing; we predict that it will be used to solve a wide variety of apparently unrelated problems. For example, it can be used to improve by $O(\log^2 V)$ factor the main result in [AS-87]. Also, it appears that it can be used to solve the "name assignment" problem [P-87], whose essence is to assign distinct names in a network of anonymous processors.

1.3. Organization of this paper

In Section 2, we state the model of the communication network, the complexity measures used to evaluate protocols, and state the problem of controlling the worst-case execution of a protocol. In Section 3, we describe Controller 1A, which is a simplified and slightly less efficient version of Controller 1; In Section 4, we analyze Controller 1A. In Section 5, we present the more efficient Controller 1. In section 6, we present the the problem of dynamic resource management, and outline its solution.

2. Static Network and Protocols

2.1. Model of Static Network

We consider a point-to-point asynchronous communication network, described by an undirected graph *G(V,E)* where the nodes, *V* represent computing units responsible for communication and the links, *E*, represent bidirectional noninterfering communication channels operating neighboring nodes. No common memory or input/output devices (e.g. common clock) are shared by the node's processors. Each node is preprogrammed to perform its part of the computation as well as to receive and send messages to neighbors. These actions are assumed to be performed atomically, i.e. as if performed in zero time. We confine ourselves only to event-driven algorithms, which do not use time-outs, i.e. nodes cannot access a global clock in order to decide what to do. Each message can carry only a *constant* number of bits. The space occupied by the node program cannot exceed a fixed number of bits per adjacent network link, i.e. is proportional to the degree of the node. The nodes *do not* have any identities.

Observe that in such network it is *impossible* for a node to know all the topology of the network. As a result, nodes are forced to communicate with each other in order to solve various network problems. Also, breaking symmetry in such network may require randomization.

In a *static network*, each message sent over a network link arrives at the second end after some finite but undetermined time, which is called *link delay*. The latter may change with time arbitrarily subject to the FIFO rule, i.e. all messages arrive in the same order as they have been sent. This model is fairly well understood and is relatively convenient for the design of efficient algorithms [G-82],[GHS-83],[A-85],[CT-86],[A-87].

2.2. Static-Network protocols

A *protocol* is characterized by a *node algorithm*, which is executed independently by every node. The input and output of each node algorithm is generated at the node itself on a special *input (output)* tape. The contents of node's input tape is fixed and is available at the start time of the node algorithm. The output tape is written only upon termination of the algorithm.

To simplify the basic techniques used in our solution, we assume the protocol has a *single* initiator. The node algorithm at the initiator is started by a special start signal from the outside world. The node algorithm at any other node is started by receiving a message of the protocol.

We denote by δ_{in} (δ_{out}) the alphabet containing all possible inputs (outputs) at a node. Let $\Delta_{in} = \delta_{in}^{|V|}$ and $\Delta_{out} = \delta_{out}^{|V|}$ be the $|V|$-dimensional cartesian products of δ_{in} and δ_{out}. The *global* input $I \in \Delta_{in}$ (output $O \in \Delta_{out}$) is the collection of all node inputs (outputs) prior (after) the execution of the protocol.

Protocol π is associated with a set $\Gamma_\pi \subset \Delta_{in}$ of *admissible* inputs. Any string $I \in \Delta_{in}$, $I \notin \Gamma_\pi$, is said to be a *non-admissible* input [A-87-E]. Also, π is associated with an input/output relation ρ_π, which contains all pairs (I,O), with $I \in \Gamma_\pi$, such that O is one of possible global outputs that could be produced by π on global input I.

2.3. Complexity measures

In communication networks, the amount of communication performed is usually far more significant than the amount of local computation performed by the nodes.

Consider the set of all possible executions of a static-network protocol π under all admissible inputs and all possible link delay distributions. The *admissible-input worst-case communication complexity* of π, denoted by c_π, is the maximum over all such executions, of the total number of *bits* received during that execution. Another relevant characterizations of the protocol π are the upper bounds on the maximal number of edges e_π and maximal number of nodes v_π traversed by messages of the protocol. (Clearly, $e_\pi \leq c_\pi$.)

Considering the set of all executions of a protocol under *arbitrary* inputs (admissible or non-admissible), with arbitrary link delays, we define similarly the *arbitrary-input worst-case communication complexity* of π, denoted by C_π, and the maximal number of edges E_π and maximal number of nodes V_π traversed by messages of the protocol.

Given a probability space over admissible inputs and link delays, we can define admissible-input average-case communication complexity $\overline{c}_\pi$ and the average numbers $\overline{v}_\pi$, $\overline{e}_\pi$ of nodes and edges traversed by messages of the protocol.

A crucial detail in our discussion is that the complexity parameters c_π and v_π are *known* at every node.

2.4. The Problem of Complexity-Preserving Extension

Definition: A protocol π is *controlled w.r.t. admissible worst-case* (*average-case*) if its arbitrary-input worst-case communication complexity is within a constant factor of its admissible-input worst-case (average-case) communication complexity.

To simplify the demonstration of the techniques used, in the rest of the paper we restrict our attention to the problem of making protocols controlled w.r.t. admissible worst-case. Our results can be easily generalized to achieve protocols which are controlled w.r.t. admissible average-case. In the future, unless stated otherwise, "controlled" is a short-hand notation for "controlled w.r.t. admissible worst-case", and "communication complexity" is a short-hand notation for "worst-case communication complexity".

Definition: A protocol ϕ is an *extension* of a protocol π if it performs same as π under admissible inputs of π. Namely, upon receiving any input $I \in \Gamma_{\pi}$, any execution of ϕ produces an output O satisfying input-output relation of π, i.e. $(I,O) \in \rho_{\pi}$.

A *protocol controller* is a protocol transformation, which receives as an input a non-controlled protocol π, and produces as an output a controlled protocol ϕ, which is an extension of π.

3. Controller 1A

3.1. Outline

The basic principle of the controller is that before sending a message of the protocol, the node has to ask permission to do so. Clearly, asking the permission from the initiator for every single message would lead to quadratic complexity. Since we assume that the protocol has a single initiator, it is very easy to maintain a tree spanning all the nodes participating in the protocol. The father of each (non-initiator) node in the tree is the neighbor from which the first message of the protocol was received. The initiator is the root of the tree. Upon receiving the first message of the protocol from its father, a node sets a parameter *Height* to be the *Height* of its father plus 1. Thus, *Height* equals the the height of the node in the tree (distance to the root).

The technique used in our algorithm is similar to some of the techniques used in [G-85], [AP-87], but yet is substantially different from them. It is also worth mentioning that the techniques of [AS-87,AP-86] may be used to *detect* that the protocol sends high number of messages, but would not *prevent* this from happening. Moreover, the detection process in [AS-87] *by itself* would require $O(V \log^4 V)$ messages, which is obviously unacceptable for a "local" protocol π (with $c_{\pi} << V$). However, the following elegant data structure of [AS-87] is useful in our algorithm.

Let *status* of a node is defined as the maximum power of 2 which divides its *Height*. That is, status=1 if *Height* is odd, status=2 if *Height* is even, but is not divisible by 4, status=4 if *Height* is divisible by 4 but not by 8, etc.

The main algorithmic idea here is the following. If a node attempts to send 2^k messages, then this will be authorized by a certain ancestor node of status $2^k \cdot \rho$, located at distance of $\Theta(\rho \cdot 2^k)$ away, where ρ is some constant, which is a power of 2. This technique, which originates from [G-85], is also used in [AP-86].

Each node of status 2^l, $l \geq \rho$, is initially authorized to permit $2^l \cdot \rho^{-1}$ message transmissions. A node can be either *full* or *empty*. Initially, before giving out any permits, each such node is *full*. Once node gives out $2^l \cdot \rho^{-1}$ permits, it becomes *empty*. Whenever such an empty node is asked to give another $2^l \cdot \rho^{-1}$ permits, it tries to borrow $2 \cdot 2^l \cdot \rho^{-1}$ permits from an ancestor node of higher status. If this attempt succeeds, the node authorizes the required $2^l \cdot \rho^{-1}$ message transmissions and becomes *full* again.

The constant ρ is selected as follows. Suppose $2^{\gamma-1} < c_\pi \leq 2^\gamma$ for some integer γ, i.e. $\gamma \approx \log_2 c_\pi$. Then, ρ as a power of 2, satisfying $4 \cdot \gamma < \rho \leq 8 \cdot \gamma$.

3.2. Controller 1A: technical description

The messages that are awaiting to be transmitted by a node are kept in a local *Queue*, until their transmission is authorized. Each time a node processes its Queue, it generates a special *REQUEST* message of *weight* 1 and sends it towards the root. A node which has forwarded such a message "blocks" all incoming *REQUEST* messages, storing them in special buffers *Buffer(e)*, kept for each adjacent edge e. The variable *Owner* may point to one of the sons in the tree, such that *REQUEST* which arrived from that son was forwarded to the father. *Owner=self* indicates that the last such *REQUEST* was generated by the node itself. *Owner=nil* indicates that each *REQUEST* which has ever arrived has already been authorized.

Whenever a node i wants to authorize 2^w message transmissions, it waits until *Owner=nil* holds at i. At that time, it generates a *REQUEST* message of *weight* 2^w, and sends it towards the root. Any *REQUEST* is required to traverse, starting from its originator, a path of length $\rho \cdot 2^w$, (where 2^w is its weight) *at least* until it is handled. The length of the traversed path is represented by an additional "length" counter. A node receiving such message over an edge e stores it in *Buffer(e)* in case that *Owner*$\neq$*nil*; otherwise it sets *Owner*:=e and checks whether the length counter is not less than its status, and that its status is $2^l = \rho \cdot 2^w$. If at least one of those conditions is not satisfied, then the length counter is incremented by 1 and *REQUEST* is forwarded towards the root.

If the root is reached prior to traversing the required length, then the message is handled at the root as follows. If the root has received requests for less than c_π messages, then it sends an *ACK* message back to i; else the *REQUEST* message is simply *ignored*. Another possibility is that after traversing a path of length $2^l = \rho \cdot 2^w$ at least, the *REQUEST* message above reaches thru an edge e a node j of status 2^l, which handles it as follows. If j is *full*, then it becomes *empty* and sends *ACK* back to i. Otherwise, if j is *empty*, then it (tentatively) becomes *full*, sets *Owner* := e, generates new *REQUEST* of weight 2^{w+1} message with initial length counter 0, and sends it towards the root. This new *REQUEST* will eventually be handled in similar way either at the root or at some ancestor node of status 2^{l+1}. Eventually, some successor request either gets handled at a node which is full, or gets blocked. In former case, an *ACK* message is sent back, tracing the whole path of the *REQUEST*, then tracing the path of its predecessor, etc., until it comes to the node which generated the original request.

Every node which propagates an *ACK* sets *Owner*:=*nil* and then attempts to process the blocked *REQUEST* messages, stored in some *Buffer(e)*. If all those buffers are empty, it processes its *Queue*, trying to get authorizations for its own awaiting messages. Eventually, *ACK* arrives at the node with *Owner=self* which originated the *REQUEST*. This node sends the message stored at the top of the *Queue*, deletes it from *Queue*, and then processes its buffers and *Queue* as above.

This process continues until either the algorithm terminates, or all *REQUEST* messages become permanently blocked, as no node can authorize any additional message transmissions. In the latter case, the algorithm does *not* produce a correct output.

3.3. Formal Presentation of Controller 1A

3.3.1. Declarations of procedures, variables and messages

Node Variables

Height=	Height of the node in the tree.
Status=	Status of the node; this is the maximal power of 2 that divides *Height*.
In-edge=	Edge from which the first message of the protocol has arrived. Points to the father of the node in the tree. Initially, *In-edge* = nil.
State=	Receives values *{full,empty}*. Operation *reverse* changes *State* from *empty* to *full* and from *full* to *empty*. Initially, *State* = *full*.
Received=	Total weight of all *REQUEST* messages which have been processes so far at the root. Initially, *Received* = *0*.
Owner=	A node who is waiting for *ACK*. Initially, *Owner*=nil.
Buffer(k)=	Buffer kept for every k. Contains the *REQUEST* messages which wait for acknowledgement. Initially, *Buffer(k)*=$\emptyset$.
Queue=	Queue containing messages which are waiting for permission to be sent. Initially, *Queue*=$\emptyset$.

Messages

REQUEST(x,d)=	message containing x requests that have traveled distance of d.
ACK=	acknowledgement for the *REQUEST* message.
M(h)=	message M of protocol π, which carries Height of the sender in the field h.

Procedures

Process(e)=	procedure which processes the message stored in *Buffer(e) buffer*.
Originate=	procedure which creates *REQUEST* messages.

3.4. Controller 1A: Node Algorithm

```
Upon receiving message $M(h)$ of the protocol $\pi$ on edge e
    if In-edge = nil then
        In-edge := e
        Height := h+1
        Status := maximal power of 2 which divides Height
        State := *full*
        Owner := nil
        Queue := $\emptyset$ ,
        Buffer(e) := $\emptyset$   for all e.

Upon request to send a message $M$ to neighbor $k$
    add $(M,k)$ to Queue /*wait for authorization */
    if Owner = nil then Call Procedure Originate

Procedure Originate /*Called when Owner= nil, Buffer(e)=$\emptyset$ for all e, Queue$\neq\emptyset$ f1*/
    send REQUEST(1,0) on In-edge
    Owner := self

Upon receiving REQUEST(x,d) from k /*x is the weight; d is the traversed distance */
    if Height = 0 then  /*this is the root */
        Received := Received + x
        if Received $\le c_\pi$ then send ACK to k /*authorization */
    else  /*non-root node */
        if Owner $\neq$ nil then put REQUEST(x,d) to Buffer(k)  /*block the request */
        else  /* Owner = nil ; process the request */
            Owner := k
            if x$\cdot\rho$ =Status $\le$ d then  /*must handle this request */
                reverse State  /*from "full" to "empty" or vice versa */
                if State = *empty* then
                    send ACK to Owner  /*it has been authorized */
                    Owner := nil
                else send REQUEST(2x,0) on In-edge  /*new request */
            else send REQUEST(x,d+1) on In-edge  /*dumping it on some ancestor */

Upon receiving ACK  /*authorization to send messages */
    if Owner $\neq$ Self then send ACK to Owner
    else  /*you are the originator of the request */
        send $M$ to $k$ where (M,k) be the contents of the top of Queue
        delete top of Queue
    Owner := nil
    if there is an edge e with Buffer(e) $\neq\emptyset$ ' then Call Process(e)
    else if Queue$\neq\emptyset$ then Call Procedure Originate

Procedure Process(e) /*processes Request stored on edge e, when Owner = nil */
    act as if REQUEST(x,d) has arrived on e, where (x,d) is the contents of Buffer(e)
    Buffer(e) := $\emptyset$
```

4. Analysis

Theorem 1A: The protocol ϕ resulting from application of Controller 1A to a protocol π is an extension of π. For any input, admissible or non-admissible, bit communication complexity of any execution of ϕ is bounded by $C_\phi = O(c_\pi \cdot \log^2 c_\pi \log\log c_\pi)$.

Proof: given below.

Definitions: Consider node i of status $\rho \cdot 2^l$. Let node j be the ancestor of i in the tree of status $\rho \cdot 2^{l+1}$ that handles *REQUEST* messages originated by i. We define *Balance* at node i at certain time t as follows.

1) If the node i is *empty* at t, then $Balance = 2^l$.

2) If the node i is *full* at t, and the last *REQUEST* message originated by i was received and handled (not just stored in the Buffer !) by node j by time t, then $Balance = 0$.

2) If node i is *full*, and the last *REQUEST* originated by i either has not yet arrived at j, or has not yet been extracted by j from a Buffer, then $Balance = 2^{l+1}$.

Comments: The *Balance* of such node i represents the number of message authorizations that this node has been responsible for in the past. Recall that a node i of status $l \cdot \rho$ handles only *REQUEST* messages of weight 2^l. This is captured in part 1) of the definition above. Also, by communicating with higher status nodes, a node can "shift" its responsibility to higher status nodes. This is captured in parts 2),3) of the definition. Namely, at the time node j handles a *REQUEST* originated by i, the *Balance* at i drops by 2^l, and the *Balance* at j raises by the same amount. Observe that a node *does not* know its *Balance*; this parameter is introduced only to simplify the analysis.

Notations: Let M_t denote the total number of authorized protocol messages nodes by time t. Let N_t denote the total number of nodes that entered the protocol by time t. We call a node "active" by a certain time if that node has handled at least one request by that time. Let $N_t(l)$ denote the total number of active nodes of status l by time t, and let $B_t(l)$ denote the total balance of all the nodes of status l by time t. Here $B_t(\gamma)$ is the balance of the root.

Claim A: At any time t, $N_t \leq M_t \leq B_t$.

Proof: Clearly, $N_t \leq M_t$, since a node can enter the protocol only in response to a message, that was sent earlier. On the other hand, $M_t \leq B_t$ since transmission of every message must be authorized. •

Claim B: At any time,

(1) The Balance of a node of status $l < \rho$ is 0.

(2) The Balance of a non-root node of status $\gamma > l \geq \rho$ is at most $2 \cdot l \cdot \rho^{-1}$.

(3) The Balance of the root, $B_t(\gamma)$, never exceeds c_π.

Proof: Immediate. •

Lemma 1: At any time, and for all $0 \leq l < \gamma$, $N_t(l) \leq \frac{N_t}{2^l}$.

Proof: Recall the "distance requirement" imposed in the protocol, according to which a node of status 2^l handles only those requests which traversed a path of length 2^l at least. This requirement guarantees that a node of status 2^l becomes active only after its depth (in terms of distance to a tree leaf) reaches 2^l at least; otherwise none of the requests arriving at that node could have traversed the required 2^l distance. (Similar argument has been used in [AS-87].) It follows that for every active node of status 2^l, there exists a path in the tree of length 2^l, and all those pathes are node-disjoint (for same l). Thus, the invariant preserved at any time thruout the algorithm, is that by time t the "active" nodes of status 2^k constitute at most $\frac{1}{2^k}$ fraction of the total number of nodes that entered the protocol by time t. •

Lemma 2: For any t,

(1) for all $l < \gamma$, $B_t(l) \leq N_t \cdot 2 \cdot \rho^{-1}$.

(2) $B_t \leq c_\pi + \frac{N_t}{2}$.

(3) $M_t \leq 2 \cdot c_\pi$.

Proof:

Statement (1): follows from Lemma 1 and Statement (2) of Claim B.

Statement (2): Summing up the Statement (1) over all $l < \gamma$, we see that the total balance of all non-root nodes is at most $\frac{N_t}{2}$; by Statement (3) of Claim B, the balance of the root is at most c_π.

Statement (3): follows from Statement (2) and Claim A. •

Comment: Statement (3) of Lemma 2 above implies that the number of *protocol messages* is small. However, the controller also sends additional messages, namely *REQUEST* and *ACK* messages. To complete the proof of the Theorem, it remains to show that the number of those messages is small as well.

Lemma 3: The number of *REQUEST* and *ACK* messages sent is at most $O(c_\pi \cdot \log^2 c_\pi)$.

Proof: We ignore *ACK* messages since they can at most double the total cost. We count separately *REQUEST* messages are never acknowledged and messages that are eventually acknowledged. There are at most $N_t \leq 2 \cdot c_\pi$ messages in the former category, as the pathes traversed by requests that will never been acknowledged are all node-disjoint. Now, we concentrate on *REQUEST* messages that are eventually acknowledged.

Clearly, the number of such *REQUEST* messages of weight 2^l is at most $\frac{2 \cdot c_\pi}{\rho \cdot 2^l}$, as each such message is responsible for transmission of $\rho \cdot 2^l$ protocol messages. The transmission cost of each such message is at most $3 \cdot 2^l$, i.e. the total communication cost of *REQUEST* messages of weight 2^l is $O(2^l \cdot \frac{2 \cdot c_\pi \cdot \rho}{2^l}) = O(c_\pi \cdot \rho) = O(c_\pi \log c_\pi)$. Summing up over all l introduces additional factor of γ, bringing total complexity to $O(c_\pi \log^2 c_\pi)$. Similar argument is used in [AP-87]. Since the "weight" of each request is a power of 2, it is easy to modify the algorithm so that each message will carry only $O(\log\log c_\pi)$ bits.

This completes the proof of the Theorem 1A.

5. Adaptor 1

5.1. Outline

We now sketch how to reduce the number of messages from $O(c_\pi \log^2 c_\pi)$ to to $O(c_\pi + v_\pi \log^2 v_\pi)$. First, we notice that we can control the number of nodes in the same way as we control the number of messages transmitted, thus guarantying that the number of "authorized" nodes is $O(v_\pi)$. Next, we "scale" down the messages, i.e. allow every node to send $b_\pi = \frac{c_\pi}{v_\pi}$ without any need for external authorization. However, once b_π non-authorized messages have been sent, the node should ask for permission for the next message. In a sense, messages are organized in "blocks" of size b_π; each new block needs to be authorized.

5.2. Implementation

All the messages that a node has requested in the past to be transmitted are kept in array *Queue[i]* at that node. The variables *Requested* (*Acked*) represent the number of messages that have been requested to transmit (authorized for transmission). First *Acked* entries of the *Queue* contain messages that have been actually transmitted; the remaining *Requested*−*Acked* entries contain messages that await authorization to be transmitted. Each *REQUEST* message originated by the node corresponds to a consecutive block of b_π messages in *Queue*. For each arriving *ACK* message, *Acked* is increased by b_π, and the next block of b_π messages from the *Queue* is transmitted.

In order to control the number of nodes together with controlling the number of message transmissions, we use the following somewhat counter-intuitive trick: we require each node to send b_π dummy messages to itself, at the time it joins the protocol. The number of dummy messages should not exceed $2 \cdot c_\pi$, since each one of at most $2 \cdot v_\pi$ nodes generates b_π dummy messages. Those messages will be accounted together with regular protocol messages; this guarantees that the number of nodes that enter the protocol will not exceed $2 \cdot v_\pi$. Thus, the total number of messages (regular or dummy) sent in a correct execution cannot exceed $3 \cdot c_\pi$. The total number of blocks of messages can reach $3 \cdot v_\pi$.

Thus, we tune the threshold at the root to $4 \cdot v_\pi$. We also update ρ and γ as follows. Suppose $2^{\gamma-1} < v_\pi \leq 2^\gamma$ for some integer γ, i.e. $\gamma \approx \log_2 v_\pi$. Then, ρ as a power of 2, satisfying $4 \cdot \gamma < \rho \leq 8 \cdot \gamma$. The resulting protocol is called *Controller 1*. Its formal presentation is given in the following sub-sections.

Theorem 1: The protocol ϕ resulting from application of Controller 1 to a protocol π is an extension of π. For any input, its bit communication complexity is bounded by $C_\phi = O(c_\pi + v_\pi \log^2 v_\pi \log\log v_\pi)$.

Proof: similar to the Proof of Theorem 1A.

Comment: It is fairly straight-forward to extend this method to the case of multiple initiators, using the methods of [GHS-83] under assumption that some node identities are available; otherwise such identities can be generated by flipping coins. We can use the techniques of [GHS-83] in order to merge together trees rooted at different initiators.

5.3. Formal Presentation of Controller 1

5.3.1. Declarations of procedures, variables and messages

Node Variables

Height=	Height of the node in the tree.
Status=	Status of the node; this is the maximal power of 2 that divides *Height*.
In-edge=	Edge from which the first message of the protocol has arrived. Points to the father of the node in the tree. Initially, *In-edge* = nil.
State=	Receives values *{full,empty}*. Operation *reverse* changes *State* from *empty* to *full* and from *full* to *empty*. Initially, *State* = *full*.
Received=	Total weight of all *REQUEST* messages which have been processes so far at the root. Initially, *Received* = *0*.
Owner=	A node who is waiting for *ACK*. Initially, *Owner*=nil.
Buffer(k)=	Buffer kept for every k. Contains the *REQUEST* messages which wait for acknowledgement. Initially, *Buffer(k)*=$\emptyset$
Queue[i]=	Array, containing all messages which have been requested to send by the original protocol. Initially, *Queue* contains dummy elements (nil,self) for all $1 \le i \le b_\pi$.
Requested=	The number of non-empty elements of the *Queue* array. Initially, *Requested*=b_π.
Acked=	The number of messages that have already been authorized and transmitted. Initially, *Acked*=0.

Messages

REQUEST(x,d)=	message containing x requests that have traveled distance of d.
ACK=	acknowledgement for the *REQUEST* message.
M(h)=	message M of protocol π, which carries Height of the sender in the field h.

Procedures

Process(e)=	procedure which processes the message stored in *Buffer(e)* *buffer*.
Originate=	procedure which creates *REQUEST* messages.

5.4. Controller 1: Node Algorithm

```
Upon receiving message $M(h)$ of the protocol $\pi$ on edge e
    if In-edge = nil then
        In-edge := e
        Height := h+1
        Status := maximal power of 2 which divides Height
        State := full
        Owner := nil
        Queue := $\emptyset$
        put dummy entries into first $b_\pi$ elements of the Queue
        Requested := $b_\pi$
        Acked := 0
        Call Procedure Originate
        Buffer(e) := $\emptyset$ for all e.

Upon request to send a message $M$ to neighbor $k$
    Requested := Requested + 1
    Insert (M,k) to Queue [Requested]
    if Requested - Acked $< b_\pi$ then send the message to k
    else if Owner = nil then Call Procedure Originate

Procedure Originate /*Called when Owner= nil, Buffer(e)=$\emptyset$ for all e, Queue$\neq\emptyset$ f1*/
    send REQUEST(1,0) on In-edge
    Owner := self

Upon receiving REQUEST(x,d) from k /*x is the weight; d is the traversed distance */
    if Height = 0 then /*this is the root */
        Received := Received + x
        if Received $\leq 4 \cdot v_\pi$ then send ACK to k /*authorization */
    else /*non-root node */
        if Owner $\neq$ nil then put REQUEST(x,d) to Buffer(k) /*block the request */
        else /* Owner = nil ; process the request */
            Owner := k
            if x$\cdot\rho$ =Status $\leq$ d then /*must handle this request */
                reverse State /*from "full" to "empty" or vice versa */
                if State = empty then
                    send ACK to Owner /*it has been authorized */
                    Owner := nil
                else send REQUEST(2x,0) on In-edge /*new request */
            else send REQUEST(x,d+1) on In-edge /*dumping it on some ancestor */

Upon receiving ACK /*authorization to send messages */
    if Owner $\neq$ Self then send ACK to Owner
    else /*you are the originator of the request */
        send messages stored in Queue[i], for all Acked $< i \leq$ Acked + $b_\pi$
        Acked := Acked + $b_\pi$
    Owner := nil
    if there is an edge e with Buffer(e) $\neq\emptyset$ then Call Process(e)
    else if Queue$\neq\emptyset$ then Call Procedure Originate

Procedure Process(e) /*processes Request stored on edge e, when Owner = nil */
    act as if REQUEST(x,d) has arrived on e, where (x,d) is the contents of Buffer(e)
    Buffer(e) := $\emptyset$
```

6. Dynamic Resource Management

6.1. The Problem

Informally, the dynamic resource management problem can be stated as follows. Consider managing a big banking system which owes a lot of money to its clients, but has few money in reserves. (Most of the clients money has been invested into third world countries, used to pay gambling debts of the bank directors, etc.). The only way that the system can give back money to its clients is if some other clients will deposit that money first.

In fact, the resources may represent money funds, network bandwidth, available buffer space, line printers, tape drives, etc. In the course of their operation, processes may need to use some resources from the pool. For that purpose, processes will try to *withdraw* required amount of resources from the pool and will *deposit* them back to the pool after using them. If withdrawal request exceeds the current balance, then the algorithm terminates with declaration of *bankruptcy*. Until then, every withdrawal or deposit request is granted.

More formally, there exists a communication network $G=(V,E)$ and a *request* protocol which operates in that network. Each node can enter and leave the request protocol many times. In general, a node is *active* if it currently participates in the protocol, and *passive* otherwise. The set of active nodes can change arbitrarily with time; however at any point of time this set of nodes is a connected subset of the communication graph (V,E).

The request protocol generates at participating nodes requests *withdraw(x)* which means: "withdraw x resources from the pool" or *deposit(x)*, which means: "deposit x resources to the pool". The *weight* of a request *withdraw(x)* is $-x$, and the *weight* of a request *deposit(x)* is x.

Another protocol that is operating in the network is the *granting* protocol which attempts to satisfy the requests above. For every withdrawal or deposit request at a node, the granting protocol may generated signal *granted* at the node which generated the request. At this time, the corresponding request is said to be *granted.* For a given time, the *global balance* at that time is the sum of weights of all granted resources plus some *initial balance.* The *normalized weight* of a request is the ratio of the amount of resources withdrawn or deposit by the request, and the global balance at the time the request is made.

We say that granting protocol satisfies *validity* if the global balance is always *non-negative*, and when the balance reaches 0, a special *bankruptcy* message is delivered to all nodes. We say that granting protocol satisfies *liveness* if either every deposit request is granted or the bankruptcy is eventually declared.

The *Communication Complexity* of a granting protocol is the amortized communication complexity per share of the the resource pool. More formally, let $s=[r_1,r_2, \cdots r_k]$ be a sequence of k requests thruout the life-time of the system. Suppose request r_i has weight w_i, and arrives at the time at which the global balance is b_i. The *flux* f_s of a sequence s is defined as $f_s=\sum_{i=1}^{k}\frac{|w_i|}{b_i}$. That is, flux of s is the total number of "shares-changes" in s. Let M_s be the number of messages sent in order to handle those requests. Then, the amortized complexity m_s of sequence s is $m_s=\frac{M_s}{f_s}$, i.e. total number of messages divided by the flux. The amortized complexity of the granting protocol is defined as a maximum value of m_s, over all finite or infinite sequences s, and over all delay distributions in the network.

Remark 1: It can be easily seen that for any integer f, there exist sequences s with flux $f_s=f$ such the amount of messages that *any* granting protocol sends while handling sequence s is $\Omega(f)$. Thus, it makes sense to define amortized complexity as above.

Remark 2: Clearly, controlling communication complexity of a protocol is a special case of dynamic resource management problem, in which resource withdrawal is transmission of messages over communication channels, resource in this case being the communication bandwidth. Here, resources can be withdrawn, but are never deposit back after being used. Saying that the number of messages sent should never exceed worst-case complexity of the protocol under admissible input is the same as prohibiting simultaneous withdrawal of too many resources. Similarly, controlling total space requirements of a communication protocol falls in the same category; here resource withdrawal is writing into local memory.

6.2. Solution of the dynamic resource management problem

Now, we sketch how to modify *Controller 1* to treat the problem of dynamic resource management. Suppose first that the set of nodes involved in the protocol is fixed, and that the number of participating nodes is N.

All nodes will maintain special variable G, which is the estimate of the global balance; this estimate is accurate within a constant factor. Thus, $q = \frac{G}{4 \cdot N}$ is the estimate of a "quarter" of a global share. Every node has a personal fund, containing q resources. The total amount of resources directly withdrawn (deposit) by the node should never exceed q. In order to deposit or withdraw sums that are integer multiples of q, the node ask for authorizations, as in the Controller 1 algorithm.

The main idea is that we handle withdrawals similarly to deposits, by creating requests of *negative* size. The deposits and withdrawals from the same node cancel each other. When the estimate of the root on the global balance or on the number of participating nodes changes by a constant factor, message is broadcast over the whole tree, updating variables G, q; all other operations are suspended while this message is broadcast. When the root estimate hits 0, a bankruptcy message is broadcast to all participating nodes.

Now, we extend this procedure to handle cases when the set of participating nodes is dynamic. The main idea here is to handle nodes as resources. Namely, every node will have an estimate N on the number of participating nodes, which is accurate up to a constant factor. When nodes join or leave, the corresponding positive or negative updates are created. Once the estimate of the root on the number of participating nodes changes by a constant factor, then appropriate message is broadcast over the whole tree, updating variables N, q; all other operations are suspended while this message is broadcast.

Recall that we assume that the tree over which updates are performed is constructed automatically during the run of the protocol. The father of a node in the tree is a neighbor from which the first message of the protocol has arrived. If the father becomes passive before its sons, then it simply relays update messages forwarded by its sons to the root. If the number of passive nodes in the tree significantly exceeds the total number of active nodes, then this strategy becomes too expensive. At this point it becomes worthwhile to rebuild the whole tree, purging out all passive nodes. With this strategy, passive nodes increase the communication cost of updates only by a constant factor. (This fact was pointed out by S.Plotkin [P-87].)

To rebuild the tree, the root broadcasts a message over the sub-network spanned by the active nodes. Observe that in order to broadcast a message over the sub-network of active nodes, we must know which edges belong to the sub-network.

Towards that goal, whenever a node becomes passive, it notifies all its neighbors in the sub-network about this fact. Thus, when the tree is rebuilt, each node knows which ones of the incident edges lead to active nodes.

The resulting protocol is referred to as *Manager 1.*

Theorem 2: The Manager 1 constitutes a solution to the dynamic resource management problem, with amortized communication complexity of $O(\log^2 n \log\log n)$ bits for each resource share and for each protocol message sent in the request protocol. Here n is the upper bound on the number of active nodes.

Proof Sketch: Arguing as in the analysis of Controller 1A, we see that at all times, the amount of deposits or withdrawals that have not been reported to the root constitutes at most a constant fraction of the global balance. Thus, the root is in control of the flow of the resources regardless of whether the global balance increases or decreases. Consequently, the balance estimate at each node is accurate up to a constant factor. The rest of the proof is similar to the analysis of Controller 1A and thus is omitted. ●

Acknowledgements

It is a pleasure to thank friends and colleagues for their generous contributions to this research. Some of the ideas behind this work originated from the author's discussions with Oded Goldreich in 1985. The author is indebted to Serge Plotkin for his generous help in developing the techniques which lead to the final algorithm. The author also thanks Silvio Micali for his inspiring support and encouragement and Paul Beame for his constructive criticism.

References

[A-85] B. Awerbuch, "Complexity of Network Synchronization", *Journal of the ACM,* Vol. 32, No. 4, October 1985, pp. 804-823.

[A-87-C] B. Awerbuch, "Adapting protocols to dynamic input and network topology" *technical memo* MIT/LCS/TM-327, MIT, April 1987.

[AG-85] B.Awerbuch and R.Gallager, "Distributed Breadth-First-Search Algorithms". *IEEE Symposium on Foundations of Computer Science,* October 1985, Portland, Oregon.

[AS-87] Y.Afek and M.Saks, "Detecting Termination in Face of Uncertainty", to appear in *Proceedings of the ACM Conference on Principles of Distributed Computing* Vancouver, British Columbia, August 1987.

[AP-87] B.Aweruch and S.Plotkin,"$O(E+V\log^2 V)$ messages Leader Election in faulty networks", *manuscript in preparation,* 1987.

[BS-83] A.E.Baratz and A.Segall, "Reliable Link Initialization Procedures", *technical report RC10032,* IBM T.J.Watson Research Center, 1983.

[BG-87] D.P.Bertsekas, R.G. Gallager, "Data Networks", Prentice-Hall, 1987.

[G-85] O.Goldreich, *private communication*, 1985.

[GHS-83] R.G. Gallager, P.A. Humblet and P.M. Spira, "A Distributed Algorithm for Minimum Weight Spanning Trees", *ACM Trans. on Program. Lang. & Systems,* Vol. 5, pp. 66-77, January 1983.

[FLBB-85] M.J.Fischer, N.A.Lynch, J.E.Burns and A.Borodin "Distributed FIFO allocation of identical resources using small shared space", *technical report* MIT/LCS/TM-290, MIT, October 1985.

[JFR-78] J.M. McQuillan, G.Falk and I. Richer, "A Review of the Development and Performance of the ARPANET Routing Algorithm", *IEEE Trans. on Comm.*, Vol. COM-26, No. 12, December 1978.

[LGFG-86] N.A.Lynch, N.D.Griffeth, M.J.Fischer and L.J.Guibas, "Probabilistic Analysis of a Network Resource Allocation Algorithm", *Information and Control,* pp. 47-85, Vol. 68, 1986.

[P-83] R. Perlman, "Fault-Tolerant Broadcast of Routing Information", Computer Networks, vol. 7, Dec. 1983.

[P-87] S. Plotkin, *private communication,* 1987.

[S-86] J.M.Spinelli, "Broadcasting Topology and Routing Information in Computer Networks", M.I.T., Tech. Report LIDS-P-1543, March 1986.