Local Management of a Global Resource in a Communication Network[1]

Yehuda Afek † Baruch Awerbuch‡

Serge Plotkin§ Michael Saks¶

July 28, 1995

Abstract

This paper introduces a new distributed data object called *Resource Controller* which provides an abstraction for managing the consumption of a global resource in a distributed system. Examples of resources that may be managed by such an object include; number of messages sent, number of nodes participating in the protocol, and total CPU time consumed.

The Resource Controller object is accessed through a procedure that can be invoked at any node in the network. Before consuming a unit of resource at some node, the controlled algorithm should invoke the procedure at this node, requesting a *permit* to consume a unit of the resource. The procedure returns either a *permit* or a *rejection*.

The key characteristics of the Resource Controller object are the constraints that it imposes on the global resource consumption. An (M, W)-Controller guarantees that the total number of permits granted is at most M; it also ensures that if a request is rejected then at least $M - W$ permits are eventually granted, even if no more requests are made after the rejected one.

In this paper we describe several message and space-efficient implementations of the Resource Controller object. In particular, we present an (M, W)-Controller whose message complexity is $O(n \log^2 n \log \frac{M}{W+1})$ where n is the total number of nodes. This is in contrast to the $O(nM)$ message complexity of a fully centralized controller which maintains a global counter of the number of granted permits at some distinguished node and relays all the requests to that node.

[1] Preliminary versions of the results presented in this paper appeared in [2, 4, 7].

† Dept. of Computer Science, Tel-Aviv University, Israel 69978. Part of the work was done while the author was with AT&T Bell Labs., 600 Mountain Ave., Murray Hill, NJ 07974.

‡ 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 and NSF contract CCR861144

§ Dept. of Computer Science, Stanford University, Stanford, CA 94305. Research supported by NSF Grant CCR-9304971, ARO Grant DAAH04-95-1-0121, and by Terman Fellowship. Part of the work was done while the author was at M.I.T., Laboratory for Computer Science, and while the author was with AT&T Bell Labs., 600 Mountain Ave., Murray Hill, NJ 07974.

¶ Department of Mathematics, Rutgers University, New Brunswick, NJ 08903. Supported in part by NSF under grants DMS87 03541, CCR-8911388, CCR-9215293 and Air Force office of Scientific Research grant AFOSR-0271.

1 Introduction

Consider a distributed protocol that has an acceptable message complexity under some reasonable assumptions (for example, a constant-time transmission delay assumption), but has a large or unbounded message complexity if these assumptions are not met. In this case it may be desirable to add some mechanism that terminates the protocol if it sends too many messages. A similar mechanism is necessary to prevent a protocol from spreading over too many nodes in the network. Since sending a message or spreading to another node can be viewed as consuming a unit of resource, the above examples indicate a need for a mechanism to manage consumption of a global resource in a distributed network.

In this paper we present an efficient construction of a *Resource Controller* distributed object that can be used to dispense "permits" to the processes participating in a distributed protocol, where each such permit allows the consumption of one unit of the resource. Examples of resources that can be managed using Resource Controller object include number of messages sent in a particular protocol, the total number of participating nodes, the total number of disk blocks in use, etc.

The interface between the controlled protocol and the Resource Controller is through the RESOURCE-REQUEST procedure, a copy of which resides at each node. Before consuming a unit of resource at a node, the controlled protocol should call the RESOURCE-REQUEST procedure at this node and wait until the procedure returns a *permit*. If, instead, it returns *reject*, the controlled protocol should interpret this as "request denied", *i.e.,* the resource is exhausted.

Clearly, in order to be useful, the Resource Controller implementation should not issue more permits than some given amount M. Furthermore, it should issue at least some minimum number of permits before it denies a request for the first time. In fact, since a non-negligible amount of time might pass between requesting a permit and getting it, it is sufficient to require that any execution sequence that includes rejections will include at least $M-W$ issued permits, even if these permits are physically issued only after the first rejection. In other words, it is sufficient to require that a request for a permit is rejected only if there are at least $M-W$ requests that are going to be *eventually* satisfied, even if no more requests are made after the rejected one. The W parameter is essentially the maximum number of "wasted" permits.

The main difficulty in implementing a Resource Controller is the fact that resources are expended independently and asynchronously at different nodes of the network. The simplest solution is to keep a counter at one distinguished node, incrementing this counter for each permit issued. Observe that maintaining such a counter in a large-diameter network can be prohibitively expensive. In particular, it is easy to envision scenarios where such a Resource Controller implementation sends $\Omega(n)$ messages for each issued permit, where n is the number of nodes in the network.

The main contribution of this paper is the construction of a Resource Controller whose message complexity is $O(n\log^2 n\log\frac{M}{W+1})$. In the case where $W = M/2$ (*i.e.,* at most M permits are granted but no less than $M/2$ in case there are rejections), we present a simpler controller whose message complexity is $O(n\log^2 n)$. This should be compared to the $\Omega(nM)$ worst case message complexity of the implementation based on a centralized counter.

Lynch, Griffeth, Fischer, and Guibas [13] have studied a related resource management problem. In their model a fixed amount of resource is initially scattered in some arbitrary way over the nodes of the network. The problem is then to design an efficient algorithm by which the requesting nodes can find available resources. Under certain probabilistic assumptions, they show that the expected number of messages used to find a unit of resource is constant, independent of the network size. Under similar probabilistic assumptions on the request pattern, it is possible to use their techniques to construct a resource controller with nearly optimal message complexity. Without those probabilistic assumptions, *i.e.,* for arbitrary request sequences, this controller might send $\Omega(nM)$ messages, like the controller based on a centralized counter.

Bar-Yehuda and Kutten [8] considered a related problem, namely, electing a leader in a network that recovers from a complete crash. In this problem it is necessary to detect when at least $n/2$ nodes are in one connected component, *i.e.,* count the number of nodes. Their techniques can be used to construct a controller with $O(n^2)$ message complexity.

Our implementation of the Resource Controller object can be extended to solve the following two related problems:

- *Dynamic Name Assignment.* Informally, the problem is to dynamically assign short unique names (ID's) to the nodes participating in some protocol. By "short" we mean that the number of bits used to represent a name is at most the logarithm of the number of participating nodes plus a constant additive term. We assume that nodes join the protocol dynamically, and hence there is no way to preassign the names.
- *Distributed Bank.* The problem is to monitor a pool of money, where nodes may withdraw money from the pool and deposit new money into the pool.

Section 2 presents the basic definitions and the model. The Resource Controller object is defined in Section 3. Section 4 presents an algorithm to implement a special case, the Basic Controller, and Section 5 shows how to use this controller as a building block to implement a full-fledged Resource Controller. Section 6 presents several extensions.

2 Model

We consider the standard point-to-point message passing asynchronous communication network model (see *e.g.* [10, 5]). The network topology is described by an undirected *communication graph*, where the set of nodes represents processors of the network and the set of edges represents bidirectional noninterfering communication channels operating between neighboring nodes. No memory is shared by the node's processors, and there is no notion of a global clock. Messages sent over a link incur an arbitrary but finite delay.

We say that a distributed algorithm *terminates* when all the processes related to this algorithm have terminated. We use the following complexity measures to evaluate the performance of the distributed algorithms in this paper. The *Node Complexity* is the maximum number of nodes participating in the algorithm. The *Communication Complexity* is the worst case total number of elementary $O(\log n)$-bit messages sent during the algorithm execution, where n is the maximum number of participating nodes. The *Bit Complexity* is the worst case total number of bits sent during the algorithm execution.

The *Time Complexity* is usually defined as the worst case time taken by the algorithm under the assumption that the maximum transmission delay is bounded by 1. Since time complexity is always bounded by the message complexity, we concentrate on the latter.

3 Resource Controller – Definition

A Resource Controller is a distributed object that is accessed through a call to the RESOURCE-REQUEST procedure; this procedure can be invoked at any node in the network. When the controlled protocol needs a unit of resource at some node, it has to invoke the RESOURCE-REQUEST procedure at this node and to wait until the procedure returns either a *permit* or a *rejection*.

Every execution of an algorithm that uses the Resource Controller can be associated with an ordered sequence σ of events corresponding to the interactions between the controlled algorithm and the Resource Controller. The set of events consists of a *request* at some node $v \in V$ (corresponds to a call to RESOURCE-REQUEST), *permit* at v, *reject* at v, and a special "resource exhausted" signal at a designated node r.

We use $\sigma(v)$ to denote the restriction of σ to the events that occur at node $v \in V$. Sequence σ is legal if, for all v, $\sigma(v)$ is either empty or (1) it starts with a request event, (2) there is a permit or reject event between any two request events, (3) there are no permit events after the first reject event and, (4) $\sigma(v)$ ends with either a permit event or with a reject event at v. Thus we assume that the controlled algorithm at a node v never issues a new request until the previous request has been given a permit. The controller is required to issue an answer to every request; no message is sent by the controller except in response to a request.

Any sequence σ that corresponds to an execution of a Resource Controller with parameters M and W and a distinguished node r should satisfy the following conditions:

Definition 3.1 [Consumption Conditions]

1. The total number of "permit" events in σ does not exceed M.
2. Sequence σ includes the "resource exhausted" signal event at the distinguished node r if and only if there exists a rejected request in σ.
3. If σ includes the "resource exhausted" signal event, then it includes at least $M - W$ "permit" events.

The last condition implies that at least $M - W$ requests are going to be eventually satisfied, (*i.e.,* if M resources are available, then at most W are "wasted" - not granted at termination).

Observe that the above definition does not preclude the situation where the "resource exhausted" signal is generated before the controller has actually issued $M - W$ permits. Instead, the requirement is that the permits will be issued *eventually*. By using "broadcast-and-echo" one can convert a controller that complies with the above definition into a controller that generates the "resource exhausted" signal only after at least $M - W$ permits were actually issued.

To simplify the description of the algorithms, we assume a "diffusing computation" model, as in the paper by Dijkstra and Scholten [9], *i.e.,* we assume that the controlled algorithm is a single initiator distributed algorithm with new nodes dynamically joining the set of participating nodes. Our algorithms can be converted to work in the case of multiple initiators, but this changes the communication complexity. Specifically, if the nodes that participate in the controlled protocol induce several connected components (which can happen if there are several initiators), we must use additional nodes to pass information from one connected component to another. As a result, the bound on the communication complexity of the controller will depend on the total number of nodes in the network rather than on the number of *participating* nodes.

We also assume that there exists a tree that spans the participating nodes, with the initiator being the root of this tree. Every participating node knows the length of the path from it to the root through the edges of the tree. This assumption can be satisfied by an auxiliary process which dynamically constructs a spanning tree of the participating nodes [9]. The Resource Controller implementations described below use only the edges of this tree, disregarding the rest of the edges in the network.

4 The Basic Controller

This section describes the BASIC CONTROLLER, which operates under the assumption that an upper bound U on the number of participating nodes is known. In the following section we show how to relax this assumption, using the BASIC CONTROLLER as a building block. As stated above, we assume that there exists a spanning tree of the participating nodes rooted at the initiator r, and denote by $D(v)$ the distance in this spanning tree from the root r to v.

A naive approach for constructing a controller is to maintain a counter at the root and to relay each request for a unit of a resource to the root. A simple way to improve on this is to allocate at each node a "bin" that can hold permits, initially holding zero permits, and a special bin at the root, initially holding M permits. When the RESOURCE-REQUEST procedure is called at some node, it tries to satisfy the request from the local bin. If necessary, an empty bin is replenished from the bin at the root. This strategy reduces the communication complexity as compared to the naive controller. Essentially, the reduction is due to the fact that we aggregate a large number (the size of the bin) of requests/permits together, and represent them by a single message. By setting the size of each local bin to $\lfloor W/U \rfloor$, we can ensure that no more than W permits are wasted. This leads to a controller with $O(\frac{M}{W}U^2)$ communication complexity.

4.1 Bin Hierarchy

In order to achieve better communication complexity we maintain at every node v both a "local" bin $b_l(v)$ and a "global" bin $b_g(v)$. The bins are organized in a hierarchical structure, where the size of each bin $b_g(v)$ and its position in the hierarchy depends on $D(v)$, the distance of v from the root. The local bins are at the bottom of the hierarchy. The bin hierarchy described below defines another tree rooted at r which is embedded in the original spanning tree. In order to distinguish the parent relation in this tree from the parent relation in the original tree, we will say that bin b is a *supervisor* of bin b' if b is a parent of b' in the hierarchy tree. Note that since

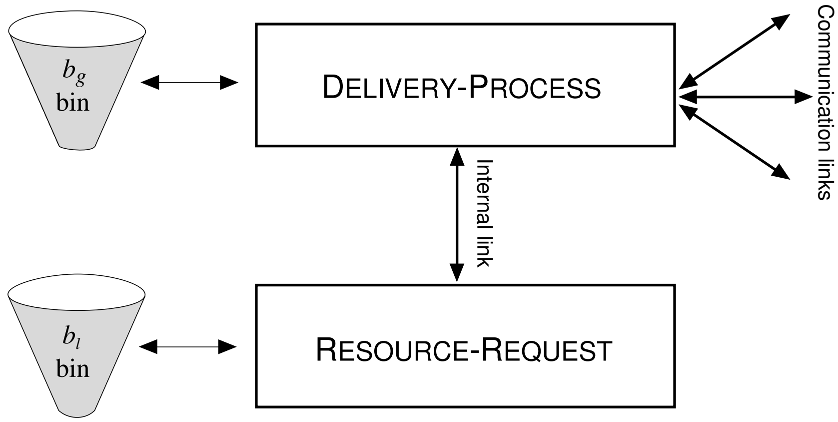

Figure 1: A schematic Diagram of a Basic Controller.

each node holds two bins, there are twice as many nodes in the hierarchy tree as there are nodes in the spanning tree.

Before presenting the specific hierarchy that is used in the Basic Controller implementation, we would like to informally describe the way this bin hierarchy is utilized. Each node has a RESOURCE-REQUEST procedure that controls the local bin b_l of this node, and a DELIVERY-PROCESS that manages the global bin b_g of this node (see Figure 1). In addition to the physical communication links attached to the node, we assume the existence of an "internal link" that connects the RESOURCE-REQUEST and the DELIVERY-PROCESS at every node. Roughly speaking, one can view this as logically splitting each node into two parts connected by the internal link, where one part executes the RESOURCE-REQUEST and the other executes the DELIVERY-PROCESS.

When the controlled algorithm invokes the RESOURCE-REQUEST procedure at node v, this procedure satisfies the request by using the permits from the local bin $b_l(v)$, if this bin is not empty. If $b_l(v)$ is empty, an attempt is made to replenish it from its supervisor bin $\mathit{Supervisor}(b_l)$. This is done by the RESOURCE-REQUEST procedure sending a multi-permit resource-request via the DELIVERY-PROCESS to $\mathit{Supervisor}(b_l)$. If bin $\mathit{Supervisor}(b_l)$ is empty, it is replenished from its supervisor $(\mathit{Supervisor}(\mathit{Supervisor}(b_l)))$, and so on. The root of the hierarchy is the $b_g(r)$ bin at the root of the spanning tree, which initially holds M permits. All other bins are initially empty.

In general, if bin b does not have enough permits to satisfy an incoming request, the algorithm always tries to refill this bin to its capacity and only then tries to satisfy the request. This is done by sending a request to the process that manages the $\mathit{Supervisor}(b)$ bin. When the refill arrives, the pending request is satisfied and the remaining permits are left in bin b. When bin $b_g(r)$ receives a request that it cannot satisfy, the algorithm generates the "resources exhausted" signal.

Before giving a formal description of the hierarchy, we would like to discuss the intuition behind its construction. Replenishment of bin b can be done with a single message propagating from the node holding $\mathit{Supervisor}(b)$ to the node holding b. Intuitively, since we always replenish the bin completely, having large-capacity bins reduces message complexity. Unfortunately, permits that are stored at some point in bins other than the bin $b_g(r)$ might never be used, which implies that we cannot have many large-capacity bins. The idea is to embed the bin hierarchy in a way such that both the bin capacities and the distances in the spanning tree between the node holding bin b and the node holding bin *Supervisor*(b) are exponentially increasing as we climb up in the hierarchy. The parameters in the algorithm are chosen in such a way that the total capacity of the bins (not counting $b_g(r)$) is smaller than W and that the average "cost per permit" in terms of messages is small.

More precisely, for each node $v \in \{V - r\}$ at distance (depth) $D(v)$ from r, the levels of its bins are defined as follows:

Definition 4.1 Levels of bins in node v at depth $D(v)$:

$$\begin{aligned} \textit{Level}(b_g(v)) &= \max\{i \mid 2^i \text{ divides } D(v)\}, \\ \textit{Level}(b_l(v)) &= -1. \end{aligned}$$

For each bin b at node v, consider the path from v to the root r of the spanning tree. We set *Supervisor*(b) to be the bin b' that is closest to b along this path such that *Level*(b') =*Level*$(b)+1$. If no such bin exists along the path, then we set $b_g(r)$ to be *Supervisor*(b). For example, if $D(v) = 52$ (binary 110100), then *Level*$(b_g(v)) = 2$ and *Supervisor*$(b_g(v))$ is the b_g bin that resides in the ancestor of v at depth 40 (binary 101000). Note that this definition implies that the supervisor of bin b_l resides either in the same node as b_l or in the parent of this node in the spanning tree.

The level of a bin (except for the bin at the root) implicitly determines whether this bin is of type b_g or b_l. Henceforth we omit the subscripts g and l unless it might cause confusion.

An example of the hierarchy for the case in which the spanning tree is a chain is given in Figure 2. Circles represent nodes and the numbers inside the circles represent the depth in the spanning tree. The bins of the hierarchy are represented by "buckets" and are shown directly above the nodes in which they reside.

The bin hierarchy has the following properties, which are central to its use in the controller.

Lemma 4.2

1. The depth of the hierarchy tree is at most $\log U + 1$,
2. If *Level*$(b) = l$ then the path from the node that holds bin b to the node that holds bin *Supervisor*(b) has either $\lfloor 2^l \rfloor$ or $\lfloor 3 \cdot 2^l \rfloor$ links,
3. The number of bins at level l that are supervisors (*i.e.*, that supervise at least one bin at level $l-1$) is at most $U/2^{l-1}$.

Proof: The first two parts follow from the definition. To prove the third part, consider two bins b and b', where *Supervisor*$(b) \neq$ *Supervisor*(b'), and both supervisors are at the same level l. The paths from b to its supervisor and the path from b' to its supervisor are node disjoint

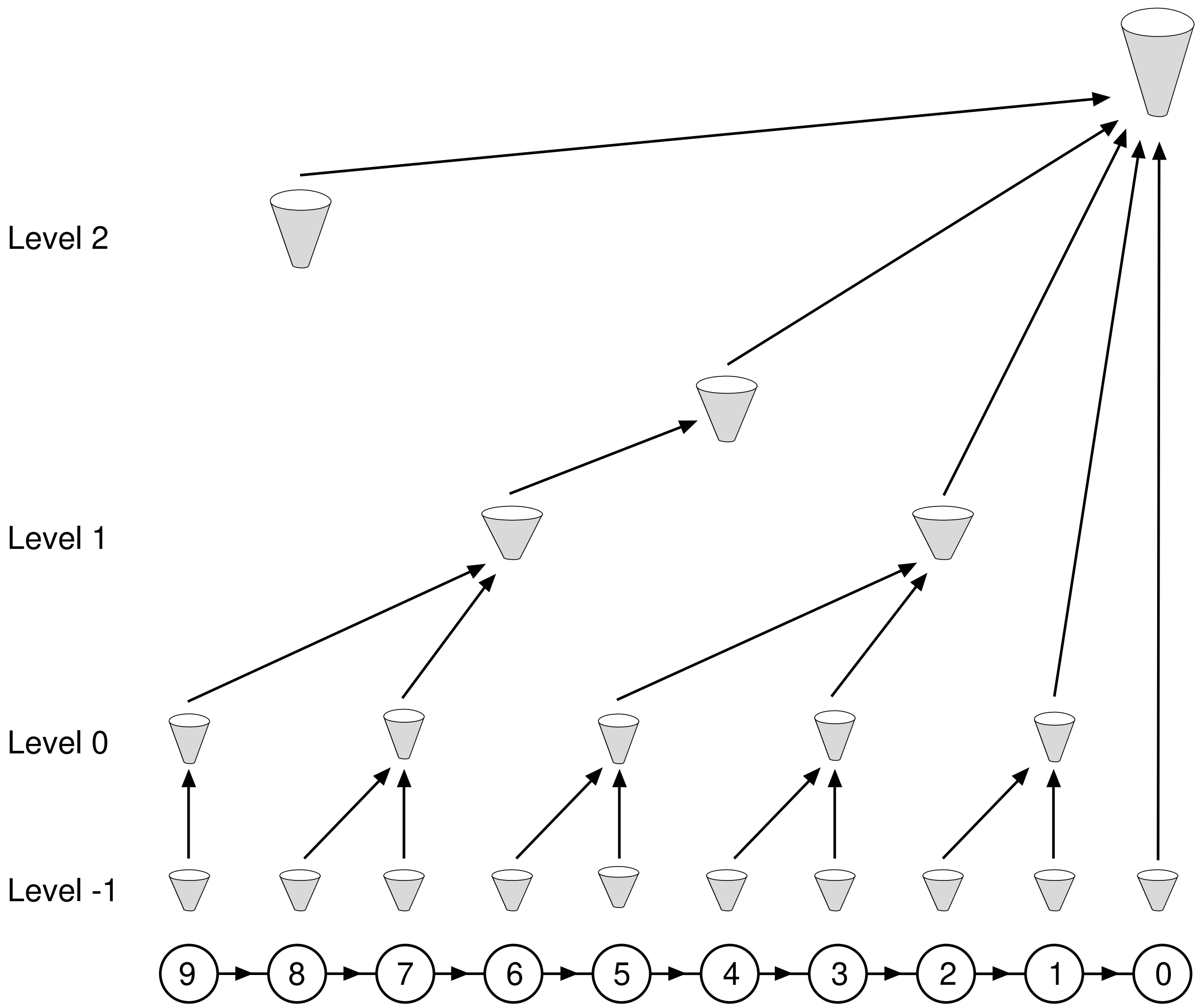

Figure 2: An example of the bin hierarchy for the case when the spanning tree consists of a single path. Circled nodes represent the original nodes of the graph; "buckets" represent nodes of the bin hierarchy.

and each contains at least 2^{l-1} vertices. The claim follows since the total number of nodes is bounded by U. ∎

To ensure that at most W resources are "wasted", we choose the bin capacities in such a way that the number of permits stored in bins outside the root never exceeds W. More precisely, let

$$\Lambda = 2^{\left\lfloor \log\left(\frac{W}{U} \cdot \frac{1}{2(\log U+1)}\right)\right\rfloor}.$$

We define the capacity of bin b at level $Level(b)$ to be

$$\mathrm{Cap}(b) = \max\{\Lambda \cdot 2^{Level(b)}, 1\}. \tag{1}$$

The definitions of bin capacities imply the following lemma, which simplifies the implementation.

Lemma 4.3 The capacity of a bin is either 1 or half the capacity of its supervisor bin.

4.2 Description of the algorithm

In the previous section we defined the bin hierarchy and gave an intuitive explanation of how to implement the Basic Controller using this hierarchy. In this section we show how to map this intuitive explanation onto the underlying message-passing system and present the pseudocode.

The Basic Controller consists of three parts: the Resource-Request procedure shown in Figure 3, the Delivery-Process shown in Figure 4, and the Root-Delivery-Process shown in Figure 5. As we have mentioned above, a copy of the Resource-Request procedure resides at every participating node; it manages the local bin at this node. A copy of the Delivery-Process exists at every node except the root; it manages the global bin b_g at this node. The bin $b_g(r)$ is controlled by the Root-Delivery-Process.

We assume that each node maintains a queue for incoming messages associated with the Basic Controller. Moreover, we assume that we have access to the following system calls:

- Get-Next-Message: Dequeues the message at the front of the node's message queue and returns two variables, the first is the message (MSG), and the second is the number of the link over which the message has arrived (LINK#). If the queue is empty, Get-Next-Message waits until a message arrives. Formally, the syntax is:
  MSG,LINK# $\leftarrow$ Get-Next-Message.

- Wait-For-Message-From-Link(Link#): Removes from the message queue the oldest message that came via link number *LINK#* and returns the content of this message. If no such message exists in the queue, it waits until such a message arrives. Any other message that arrives while this primitive is waiting, is queued in the message queue.

- Send-Message(MSG,Link#): Sends the message *MSG* on link *Link#*. (Used to send messages over the internal link as well.)

Resource-Request: The pseudocode of the Resource-Request procedure is shown in Figure 3. During initialization the size of the local bin is determined, and its content is set to zero. Upon invocation, the procedure tries to satisfy the request from the local bin. If the bin is empty, the Resource-Request sends a message over the internal link to the Delivery-Process at the same node and waits on this link for a reply. As we will see below, the Delivery-Process either returns a "reject" or sends exactly as many permits as can fit into the local (level= -1) bin. Thus, there is no explicit "number of permits" field in the messages that propagate between the Resource-Request procedure and the Delivery-Process. If the Delivery-Process returns a "permit", we refill the local bin, and return "permit" to the caller, decrementing the number of permits in the bin by one. Otherwise, we return a rejection. In order to eliminate unnecessary messages, we continue to reject every request after the first rejection message received from the Delivery-Process.

The requests sent by the Resource-Request procedure are propagated through the local Delivery-Process in order to simplify the code and ensure that only the Delivery-Process

```
/* Initialization */
Permits-In-Bin ← 0;                                        /*Initially the bin is empty.*/
My-Level ← (−1);                                 /*Local bins are on level -1 of the hierarchy.*/
Bin-Capacity ← max{Λ/2, 1};                                 /*Capacity of the local bin*/
Not-Exhausted ← true;                                       /*Resource not exhausted yet.*/

Procedure RESOURCE-REQUEST
    if Permits-In-Bin = 0 & Not-Exhausted
      then begin
            SEND-MESSAGE(My-Level, Internal-Link);                   /*Ask for a refill.*/
            Reply ← WAIT-FOR-MESSAGE-FROM-LINK(Internal-Link);       /*Wait for reply.*/
            if Reply = "permit" then Permits-In-Bin ← Bin-Capacity;
            end;
    if Permits-In-Bin > 0
      then begin
            Permits-In-Bin ← Permits-In-Bin - 1;
            return "permit";                                         /*Issue a permit.*/
      else begin
            Not-Exhausted ← false;
            return "reject";                                         /*Rejection.*/
            end;
    end.
```

Figure 3: The code of the RESOURCE-REQUEST procedure

accesses the physical communication links (as opposed to the internal link). Recall that we have assumed that the controlled algorithm always waits for the RESOURCE-REQUEST to complete before calling it again, thus there can be no concurrent invocations of the RESOURCE-REQUEST procedure at the same node.

Delivery-Process: The pseudocode of the DELIVERY-PROCESS for bin $b_g(v)$ which is at level $Level(b_g(v))$ in the hierarchy appears in Figure 4. Initialization includes determining the capacity of bin $b_g(v)$ and setting its content to zero. Recall that any request that should be satisfied out of $b_g(v)$ is sent from a bin at level $Level(b_g(v)) - 1$. Thus, all such requests are for the same number of permits, which we precompute and store in *Request-Size*.

Once initialized, the DELIVERY-PROCESS enters the message-forwarding loop. Upon receiving a message (on any link), it checks whether it is the addressee of this message, *i.e.,* whether the message comes from a node at level $Level(b_g(v)) - 1$. If not, the message is forwarded through the *Parent-Link* to the node that is the parent of v in the spanning tree. Then, the DELIVERY-PROCESS waits for the permits that are supposed to arrive as a response via the parent of v. All the messages that arrive during this wait are stored in the message queue. When the message carrying the permits (or the rejection) arrives, it is forwarded to the appropriate child.

If the message is addressed to the DELIVERY-PROCESS, it tries to satisfy the request from its bin $b_g(v)$. If this bin is empty and the DELIVERY-PROCESS has not received any "reject" messages addressed to it yet, it sends a request for permits to its supervisor through the *Parent-Link* and waits for a reply. All messages that arrive during this wait are stored in the message queue. If the reply is "permit", $b_g(v)$ is refilled and the pending request is satisfied from it. Otherwise, the DELIVERY-PROCESS returns "reject" and sets a flag to make sure that no more messages are sent to the supervisor.

Root-Delivery-Process: The pseudocode of the ROOT-DELIVERY-PROCESS, the process that manages bin $b_g(r)$ at the root, is shown in Figure 5. It is similar to the DELIVERY-PROCESS executing at the other nodes in the tree with several important differences. First, it has to reply to all the messages since it has no one to forward them to. Second, it generates the "resource exhausted" signal when its bin is depleted.

We start by initializing the bin to M. Upon receiving a message, we compute (from the level of the bin that originated this message) the number of requested permits, and try to satisfy them from the bin. If there are not enough permits, we generate the "resource exhausted" signal and reply with a "reject". The code ensures that the signal is generated only once.

4.3 The basic controller is deadlock free

Since there are "wait" statements in the code it is necessary to prove that the algorithm is deadlock free.

Lemma 4.4 Any "request for permits" message that is generated by b, eventually reaches *Supervisor*(b).

```
Procedure Delivery-Process
/* Code for node v at level Level(v) of the hierarchy. */

Bin-Capacity ← max{Λ · 2^Level(v), 1};          /*Bin capacity of the Delivery-Process bin at node v.*/
Permits-In-Bin ← 0;                               /*Initially, the bin is empty.*/
Not-Exhausted← true;                              /*Initially we assume resource not exhausted.*/
Request-Size ← max{Λ · 2^(Level(v) − 1), 1};     /*Permits will be requested in multiples of Request-Size */

Do forever
    From-Level, Link ← Get-Next-Message;          /*Wait for the next request, which originated from a*/
                                                  /*node at level From-Level and arrived on link number Link.*/
    if Level(v) = From-Level +1
      then begin                                  /*We have to satisfy the request,*/
                                                  /*try to satisfy it from our bin.*/
           if Permits-In-Bin= 0 and Not-Exhausted /*Test whether the bin is empty.*/
             then begin                           /* Bin empty - need a refill.*/
               Send-Message(Level(v), Parent-Link);               /*Request bin replenishment.*/
               Answer = Wait-For-Message-From-Link(Parent-Link) ; /*Wait for the refill.*/
               if Answer = "permit" then Permits-In-Bin ← Bin-Capacity;
               end;
           if Permits-In-Bin > 0
             then begin
               Permits-In-Bin ← Permits-In-Bin − Request-Size;
               Send-Message("permit", Link);                      /*Forward the permits.*/
               end;
             else
               Send-Message("reject", Link);                      /*Forward the rejection.*/
               Not-Exhausted ← false;
               end;
           end;
    else begin
         Send-Message(From-Level, Parent-Link);                   /*Relay the request to the parent.*/
         Answer = Wait-For-Message-From-Link(Parent-Link);        /*Wait for reply,*/
         Send-Message(Answer, Link);                              /*and relay it down.*/
         end;
end.
```

Figure 4: The code of the Delivery-Process.

```
/* Initialization */
Signal ← false;                                              /*Signal not generated yet.*/
Procedure ROOT-DELIVERY-PROCESS
    Permits-In-Bin ← M ;                                     /* All the permits we have.*/
    do forever
    Level, Link ← GET-NEXT-MESSAGE ;
    Requested-Permits = max{Λ · 2^Level, 1};                 /*The number of permits requested.*/
    if Permits-In-Bin ≥ Requested-Permits
       then begin
            Permits-In-Bin ← Permits-In-Bin − Requested-Permits ;
            SEND-MESSAGE("permit",Link);
            end;
       else begin
            SEND-MESSAGE("reject",Link);                     /* The Root-Delivery-Process bin is empty.*/
            if (not Signal)
               then begin
                    generate "resource exhausted" signal;
                    Signal ← true;
                    end;
            end;
    end.
```

Figure 5: The code of the ROOT-DELIVERY-PROCESS.

Proof: Such a message propagates along the path from v to r until it either reaches r, or reaches a bin at level *Level*$(b)+1$, or gets "stuck" (delayed indefinitely) in a message queue of one of the nodes along the path. By the definition of the hierarchy, the node that stops propagating the message and tries to satisfy it is indeed *Supervisor*(b).

We claim that a message cannot be delayed indefinitely in one of the message queues. Indeed, let v be the node with the smallest $D(v)$ that has a message indefinitely delayed in its queue. From the pseudocode we can see that the ROOT-DELIVERY-PROCESS never waits, and hence indefinite delay can happen only if one of the invocations of *Wait-For-Message-From-Link* by a DELIVERY-PROCESS at $v \neq r$ does not return, *i.e.,* there is no reply to the message sent by *Send-Message* from v to v's parent just before calling the *Wait-For-Message-From-Link*. But this is impossible since v is the node with the smallest $D(v)$ that has a message "stuck" in its queue. ∎

The above lemma together with the observation that the propagation of messages down the tree cannot be delayed, implies the following lemma.

Lemma 4.5 Every call to a RESOURCE-REQUEST procedure eventually returns either a permit or a rejection.

Remark: Consider a bin b_g with capacity $\mathrm{Cap}(b_g) > 1$. Lemma 4.3 implies that any request that has to be satisfied from this bin asks for exactly $\mathrm{Cap}(b_g)/2$ permits. Therefore, exactly two requests can be satisfied for each replenishment of b_g, and thus there is no need to represent explicitly the number of permits stored in b_g. All we need is a flag that says whether the bin is empty or half full. A bin with capacity 1 can satisfy only one request; hence it merely serves as a relay of requests and permits between the bins it supervises and its supervisor. The only bins whose content need to be explicitly represented are the b_l bins, controlled by the RESOURCE-REQUEST procedures, and the $b_g(r)$ bin, controlled by the ROOT-DELIVERY-PROCESS.

4.4 Analysis of the Basic Controller

In this section we show that the Basic Controller implementation described above satisfies the conditions of Section 3. The legality of executions follows directly from the construction. In particular, it follows from the fact that the Basic Controller never initiates any actions on its own. The RESOURCE-REQUEST returns a rejection only if it got a "reject" message from the DELIVERY-PROCESS. The fact that DELIVERY PROCESS at level l produces a reject message only if it got a reject message from a process at level $l+1$ implies that the first reject message is generated by the ROOT-DELIVERY-PROCESS, *i.e.,* the second condition in (3.1) is satisfied. Thus, it remains to show that the Basic Controller satisfies the first and the third conditions. Informally, these conditions state that the total number of issued permits does not exceed M and that no requests are rejected unless the number of issued permits exceeds $M - W$.

Lemma 4.6 At most M permits are issued.

Proof: The claim follows from the fact that initially all bins except bin $b_g(r)$ are empty and that bin holds M permits. ∎

Lemma 4.7 If during the execution of the BASIC CONTROLLER the number of participating nodes does not exceed U then the total number of permits in bins outside the root is bounded by W.

Proof: Initially all bins are empty except the global bin at the root. For a bin to become nonempty, it has to receive[1] a request for a permit. This can happen only to a b_g bin that is a supervisor of some bin or to a b_l bin. Moreover, if the capacity of a bin is equal to 1, the bin always remains empty since it serves only as a relay between the node it supervises and its supervisor. Using Lemma 4.2 and the definition of the bin capacity (Equation (1)), the total capacity of supervisor bins with capacity of at least 2 at level L is at most

$$\frac{U}{2^{L-1}} \cdot \Lambda \cdot 2^L \leq \frac{W}{\log U + 1}$$

Recall that the level of any bin b_l is defined to be (-1), implying that the capacity of b_l is equal to $\max\{\Lambda/2, 1)$. Hence, if the capacity of each bin b_l is above 1, the total capacity of these bins is

$$U \cdot \frac{\Lambda}{2} \leq \frac{W}{\log U + 1}$$

Thus, the total capacity of bins at any level that has bins with capacity above 1 is bounded by $\frac{W}{\log U+1}$. The claim follows since, by Lemma 4.2, the number of levels (apart from the root) in the bin hierarchy is at most $(\log U + 1)$. ▮

Lemma 4.8 If a legal execution of the BASIC CONTROLLER executing on at most U nodes includes a rejected request, then it includes at least $M - W$ satisfied requests.

Proof: Consider the moment when the last one of the messages sent by the BASIC CONTROLLER is received. (The fact that the number of these messages is finite is proved below in Lemma 4.9.) At this time every permit has either been issued or is stored in bins of non-unit capacity. The fact that there exists a rejected request implies that the bin of the root is empty. Hence, the number of issued permits is at least M minus the total number of permits stored in the rest of the bins. The claim follows from Lemma 4.7. ▮

Lemma 4.9 The message complexity of the BASIC CONTROLLER is $O(\frac{M}{W} U \log^2 U)$.

Proof: The algorithm sends only three types of messages: requests, permits, and rejections. Observe that the DELIVERY PROCESS running at node v does not send a new request for permits until its previous request was fulfilled or rejected. Moreover, it never tries to refill a bin after getting the first rejection. This implies that the number of requests sent from b to *Supervisor*(b) is at most one more than the number of permit messages sent from *Supervisor*(b) back to b. Therefore it suffices to bound the number of permit messages.

[1] In order to simplify notation, we say that "bin b sends a message to b'" when we mean that "the process controlling b sends a message to the process controlling b'".

The total number of permits is at most M. The number of permits in any message sent by b' to b where $b' = \mathit{Supervisor}(b)$ is equal to the capacity of b, given by Equation (1). Hence, the total number of permit messages received by bins at level $\mathit{Level}(b)$ is at most

$$\left\lceil \frac{M}{\mathrm{Cap}(b)} \right\rceil = \left\lceil \frac{M}{\max\{\Lambda \cdot 2^{\mathit{Level}(b)}, 1\}} \right\rceil \leq 2\frac{M}{W} U(\log U + 1) 2^{-\mathit{Level}(b)}$$

By Lemma 4.2, each one of these messages travels a distance of at most $3 \cdot 2^{\mathit{Level}(b)}$. The claim follows since, by Lemma 4.2, the depth of the hierarchy tree is at most $(\log U + 1)$. ▮

A node queues all the messages from its children while waiting for a permit (or a reject) from its parent. Since the Basic Controller never sends a new message on a link to the parent before getting a reply to the previous message it sent up, the number of messages queued at a node is bounded by its degree in the spanning tree. A request message propagating up the tree from b to $\mathit{Supervisor}(b)$ consists only of a single field, containing the level of b, which can be represented by $O(\log \log U)$ bits; the permit and reject messages propagating down the tree can be represented by $O(1)$ bits. Thus, we have the following lemma.

Lemma 4.10 The bit complexity of the Basic Controller is $O(\frac{M}{W} U \log^2 U \log \log U)$ and memory requirements are $O(\log \log U)$ bits per link plus $O(\log M)$.

Remark: We have described a version of the controller which has very low memory requirements at each node and very low bit message complexity. This is accomplished by allowing each bin to have at most one outstanding request at a time. The drawback of this approach is that if requests occur at many different nodes that share a supervisor, then this supervisor becomes a bottleneck and some of these nodes may wait a long time for a response. An alternative version of the controller can be devised which allows a bin to make multiple requests, in order to reduce the delay incurred at any given node. This approach requires more memory and has higher bit message complexity, because we now must tag messages by the bin at which they originate. The details are straightforward, and hence are omitted.

5 Main Controller

The previous section presented the Basic Controller algorithm which assumed an a priori knowledge of an upper bound U on the number of participating nodes. The complexity of the Basic Controller is a function of this upper bound regardless of the actual number of participating nodes. In this section we show how to relax this assumption and describe the Main Controller whose complexity depends only on the actual number of participating nodes.

The basic idea is to run two Basic Controllers concurrently. Controller-N monitors and controls the number of participating nodes; Controller-R monitors and controls the resource consumption. We require that a new node that wishes to join the controlled protocol does not join until it gets a permit to do so from Controller-N. In fact, we need to slightly modify Controller-N. When its bin at the root becomes empty, it does not start sending "reject" messages. Instead, it produces the "resource exhausted" signal, sending it to the Main Controller.

The Main-Controller, invoked with parameters (M, W), proceeds in iterations. The idea is to use Controller-N to terminate the iteration when the number of nodes has at least doubled compared to the number of nodes in the beginning of the iteration. More precisely, denote the parameters of Controller-N and Controller-R during iteration i by (M_i^N, W_i^N) and (M_i^R, W_i^R), respectively. (Their numerical values will be defined below.) Denote by U_i the upper bound on the number of participating nodes in iteration i; the value of U_i is used both by Controller-N and Controller-R. Let M_i^R be the number of unused permits at the beginning of iteration i, initially $M_1^R = M$, and let n_i be the number of nodes participating in the controlled algorithm at the beginning of iteration i. At the start of iteration i, we set the parameters as follows:

$$\begin{array}{lcll} (M_i^N, W_i^N) & = & (2n_i, n_i) & \text{(parameters of Controller-N)}, \\ (M_i^R, W_i^R) & = & (M_i^R, W) & \text{(parameters of Controller-R)}, \\ U_i = 3n_i. & & & \end{array}$$

Note that the "waste" parameter of Controller-R does not depend on the iteration number. If $M_i^R \leq W$, Main-Controller produces a "resource exhausted" signal at the root. If, during iteration i, Controller-R produces a "resource exhausted" signal, the Main-Controller produces "resources exhausted" signal as well. Otherwise, the iteration continues until Controller-N produces the signal. When this happens, the root initiates a "broadcast and echo" on all the participating nodes, making sure that all permits given out during this iteration have reached their destinations and counting the current number of participating nodes n_{i+1}. Then it collects the contents of all the bins (including the root bins) that belong to Controller-R and sets M_{i+1}^R to be equal to this value.

Recall that Controller-N never sends rejections. Thus, at the end of an iteration, there might be nodes waiting for a permit to join the controlled protocol. Before proceeding with iteration $(i+1)$, the Main Controller uses the broadcast-and-echo to identify unfulfilled requests of Controller-N at iteration i and resubmits them to the new incarnation of Controller-N at the next iteration.

Lemma 5.1 The Main-Controller satisfies the "consumption conditions" (3.1).

Proof: Properties (1) & (2) follow from the fact that the Main Controller never generates new permits and from the correctness of the Basic Controller. Iteration i starts with n_i participating nodes. Since Controller-N allows at most $2n_i$ new nodes to join the controlled protocol during iteration i, the total number of participating nodes during this iteration is bounded by $3n_i$. Thus, U_i is a valid upper bound on the number of nodes participating in the algorithm during iteration i. This fact, combined with Lemma 4.8, implies property (3). ∎

Theorem 5.2 The message complexity of the Main Controller is $O(\frac{M}{W} n \log^2 n)$ and the bit complexity is $O(\frac{M}{W} n \log^2 n \log\log n)$, where n is the final number of participating nodes.

Proof: The message complexity of the MAIN CONTROLLER is the sum of the message complexities of CONTROLLER-N and CONTROLLER-R. In iteration i, these complexities are $O(U_i \log^2 U_i)$ and $O(\frac{M_i}{W} U_i \log^2 U_i)$, respectively. By construction, $M \geq M_i^R \geq W$. Moreover, for each i, $3n_{i-1} \geq n_i \geq 2n_{i-1}$ and $U_i = 3n_i$. The bound follows. Note that the communication complexity of collecting the contents of all the bins over all the iterations is at most $O(n \log n)$ messages or $O(n \log^2 n)$ bits. ∎

6 Extensions

In this section we describe several controllers that are extensions of the MAIN CONTROLLER.

The "zero waste" case

Some applications require zero "waste", *i.e.,* $W = 0$. For example, we might want to terminate an algorithm when the number of participating nodes reaches *exactly* M. If we have a controller that works with constant W, we can distribute the remaining permits by collecting them using a broadcast-and-echo technique and then using the naive controller, described in the beginning of Section 4. Hence, in the presentation below, we will assume that $W \geq 1$. Observe that employing the MAIN CONTROLLER in cases where W is small compared to M leads to a large complexity.

In general, in order to deal with the cases where M/W is large, we use the MODIFIED CONTROLLER. It iterates the MAIN CONTROLLER $O(\log \frac{M}{W+1})$ times. In each iteration the "waste" is at least halved. That is, we set $M_0 = M$. In the ith iteration we execute the MAIN CONTROLLER with parameters $(M_i, M_i/2)$. When it terminates the root performs a "broadcast and echo" to count the number of unused resources, which is $M_{i+1} \leq M_i/2$. After $O(\log \frac{M}{W+1})$ iterations, the number of remaining unused resources is within a constant multiplicative factor of W, and we can use the MAIN CONTROLLER directly. Hence, we have the following lemma.

Lemma 6.1 The communication complexity of the MODIFIED CONTROLLER is $O(n \log^2 n \log \frac{M}{W+1})$.

Dynamic Name Assignment

Many protocols assume that each node in a network has a unique name (ID) and use these names to break symmetry. The bit message complexity of these protocols is expressed in terms of the number of bits needed to represent a name, which is usually assumed to be equal to the logarithm of the number of nodes (see *e.g.* [3, 10, 11, 12]). This assumption is correct only if at least a constant fraction of the total number of named nodes are participating in the protocol, which may not always be true.

We define the *Dynamic Name Assignment* problem as follows. As in the case of the MAIN CONTROLLER, we consider a single-initiator protocol (the extension to multiple initiators is not difficult) that executes in a large network and dynamically activates new nodes, which start participating in the protocol. The goal is to assign unique integer names to all the participating nodes, such that the largest name will be at most a constant factor larger than n, the number of participating nodes.

To solve this problem we use the BASIC CONTROLLER with the following change: the "permit messages" carry the *range of allocated names* instead of carrying permits. Note, that as opposed to the case of the BASIC CONTROLLER, every message in the Dynamic Name Assignment algorithm carries a range of names, and hence it is $O(\log n)$ bits long. Moreover, since only contiguous ranges can be represented concisely, we cannot reuse names that are in the bins when the BASIC CONTROLLER is re-initialized (which happens each time the number of participants has approximately doubled). This increases the number of unused names, but only by a constant factor.

Lemma 6.2 The message complexity of the Dynamic Name Assignment algorithm is $O(n \log^2 n)$ with messages of $O(\log n)$ bits, where n is the final number of participating nodes.

Proof: Similar to Lemma 5.2. ∎

Distributed Bank

The MAIN CONTROLLER described in the previous sections deals with resources that can be only consumed. Here we sketch the main ideas of how to extend the MAIN CONTROLLER to the case where resources are both consumed and generated.

A Distributed Bank Controller with parameter W is a distributed algorithm which interacts with the controlled algorithm via the DEPOSIT-RESOURCE and WITHDRAW-RESOURCE procedures at the nodes. As in Section 3, each execution can be associated with an ordered sequence of events σ, where the events can be "deposit-request", "deposit-ack", "withdraw-request", and "withdraw-grant". In what follows, unless it causes confusion, we refer to "withdraw-request" and "deposit-request" as requests and to "deposit-ack" and "withdraw-grant" as replies. As before, we use $\sigma(v)$ to denote the subsequence of events in σ that occured at node v. A sequence is considered legal only if for all v, the corresponding $\sigma(v)$ does not contain two adjacent requests. In other words, the algorithm that uses the distributed bank object should wait for a reply to each one of the requests before issuing a new one. In addition, the following properties should be satisfied:

1. For every v, $\sigma(v)$ contains a corresponding reply immediately after every request; the only exception can be the last "withdraw-request" in $\sigma(v)$.
2. In any prefix of σ, the total number of withdrawal requests granted is at most the total number of deposit requests.
3. If a "withdraw-request" does not have a corresponding "withdraw-grant" in σ, then the number of "deposit-request" events less the number of "withdrawal-grant" events in σ is below W.

A Distributed Bank Controller can be built by combining two MAIN CONTROLLERS, where one controls the withdraw requests and the other controls the deposit requests. The main modification that is needed is to change the controller responsible for withdrawals to never return a rejection. Instead, it should try to satisfy the requests from the root bin of the controller responsible for the deposits, waiting until this bin contains a sufficient number of deposits.

Let $\mathcal{M}$ be the total number of withdraw and deposit requests made.

Lemma 6.3 The message complexity of the Distributed Bank Controller is $O(\frac{\mathcal{M}}{W} n \log^2 n)$ and the bit complexity is $O(\frac{\mathcal{M}}{W} n \log^2 n \log\log n)$.

Proof: Similar to the proof of Lemma 4.9 and Theorem 5.2. ∎

7 Conclusions

The search for the right set of paradigms for designing efficient distributed algorithms is an important task of the theory of distributed computation. The resource controller object described in this paper has already been used as an integral part of several distributed algorithms [1, 6]. The controller is simple to implement and the constant factors involved in the complexity computations are quite small. We believe that the Resource Controller described in this paper can serve as a convenient building block for design of efficient distributed algorithms that have to deal with global resources.

Acknowledgments

The authors wish to express their thanks to Paul Beame, Eli Gafni, Andrew Goldberg, Oded Goldreich, Joe Green, Michael Merritt, and Silvio Micali for useful discussions. We would also like to thank the referees whose comments have greatly improved the presentation of the paper.

References

[1] Y. Afek, B. Awerbuch, and E. Gafni. Applying Static Network Protocols to Dynamic Networks. In *Proc. 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 358–370, 1987.

[2] Y. Afek, B. Awerbuch, S. Plotkin, and M. Saks. Local Management of a Global Resource in a Communication Network. In *Proc. 28th IEEE Annual Symposium on Foundations of Computer Science*, pages 347–357, 1987.

[3] Y. Afek, G. M. Landau, B. Schieber, and M. Yung. The power of Multimedia: Combining Point-to-Point and Multiaccess Networks. In *Proc. 7th Annual ACM Symposium on Principles of Distributed Computing*, August 1988.

[4] Y. Afek and M. Saks. Detecting Global Termination Conditions in the Face of Uncertainty. In *Proc. 6th Annual ACM Symposium on Principles of Distributed Computing*, August 1987.

[5] B. Awerbuch. Complexity of network synchronization. *J. Assoc. Comput. Mach.*, 32:804–823, 1985.

[6] B. Awerbuch. On the effects of feedback in dynamic network protocols. In *Proc. 29th IEEE Annual Symposium on Foundations of Computer Science*, pages 231–245, October 1988.

[7] B. Awerbuch and S. Plotkin. Approximating the size of a dynamically growing distributed network. Technical Report MIT/LCS/TM-328, M.I.T., April 1987.

[8] R. Bar-Yehuda and S. Kutten. Fault toletant distributed majority commitment. *J. of Algorithms*, 9:568–582, 1988.

[9] E. W. Dijkstra and C. S. Scholten. Termination detection for diffusing computations. *Information Processing Let.*, 11:1–4, 1980.

[10] R. G. Gallager, P. A. Humblet, and P. M Spira. A Distributed Algorithm for Minimum-Weight Spanning Trees. *ACM Transactions on Programming Languages and Systems*, 5:66–77, 1983.

[11] A. Goldberg and S. Plotkin. Parallel $(\Delta + 1)$ Coloring of Constant-Degree Graphs. *Information Processing Letters*, 25(4):241–245, June 1987.

[12] A. V. Goldberg, S. A. Plotkin, and G. E. Shannon. Parallel Symmetry-Breaking in Sparse Graphs. *SIAM J. Disc. Math.*, 1:434–446, 1988.

[13] N. A. Lynch, N. D. Griffeth, M. J. Fischer, and L. J. Guibas. Probabilistic Analysis of a Network Resource Allocation Algorithm. *Information and Control*, 68:47–85, 1986.