数理解析研究所講究録
950 巻 1996 年 106-112

Maintaining a Dynamic Set of Processors in a Distributed System

Satoshi Fujita[†] Masafumi Yamashita[†]

Abstract

Consider a distributed system consisting of a set V of processors, and assume that every pair of processors can directly communicate with each other. A simple scheme is proposed, for keeping a dynamic set $U \subseteq V$ of processors in a distributed manner. The dynamic set supports the following three basic operations: Insert *inserts the caller itself in U, and* Delete *removes the caller itself from U.* Find *searches a processor in U. To demonstrate the efficiency of scheme, an amortized analysis of the message complexity of operations is performed; each operation requires $9 + 3\log_2(|V| - 1)$ messages.*

The proposed scheme can be applied to many important problems: e.g., to the load balancing problem by simply using the set as a "free list", to the mutual exclusion problem by constructing a circular list in which a token is circulated, and to construct another data structure like FIFO queue.

1 Introduction

Data structures play an important role in designing time efficient algorithms. In a similar sense, "processor structures" can be used to design communication efficient distributed algorithms. For example, many of the routing and broadcasting problems can be solved efficiently, by letting the processors maintain a *spanning tree* of the network. Also the spanning tree can be used to efficiently solve other problems such as the leader election problem (e.g., the extrema-finding problem) [1, 3, 5, 7]. An *in-tree* is used to solve the mutual exclusion problem [2, 10, 11], and the decentralized object finding problem [4].

Unlike a data structure used in a sequential algorithm, a processor structure is (of course) shared by the processors in a distributed system. However, it does not mean that every processor must know the whole structure. Rather, each processor usually maintains only a local structure, and the whole structure is kept consistent, nevertheless. For example, a processor recognizes only the set of local ports corresponding to the tree edges incident on it (rather than all tree edges), for maintaining a spanning tree.

This paper discusses how to maintain a **dynamic set** of processors. The dynamic set supports the following three operations:

- Insert: This operation adds the processor that calls the operation in the set as a new element.
- Delete: This operation removes the processor that calls the operation from the set.
- Find: This operation returns the identifier of a processor in the set.

The message complexity of each of the operations is very small. To observe it, we perform an amortized analysis which is based on the amortized analysis invented by Ginat *et al* [6], and show that only $9 + 3\log_2(|V| - 1)$ messages are required per operation.

The main idea of the scheme is to maintain a dynamic set U of processors in the form of a distributed circular list: Each processor has a local register to store the *next* pointer to "next processor". We naturally identify the distributed system with a directed graph G with the node set being the set of processors, i.e., there is a directed edge from u to v iff the next pointer of u points to v. Dynamic set U is maintained in such a way that all processors in U are included in a unique directed cycle C in G, and that $G - C$ are in-trees, each of whose sinks has the next pointer to a processor in C.

C may include processors not in U, since deletion is simply achieved by marking itself in most of the cases. Removal of marked processors from C will be done later, when Find is executed. Insert and Find called by a processor not in C, say u, follow the directed path starting

[†]Department of Electrical Engineering, Faculty of Engineering, Hiroshima University, Kagamiyama 1-4-1, Higashi-Hiroshima, 739 Japan.

with u, until it encounters a processor in cycle C, say v, and for Insert, u and v update the next pointers so as to insert u as the next processor of v. The message complexity of those operations therefore mainly depends on the length of directed path the caller traverses. To shorten the path, we adopt the heuristic of path compression used in the Union-Find algorithm [8].

The data structure proposed in this paper can be used to solve many important problems: To solve the load balancing problem, we can simply regard the dynamic set as the "free list". To solve the mutual exclusion problem, we circulate a single token along C and regard it as a token ring system. We can further use it to construct another data structure like FIFO queue.

The paper is organized as follows. In Section 2, we propose a basic scheme for sharing a set in a distributed manner. In Section 3, we show several applications of the scheme. Section 4 concludes the paper with future problems.

2 The Scheme

2.1 The Model

In what follows, we call a processor a *node*, since we will identify a distributed system with a directed graph with the node set being the set of processors (in the sense we explained in Section 1). Consider a distributed system with a node set $V = \{0, 1, \ldots, n-1\}$. We assume that every pair of nodes in V can directly communicate with each other in the blocking mode, i.e., each message transfer is synchronized using the hand shaking. A local area network connected by a shared bus (e.g., Ethernet) is an example of systems satisfying the assumption on communication.

Let U ($\subseteq V$) be the subset of nodes that satisfy an arbitrary fixed property $\mathcal{P}$. Suppose that each node in V knows if it is a member of U, and that the membership to U may dynamically change. This paper discusses how to implement the three operations Insert, Delete and Find efficiently, with respect to such a dynamic set U. The efficiency is measured by the message complexity, i.e., the number of messages exchanged.

2.2 Data Structure

In the proposed scheme, each node $v \in V$ has a local register r_v to store the *next* pointer that points to a node

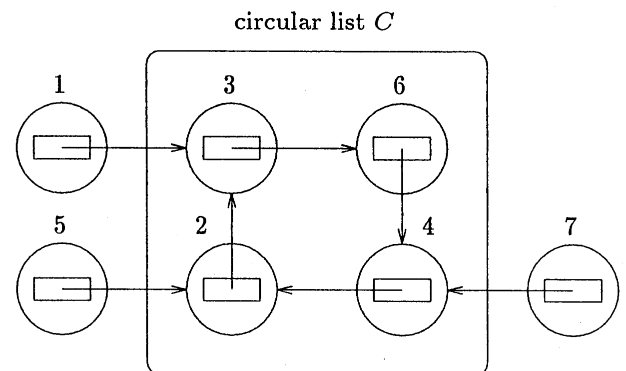

Figure 1: An example of graph G.

in V. (We use two more registers t_v and m_v. We explain about them later.) The node pointed by r_v is called the *next* node of v. By associating a directed edge from v to r_v with each ordered pair (v, r_v), we naturally obtain a directed graph G, in which every node has exactly one outgoing edge. Figure 1 illustrates an example of G, in which, for example, node 1 points to node 3 (i.e., $r_1 = 3$) and node 3 points to node 6 (i.e., $r_3 = 6$). Graph G is dynamic in the sense that its configuration dynamically changes, since nodes autonomously update their local registers.

In our scheme, graph G is always kept connected. It therefore consists of a unique directed cycle and a set of in-trees whose sinks point to a node in the cycle, since every node in G has exactly one outgoing edge. Further, the cycle is maintained so as to include all nodes in U. We implement the dynamic set U in a form of distributed circular list occured in G as the cycle. As you will see, the circular list may contain nodes not in U, since, in many cases, Delete only marks the caller to distinguish it from the nodes in U and the actual deletion process from the circular list is postponed until Find is executed. Local register (flag) m_v is used to memorize if v is marked or not. That $v \in U$ iff $m_v = 0$ is intended to hold.

We create a single token to show the "anchor" node, and let it circulate along the circular list. Local register (flag) t_v is used to memorize if v has the token. That v is the anchor iff $t_v = 1$ is intended to hold. The anchor is handled in such a way that it is not marked unless $U = \emptyset$. The role of the anchor is two folds. First, it is

used to guarantee the deadlock-freedom by using it as an arbiter. Next, it is used to check whether $U = \emptyset$ or not.

2.3 Primitive Operations

The system is initialized as follows, assuming that initially $U = V$ holds. Hence, for each $v \in V$, $m_v = 0$, initially. Local registers r_v are initialized to $r_v := v + 1 \pmod n$ for each $v \in V$, i.e., the corresponding G is a single directed cycle. The token is initially given to node $0\ (\in V)$. Hence, initially, $t_v = 1$ iff $v = 0$. In the following, we present three procedures Find, Insert and Delete that are implementations of corresponding primitive operations, respectively. Those procedures are assumed to be executed as atomic ones, in the sense that once the execution starts, the processor is dedicated to execute it, until it finishes. The only exception is the case in which Delete is initiated by the anchor; the anchor may handle a message that Delete does not expect before it finishes, to avoid possible deadlocks.

2.3.1 Procedure FindNode

We first present a procedure FindNode that is used as a central subroutine in the implementations of the three operations. When a node v calls FindNode, it returns w, r_w, t_w, m_w and S, where w is either the first node in U appeared in the unique directed path $P_v(G)$ from v (if $U \neq \emptyset$), or the anchor node (otherwise). Here S is the set of nodes appeared in $P_v(G)$, excluding v and w, and is used to apply the heuristic of path compression later.

If a returned value $m_w = 1$, i.e., if FindNode cannot find a node in U, $U = \emptyset$ and therefore w is the anchor, i.e., $t_w = 1$. FindNode is given in Figure 2.

Each node u on the path traversed, upon receiving a message M' from v, acts as follows:

- If $M' =$ `inquire`, then
  - if $m_u = 1$ and $t_u = 0$, i.e., if $u \notin U$ and is not the anchor, then it replies `skip_me`(r_u) to v and *blocks* the execution of u, until it receives message `contract` from v.
  - else it replies `found`(r_u, t_u, m_u) to v, and *blocks* the execution of u until it receives a message from v.

```
procedure FindNode { For initiator v. }
begin
    S := ∅; { Nodes to be contracted. }
    w := r_v;
    repeat
        Send(inquire, w);
        Receive(M, w);
        if M = skip_me(r_w) then
            S := S ∪ {w} and w := r_w
    until M = found(r_w, t_w, m_w);
    return w, r_w, m_w and S
end
```

Figure 2: Procedure FindNode.

- If $M' =$ `contract`(w), then $r_u := w$.
- If $M' =$ `place_token`, then $t_u := 1$. { Used in Delete. }
- If $M' =$ `remove_token`, then $t_u := 0$ and $r_u := v$. { Used in Insert. }
- If $M' =$ `unblock`, then it *unblocks* the execution of u, but does nothing else.

All nodes traversed by FindNode block themselves until a further instruction (message) arrives from v, which will be issued in the procedure that calls FindNode as a subroutine. Note that for the simplicity of description, v may send or receive a message to or from v itself. In this case, we assume that v behaves like other u (without actually exchanging messages), except that v does not block itself.

2.3.2 Procedure Find

Suppose that a node v wishes to find a node $u \in U$ and calls Procedure Find. If $m_v = 0$, it returns v itself, since $m_v = 0$ implies $v \in U$. If $t_v = m_v = 1$, it returns *Fail*, since $U = \emptyset$ in this case. When v is neither an element in U nor the anchor, Find returns a $u \in U$ as long as $U \neq \emptyset$. If $U = \emptyset$, *Fail* is returned. The node u found is the first node in U appeared in the directed path $P_v(G)$ from v. Find is given in Figure 3.

2.3.3 Procedure Delete

Suppose that a node v wishes to delete itself from U and calls Procedure Delete. If v is not the anchor, i.e., if

```
procedure Find { For initiator v. }
begin
        if m_v = 0 then return v;
        if t_v = 1 then return Fail;
        Call FindNode;
        { It returns w, r_w, t_w, m_w, and S. }
        r_v := w; { r_v points to the found node. }
        Send(contract(w), x) for all x ∈ S;
        { Path compressed. }
        if m_w = 0 then return w
        else return Fail
end
```

Figure 3: Procedure Find.

```
procedure Delete { For initiator v with m_v = 0.}
begin
        if t_v = 0 then m_v := 1 and terminate;
        Call FindNode;
        { It returns w, r_w, t_w, m_w, and S. }
        r_v := w;
        if w ≠ v then t_v := 0 and
        Send(place_token,w); { Send token. }
        m_v := 1; { Mark itself. }
        Send(contract(w), x) for all x ∈ S;
        { Path compressed. }
        Send(unblock, w)
end
```

Figure 4: Procedure Delete.

$t_v = 0$, then the deletion is simply achieved by marking itself.

Otherwise, if it is the anchor, Delete first finds a node $u \in U$ by calling FindNode, transfers the token to u, marks itself, and then the path compression is applied. As mentioned, if a message is waiting for being processed when the anchor v is executing FindNode, suspending FindNode, it first handles the message, which belongs to another execution instance of operation initiated by a node v', in order to guarantee deadlock-freedom. The message must be an `inquire`. When v receives the `inquire`, it freezes the execution of FindNode after applying the path compression for nodes in the current S, and responds to the `inquire` first. FindNode resumes control when v finishes the execution of instruction given by v'. A more formal description of this "interrupt handler" is given in below.

- If `inquire` arrives from v' while waiting for a reply from a node w, then Send (`contract`$(w), x$) for all $x \in S$, $S := \emptyset$, and Send(`noblock`$, w$) (i.e., it "flushes" the current S). The message arriving from w is discarded. It then replies `found`(r_v, t_v, m_v) to v' (since it does not complete the deletion), and *blocks* itself until it receives a further instruction (including `unblock` instruction) from v' and finishes executing it. Finally, v resumes the execution of `FindNode` from the point it is suspended.

- If `inquire` arrives from v' immediately after receiving `skip_me`(r_w) from w (before sending out another `inquire`), it first executes $S := S \cup \{w\}$ and $w := r_w$, and then does the sequence of instructions given for the above case.

- Finally, if `inquire` has already arrived when v enters FindNode, it simply delays FindNode and processes the `inquire` first.

FindNode finds v itself as w, when $U = \{v\}$. In this case, as the result, $t_v = m_v = 1$ holds. Delete is given in Figure 4.

2.3.4 Procedure Insert

Suppose that a node v wishes to add itself in U and calls procedure Insert. It first looks for a node w in U by calling FindNode. Then it inserts v in the circular list as the next node of w. If $t_w = m_w = 1$, which implies that $U = \emptyset$, the anchor is transferred to v from w. A formal description of Insert is given in Figure 5.

2.4 Observing Correctness

First, observe that the circular list C contains every node in U all the time. To this end, we observe that all nodes not in C are marked. Every node is not marked and resides in the circular list C at the time of initiation. Suppose that an unmarked node u exists outside of C at some time instant. Then either u was removed from C despite that it was not marked, or its mark was removed despite that it was outside of C. The latter case never occur since the removal of mark occurs only in Insert and it always inserts the caller as the next node of the node in U that FindNode found. Let us check that the former case does not occur, either. This occurs only

procedure Insert { For initiator v with $m_v = 1$.}
begin
Call FindNode;
{ It returns w, r_w, t_w, m_w, and S. }
$m_v := 0$; { Remove mark. }
if $t_w = m_w = 1$ **then**
Send(remove_token,w), $r_v := v$ and
$t_v := 1$ { Token Received. }
else $r_v := r_w$ and Sendcontract(v), w);
{ v inserted. }
Send(contract(w), x) for all $x \in S$
{ Path compressed. }
end

Figure 5: Procedure Insert.

when a contract message is sent to an unmarked node. However, FindNode always returns a set S of marked nodes. Therefore, C includes all nodes in U.

Second, we observe that the graph G is weakly connected, i.e., the procedures never generate a new directed cycle besides C. To this end, we examine the following claims:

1. At any time instant, C contains the anchor.
2. Any contract message to a node x asks for setting its next pointer to a node in C.

The first claim holds, since the anchor is in C at the initiation time, and it is transferred to a node found by FindNode, i.e., to a descendant of the current anchor. The second claim also holds, since in each procedure, each of contract messages takes w as its argument, where w is the node found by FindNode (cases of Find and Insert), or a node reachable from the anchor (the case of Delete initiated by the anchor). In both cases, w is a node in C. Hence the anchor is reachable from every node in G at any time instant, or in other words, G is weakly connected.

Finally, we observe the deadlock-freedom of the scheme. Suppose that a deadlock occurs. Then there are a set D of nodes who are executing FindNode, and each of search paths encounters a node already blocked by another search path. Since the number of outdegree of every node is 1, it implies that every node in C belongs to one of the search paths (because there is only one cycle in G). However, this is a contradiction, since an execution instance of FindNode sends an inquire to the anchor, and it terminates by finding the anchor.

The following theorem can be shown using the observations, by induction on the set of time instants, at which an operation is performed.

Theorem 1 *The implementations of operations* Find, Insert *and* Delete *are correct.* □

2.5 Amortized Message Complexity

In this subsection, we evaluate the amortized message complexity of the scheme.

Let $\mathcal{E}$ be a sequence of *events* (i.e., operations performed) with

- X Insert operations and
- Y Find or Delete operations.

Let N^- (resp. N^+) denote the total number of nodes deleted from (resp. added to) U during $\mathcal{E}$.

2.5.1 Inquiries Received by Nodes in the Circular List

The total number of messages received by or sent from nodes in the circular list during $\mathcal{E}$ is at most

$$3(X+Y) + 3N^- + 3(X+Y)$$

since each call of FindNode discards at most three messages (due to FindNode initiated by the anchor), and since, upon receiving an inquiry, a node either replies skip_me and then receives contract (and it is excluded from the circular list), or replies found and then receives unblock, remove_token, or place_token. Since $N^- \leq Y$, the total number of messages handled by those nodes during $\mathcal{E}$ is at most

$$6X + 6Y + 3Y \;=\; 6X + 9Y,$$

which is a constant per operation in an amortized sense.

2.5.2 Inquiries Received by Nodes Not in the Circular List

Next, let us count the total number of messages handled by nodes not in the circular list. Let T be the set of nodes excluded from the circular list. Note that T forms a forest. In what follows, a *primitive event* implies an event associated with an event in $\mathcal{E}$ which modifies the configuration of T; i.e., it either deletes a node from T

with cost zero, inserts a node to T as a root of existing trees with cost zero, or applies a path compression to T with a cost equals to the length of the compressed path. Note that the cost of a primitive event equals to the number of `inquire` messages received by nodes not in the circular list during the event in $\mathcal{E}$ associated with the primitive event.

In the following, we prove that the cost of a primitive event is at most $\log_2 |V|$ in the amortized sense. To obtain the bound, we apply the potential function method invented in [6]. Let the size $s(v)$ of a node v in T be the number of descendants of v including v itself. Let the potential of T be

$$\Phi(T) = \frac{1}{2}\sum_{v\in T}\log_2 s(v).$$

Define the *amortized cost* of a path compression over a path of k edges to be $k - \Phi(T) + \Phi(T')$, where T and T' are the forests before and after the compression, respectively. For any sequence of m events, we have

$$\sum_{i=1}^{m} a_i = \sum_{i=1}^{m}(t_i - \Phi_{i-1} + \Phi_i) = \sum_{i=1}^{m} t_i - \Phi_0 + \Phi_m,$$

where a_i, t_i, and Φ_i are the amortized cost of the ith primitive event, the actual cost of the ith primitive event, and the potential after the ith primitive event, respectively, and Φ_0 is the potential of the initial forest. Since $\Phi_0 = 0$ (recall that T_0 is an empty forest) and $\Phi_m \geq 0$, we have

$$\sum_{i=1}^{m} t_i \leq \sum_{i=1}^{m} a_i.$$

Now, let us bound the amortized cost a_i for each primitive event.

An insertion of a node into T increases the potential by at most $\log_2(|V|-1)$, and a deletion of a node from T following a path compression, does not increase the potential. Since none of those primitive events take actual cost, the amortized cost of those primitive events is at most $\log_2(|V|-1)$. On the other hand, by using a similar argument to [6] we can also claim that the amortized cost of a path compression is at most $\log_2(|V|-1)$. Since the number of inquiry messages per event in $\mathcal{E}$ associated with a primitive event is at most $\log_2(|V|-1)$ in the amortized sense, and since each of such inquiries is followed by two messages (i.e., `skip_me` and `contract`), we have the following theorems.

Theorem 2 *The number of messages handled by nodes not in the circular list is at most* $3\log_2(|V|-1)$ *per event in* $\mathcal{E}$ *in the amortized sense.* □

Theorem 3 *For any sequence of events with* X Insert *operations and* Y Find *or* Delete *operations, the amortized message complexity of each operation of the proposed scheme is at most*

$$(6 + 3\log_2(|V|-1))X + (9 + 3\log_2(|V|-1))Y.$$

□

3 Applications

3.1 Load Balancing

Suppose that at any time instant, each node in G is either lightly loaded, mediumly loaded or heavily loaded. For convenience, we associate the load of a node with the number of "tasks" in the ready-queue of the node. A task is dynamically created and removed on each node. A **load balancing problem** is the problem of migrating tasks of heavily loaded nodes to lightly loaded nodes as quick as possible [9].

The scheme proposed in Section 2 can be applied to the load balancing problem as follows. The basic idea is to let U be the set of lightly loaded nodes. When the load balancing algorithm is initiated, all nodes in V are assumed to be lightly loaded, i.e., $U = V$. If the load of a node v not in U becomes light, it inserts itself in U by performing Insert. On the other hand, when the load of a node $v \in U$ increases to medium, it deletes itself from U by performing Delete. If a node with heavy load wishes to migrate some tasks to a light one, it performs Find to find a light node.

Theimer and Lantz's load balancing algorithm, for example, essentially requires $O(|V|)$ messages to find out a lightly loaded node in the worst case, while ours requires $O(\log_2 |V|)$ messages (in the amortized sense).

3.2 Mutual Exclusion

We initiate the system such that $U = \emptyset$. If a node wishes to enter the critical section, it performs Insert and waits for it becoming the anchor. If a node v wishes to leave the critical section, it performs Delete. Since v is the anchor, Delete first sends the token to the next node

in C (who is waiting for its turn), and then deletes v from C. This mutual exclusion algorithm requires only $O(\log_2 |V|)$ messages per entry. Furthermore, this has the following advantage.

In many cases, the set of nodes which are interested in entering the critical section is a rather small dynamic subset U of the whole node set V. The above solution can be viewed as a token ring system among U, in which U may dynamically change. The token is circulated among the small group U.

3.3 FIFO Queue

A FIFO queue can be implemented, by modifying the proposed scheme as follows. We let the token represent the *head* of the circular list. We modify FindNode so that it looks for only the head. Operation Find then returns the head. Operation Insert first finds the head and inserts the caller as the predecessor of the head, i.e., from the tail of the circular list Operation Delete is exactly the same as in Section 2. It passes the token to the next node, and delete the previous head (the caller) from C.

4 Concluding Remarks

In this paper, we proposed a simple scheme for keeping a set U of nodes in a distributed manner, and for finding a node in U by using at most $9 + 3\log_2(|V| - 1)$ messages per operation in an amortized sense. The proposed scheme can be applied to many important problems, which includes a load balancing problem, the mutual exclusion problem, and a construction of distributed FIFO queue.

An important future problem is to apply the scheme to distributed systems in which the communication cost between two nodes is "not" unique; i.e., to consider the problem of finding a "closest" node in U in such systems. Another interesting problem is to apply the scheme to several practical problems such as load balancing problem described in Subsection 3.1, and evaluate the performance experimentally. Also implementing other processor structures such as a stack and a priority queue is an interesting open question.

References

[1] B. Awerbuch. Optimal distributed algorithms for minimal weight spanning tree, counting, leader election and related problems. In *Proc. 19th STOC*, pages 230–240. ACM, 1987.

[2] José M. Bernabéu-Aubán and Mustaque Ahamad. Applying a path-compression technique to obtain an efficient distributed mutual exclusion algorithm. In *Proc. 3rd WDAG (LNCS 392)*, pages 33–44, 1989.

[3] F. Chin and H. F. Ting. An almost linear time and $o(n \log n + e)$ messages distributed algorithm for minimum-weight spanning trees. In *Proc. 26th FOCS*, pages 257–266. IEEE, 1985.

[4] Robert Joseph Fowler. The complexity of using forwarding addresses for decentralized object finding. In *Proc. 5th PODC*, pages 108–120. ACM, 1986.

[5] R.G. Gallager, P.A. Humblet, and P.M. Spira. A distributed algorithm for minimum-weight spanning tree. In *ACM Transactions on Programming Languages and Systems 5*, 1 66–77, 1983.

[6] David Ginat, Daniel D. Sleator, and Robert E. Tarjan. A tight amortized bound for path traversal. *Information Processing Letters*, 31:3–5, April 1989.

[7] Gurdip Singh and Arthur J. Bernstein. A highly asynchronous minimum spanning tree protocol. *Distributed Computing*, 8:151–161, 1995.

[8] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[9] Marvin M. Theimer and Keith A. Lantz. Finding idle machines in a workstation-based distributed system. In *Proc. 8th ICDCS*, pages 112–122, IEEE, 1988.

[10] Michel Trehel and Mohamed Naimi. A distributed algorithm for mutual exclusion based on data structures and fault tolerance. In *Proc. 6th Annual Phoenix Conf. on Computers and Communications*, pages 35–39. IEEE, 1987.

[11] Tai-Kuo Woo. Huffman trees as a basis for a dynamic mutual exclusion algorithm for distributed systems. In *Proc. 12th IEEE Int. Conf. on Distr. Comp. Sys.*, pages 126–133. IEEE, 1992.