Dijkstra and Scholten's Algorithm - Simplified

Vijay K. Garg

TR-PDS-1998-012 November 1998

Parallel & Distributed Systems group
Department of Electrical & Computer Engineering
University of Texas at Austin
Austin, Texas 78712

Dijkstra and Scholten's Algorithm - Simplified

Vijay K. Garg*
Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712.

November 18, 1998

Abstract

Dijkstra and Scholten's algorithm for termination detection requires every node to maintain the sum of deficit of incoming edges and the sum of deficit of outgoing edges. I show that it is sufficient to maintain the sum of deficit of outgoing edges. The resulting algorithm is also simpler to understand and prove.

1 Introduction

Consider a computation on a distributed system which is started by a special process called environment. This process starts up the computation by sending messages to some of the nodes. Each node in the system is either passive or active. It is assumed that a passive node can become active only on receiving a message. Further a message can be sent by a node only if it is in the active state. From these properties it follows that if all nodes are passive in the system and there are no messages in transit then the computation has terminated. Our problem is to design a protocol by which the environment process can determine if the computation has terminated. We describe below an algorithm which is simpler than that of Dijkstra and Scholten [DS80].

We say that a process is in green state if it is passive and all of its outgoing channels are empty. How can a process determine whether its outgoing channel is empty ? This can be done if the receiver of the channel signals the sender of the channel the number of messages received along that channel. This can be achieved if the sender keeps a variable $D[i]$ (for deficit) for each of the outgoing channel i which records the number of messages sent minus the number of messages that have been acknowledged via signals. Thus, the sender can determine that the channel i is empty by checking whether $D[i] = 0$. Observe that $D[i] \geq 0$ is always true. Therefore, if O is the set of all outgoing channels, it follows that $\forall i \in O : D[i] = 0$ is equivalent to $\sum_{i \in O} D[i] = 0$. Thus, it is enough for a node to maintain just one variable D which represent the total deficit for the node.

It is clear that if all processes are in the green state then the computation has terminated. How do we check for this condition ? An equivalent question is how would the environment determine

*supported in part by the NSF Grants ECS-9414780, CCR-9520540, TRW faculty assistantship award, a General Motors Fellowship, and an IBM grant. http://maple.ece.utexas.edu

that all processes that were made active by it directly or indirectly have returned to the green state. Since checking for nodes that were made active indirectly is difficult, we shift the responsibility of checking the condition "have returned to the green state" to those nodes who are responsible for turning them to the active state in the first place. Thus, the responsibility can be viewed as a set of directed edges - an edge from node j to k signifies that P_j is responsible for determining whether P_k has turned green or not. We call P_j as the parent of the node P_k. Since each node that ever turned active has exactly one incoming edge (except for the environment which has none), and only these nodes have outgoing edges it follows that responsibility edges form a rooted spanning tree of all such nodes. By ensuring that a node reports to its parent only after getting reports from all its children, we get that a node P_j on receiving a report from P_k can assume that all nodes turned active by P_k directly or indirectly have returned to the green state. Thus, when the environment receives reports from all its children, the computation has terminated.

There is one point that needs further elaboration. The property that a node is green is not stable and hence a node, say P_k, that is green may become active once again on receiving a message. However, since a message can only be sent by an active process we know that some active process (which is already a part of the spanning tree) will be now responsible for the node P_k. Thus, the spanning tree changes with time but maintains the invariant that all active nodes are part of the tree.

The algorithm given above can be optimized by combining messages from the reporting process and the messages for detecting whether a node is green. To detect whether an outgoing channel is empty we assumed a mechanism by which the receiver tells the sender the number of messages it has received. One implementation could be based on control messages called signal. For every message received a node is eventually required to send a signal message to the sender. To avoid the use of report message, we require that a node not send the signal message for the message that made it active until it is ready to report. When it is ready to report the signal message is sent. With this constraint we get an additional property that a node will not turn green unless all its children in the spanning tree have reported. Thus a node is ready to report when it has turned green i.e. it is passive and $D = 0$.

The entire algorithm is given next.

```
var
   state: {passive, active} initially passive except for environment;
   D: integer initially 0;
   parent: process id initially null;

 on receiving a message from P.j
   if parent=null then
        parent = P.j
        state = active;
   else send signal to P.j;

 on receiving a signal
   D := D - 1;
```

```
 to send a message (enabled only if active)
   D := D + 1;

 on passive and (D=0) and (parent != null)
   send signal to parent;
   parent := null;
```

The environment detects termination when D = 0.

References

[DS80] E. W. Dijkstra and C. S Scholten. Termination detection for diffusing computations. *Information Processing Letters*, 11(4):1–4, August 1980.