Distributed Summation (for the record)

0 Introduction

In the first lecture, last spring, of the course on distributed systems I presented an algorithm for the computation of the sum of a distributed collection of numbers. My purpose was to make the students familiar with the notion of a distributed algorithm, and this problem I considered the simplest possible one serving that purpose. In addition, this problem and its solution occur as an ingredient in other, more complicated, protocols.

Here I wish to record how I discussed this problem during that first lecture. In the mean time it has become clear that further separations of concerns are possible; it is, for instance, possible to view the sending and receiving of messages as a (postponable) implementation issue and to formulate the solution as an "ordinary" parallel program first. This I will not do now, but I will pay some attention to it in the Epilogue. Also, I will present this algorithm with a degree of informality that matches its simplicity.

1 Distributed Summation

We consider a finite, connected, undirected graph the nodes of which are "machines" and the edges of which are (bidirectional) "channels" via which the machines can transmit "messages" to their neighbours. (Two nodes are neighbours if and only if they are connected by an edge.) Each node holds a (fixed) number and the problem is to construct an algorithm for the computation of the sum of these numbers; upon termination of the computation this sum is to be available in one, a priori designated, node called the *Root* of the graph.

More precisely, the number in each node must contribute to the calculated sum exactly once; this requirement can be split into two simpler requirements: each number must contribute to the calculated sum *at least once*, so as to avoid "omissions", and each number must contribute to the calculated sum *at most once*, so as to avoid "duplicates".

In addition, we are heading for a *fully distributed* solution; this means, that no node contains information about the graph as a whole; for example, even the size of the graph is not known to the individual nodes. The only information available to a node is the identities of its neighbours.

As for message transmission, we assume that every message sent along an

edge will arrive after a finite amount of time, but we do not impose any upperbound on the amount of time this takes. This is known as *asynchronous* communication. A consequence of the absence of an upperbound is, that a node does not obtain any information whatsoever from *not* receiving a message: then, either no message has been sent or a message has been sent but has not yet arrived. As we will see, this plays a role in the design.

2 Spanning Trees

A tree is a graph that is connected and that contains no cycles. A connected graph has, for every two nodes, *at least one* path connecting these nodes. A graph without cycles has, for every two nodes, *at most one* path connecting these nodes. As a result, a tree has, for every two nodes, *exactly one* path connecting these nodes.

For our given graph of machines, a spanning tree is a tree containing all nodes of the graph and whose edges are edges of the graph; so, a spanning tree is a subgraph of the given graph.

The requirements of no omissions and no duplicates can now be met by means of a spanning tree: for each node, its number will be sent –in a way still to be designed– from that node to the Root along the unique path in the spanning tree.

The algorithm to be presented here consists of two phases. In the first phase a spanning tree is constructed and in the second phase this tree is used to collect the numbers and to calculate their sum. These two phases can be executed (partly) in parallel: the first phase need not have terminated before the second one can start.

3 The First Phase

We use colours to represent the spanning tree. Initially, all nodes and all edges are white. The tree will consist of red nodes and red edges.

During the first phase dedicated messages will be sent from nodes to their neighbours; these messages carry no further content and we call such a message a “trigger”. (In the second phase we will introduce another type of message.)

The first phase is started by the Root, which does so by, first, becoming red and, second, sending a trigger to each of its neighbours. Furthermore, after having received at least one trigger, each white node will, first, designate one of its neighbours from which it has received a trigger as its (so-called) *father*

and, second, make itself and the edge towards its father red and, third, send a trigger to each of its neighbours *except* its father.

An invariant of this phase is that the red nodes and red edges form a tree. Hence, if the graph contains no more white nodes, the red nodes and edges form a spanning tree. As long as the graph contains a white node it also contains a white node with a red neighbour –here the connectivity of the graph is needed–, and such a white node will eventually receive a trigger –here we need that messages sent will eventually arrive–, and become red; so, as long as the graph contains white nodes the number of white nodes will decrease –here the finiteness of the graph is needed– and eventually become zero: the first phase terminates, and it does so in a state where all nodes are red.

The red nodes and edges form a spanning tree; in addition, the father designations represent, for each node in the tree, the unique path (in the tree) from that node to the Root; this will be used in the second phase.

4 The Second Phase

As we are only interested in the sum of the numbers in the graph, it is not efficient to let two (or more) numbers travel partly along the same path: such numbers may be replaced by their sum as soon as this is possible.

The father designations turn the spanning tree into a directed one, where all father designations are directed towards the Root; more precisely, the father of every node is closer to the Root than the node itself (except for the Root, which has no father). Also, every node can now be viewed as the root of a subtree: the subtree of node x is (recursively defined as) node x together with the subtrees of the *children* of x –those nodes that have designated x as their father–.

During the second phase dedicated messages will be sent from nodes to their neighbours; these messages carry a value and we call such a message a "response", to distinguish them from the triggers. (Because the two phases may proceed in parallel the two kinds of messages must be distinguishable.)

Each red node x waits until it has received a response from all of its children. By Induction Hypothesis, the value in such a response equals the sum of all numbers in the nodes of the subtree of the child from which that response has been received. (The basis of the induction is formed by the *leaves* of the tree, which are the nodes without children.) Having received all responses, node x calculates the sum of: its own number and all values received from its children. The value thus obtained is the sum of all numbers present in the subtree of x, and, finally, x sends this value in a response to

its father, except when x is the Root: if x is the Root, the Root has obtained the desired answer and the computation terminates.

The second phase starts whenever a node has become red without "generating" any children, that is, whenever a leaf emerges in the spanning tree. This may happen at a time when the graph, elsewhere, still contains white nodes: the two phases can be really concurrent. For each individual node, though, the algorithm is strictly sequential: first, the white node receives a trigger, then it becomes red and sends triggers, and only then it receives responses and sends a response.

5 A Minor Difficulty

Each node, except the Root, designates one of its neighbours as its father, but this information is not transmitted to that father. So, the identities of its children are not available to a node. How, then, can a node decide that it has received a response from all of its children? The answer is surprisingly simple.

Each node sends one trigger to each of its neighbours, except its father, and it sends one response to its father. So, each node sends exactly one message –either trigger or response– along each of its edges. As a result, each edge in the graph will conduct one message in either direction, and, hence, each node will also receive exactly one message along each of its edges. If the message received is a trigger, the neighbour sending it is not a child, and if the message received is a response, the sending neighbour is a child. From not receiving a response, a node cannot conclude that its neighbour is not a child, but from every non-child neighbour the node will receive a trigger.

Therefore, the second phase can be implemented thus: each red node waits until it has received as many messages as it has neighbours, and then it proceeds as defined before. Remember that the triggers carry no information: they only have to be counted.

This shows that, in such an asynchronous system, it really is necessary that a node sends triggers to each of its non-father neighbours, even if such a neighbour already has become red: these triggers are needed to identify the node as a non-child.

6 Epilogue

Messages serve to convey information about the state of a node to its neighbours; so, we might consider messages as a (particular) implementation of a

mechanism to inspect the state of a neighbour, and we might try to formulate the algorithm in terms of state inspections first. For our problem this is relatively easy.

In the first phase, a white node having at least one red neighbour designates one of its red neighbours as its father and, then, makes itself and the edge towards its father red. The triggers can now be viewed as messages conveying the information "the sender of this trigger has become red".

The second phase requires somewhat more care. Formulated in terms of states only, a red node calculates its value as the sum of: its own number and all values calculated by its children. The problem is that a node can calculate its value only *after* all its children already have calculated their values; the response messages implicitly provided the required synchronisation, but without these messages this synchronisation must be programmed explicitly. We can do so by introducing an additional colour in the states of the nodes, black, say, and by introducing as additional invariant that all black nodes have calculated their values. The algorithm for the second phase then becomes: a red node waits until all its children have become black, then it calculates its value and, finally, it becomes black itself. The computation now has terminated when the Root has become black. The responses can now be viewed as messages conveying both the information "the sender of this response has become black" and the value calculated by that sender.

* * *

We have discussed summation of a collection of numbers, but this algorithm can, of course, be applied to any datatype with a symmetric and associative binary operator. For example, instead of a number each node may hold a boolean and we may be interested in the conjunction of these booleans. If each boolean represents "local permission to perform a critical action" then this algorithm could be an ingredient of a fully distributed algorithm for mutual exclusion. If each boolean represents "local termination" then this algorithm could be used for termination detection (or phase synchronisation).

Eindhoven, 17 september 1998

Rob R. Hoogerwoord
department of mathematics and computing science
Eindhoven University of Technology
postbus 513
5600 MB Eindhoven