Synchronizing Asynchronous Bounded Delay Networks

CHING-TSUN CHOU, ISRAEL CIDON, INDER S. GOPAL, AND SHMUEL ZAKS

Abstract—**An efficient way to synchronize an asynchronous network with a bounded delay message delivery is presented. Two types of synchronization algorithms are presented. Both types require an initializing phase that costs $|E|$ messages (where $|E|$ is the number of links). The first requires an additional bit in every message and increases the time complexity by a factor of 2. The second does not require any additional bits but increases the time complexity by a factor of 3. We also explain how to overcome differences in nodal timer rates.**

I. Introduction

Several models for distributed communication networks have been described in the literature. Two of them have been extensively used for the development of distributed algorithms—the synchronous and the asynchronous models.

The synchronous model assumes a common clocking system for all nodes and a bounded message delivery delay. Essentially, time can be thought of as being partitioned into slots. Message transmissions always occur at a fixed position in a slot. If a node transmits a message during slot i it is guaranteed that the message will be received (and processed) by all neighbors by the start of slot $i+1$. Distributed algorithms that operate in phases or cycles fit naturally into this model. The synchronous nature of the message transmissions ensures that all the information of the previous cycle is available to a node before sending the messages of the next cycles.

The asynchronous model assumes no common clocking facility and typically assumes a finite but unbounded delay for message delivery. Consequently, many distributed algorithms designed for asynchronous networks are considerably less efficient in terms of time and message complexity than those designed for synchronous networks.

An interesting approach is given in [1] where the author implements a *synchronizer* in an asynchronous network in order to simulate the synchronous model and to execute a synchronous distributed algorithm in an asynchronous network. The synchronizer itself is a distributed algorithm which enables the nodes to define time slots and to detect when such slots start and end. The slots are guaranteed to have the same property as in the synchronous network, i.e., messages transmitted by a node during its slot i will be received (and processed) by all neighbors before the start of their slot $i+1$. Note that the slots in different nodes do not necessarily occur at the same instants in time. While synchronous algorithms can now operate in this network, an overhead must be paid for the operation of the synchronizer itself, resulting in increased complexity for the operation of the given synchronous algorithms.

In many practical communication systems the asynchronous model can be strengthened. While it is still true that most systems lack a common clocking mechanism, they do often guarantee message delivery within a fixed (and small) time bound. This is particularly true of the new generation of computer networks (for example, [2]), which are comprised of high speed fiber optic lines and in which the messages are routed through specialized high speed hardware rather than in general purpose processors. In addition, the nodes in these systems have highly accurate timers which are not synchronized with each other but operate at equal rates.

In this paper we show that in such a model—asynchronous with bounded delay (ABD)—the implementation of a synchronizer is considerably simpler than in the truly asynchronous model. Consequently, the complexity added to the operation of synchronous algorithms in an ABD network is considerably lower than in a truly asynchronous network. Two synchronization algorithms are presented in this paper. In both algorithms there is an initialization phase that costs $|E|$ messages (where $|E|$ is the number of links). The first algorithm requires an additional bit in every message and increases the time complexity by a factor of 2. The second does not require any additional bits but increases the time complexity by a factor of 3.

Thus, either synchronizer preserves the order of complexity of the synchronous algorithm while only increasing the constant factor. Consequently, for many problems such as breadth first search (BFS) [2] and finding maximum flow, an ABD network can be substantially better than a truly asynchronous network.

While the timers in the nodes are usually accurate, it is useful to have a synchronizer that is robust enough to work with some timer inaccuracies. We show that the synchronizers presented in the paper have this property at the cost of some overhead in time complexity.

II. Preliminaries

A communication network is represented by a graph (V, E) where V is the set of processing nodes and $E \subset V \times V$ the set of communication links. All links are bidirectional and a message transmitted by a node over a link arrives at the other end with an arbitrary delay less than one unit of time. Each node has a timer which can be reset to zero at any arbitrary time. For now, we assume that all timers are accurate and there is no mutual drift between timers in different nodes. In the description of the algorithms we shall often refer to a global time. While all timers proceed at the same rate as global time, the actual value of this time is not available to the individual nodes and is introduced only for descriptive clarity.

In order to motivate the need for synchronization consider a distributed implementation of breadth first search (DBFS). The objective of a DBFS algorithm is to compute the minimum distances (measured in number of hops) from a distinguished node s to all other nodes in the network. Upon termination of the DBFS algorithm every node has to know its minimum distance from s and the link on the corresponding shortest path to s. The synchronous implementation of DBFS is to broadcast from s a message carrying a distance counter, whose value determines the distance from s. Node s first sends its neighbors the message with distance counter set to zero. Each node, when it receives the first copy of the message, will set its distance to be one larger than the value of the counter, increment the counter, and send the message to all of its neighbors. Subsequent copies of the message are discarded. The algorithm takes $O(|E|)$ messages and $O(D)$ time where E is the set of links in the network and D is the maximal distance from any node to s. The algorithm works because the synchronous nature of the transmission ensures that the first message is always received over the shortest path from s.

Now consider an implementation of DBFS on an ABD network. Assume the algorithm starts at global time 0. Assume further that every node in the network receives a signal (say, from an obliging "genie") informing it of the start of the algorithm. The nodes now

Paper approved by the Editor for Network Protocols of the IEEE Communications Society. Manuscript received February 16, 1987; revised July 5, 1988. This paper was presented at the Second International Workshop on Distributed Algorithms, Amsterdam, The Netherlands, July 1987.

C.-T. Chou is with the Department of Computer Science, University of California, Los Angeles, CA 90024.

I. Cidon and I. S. Gopal are with IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598.

S. Zaks is with the Department of Computer Science, Technion—Israel Institute of Technology, Haifa 32000, Israel.

IEEE Log Number 8933276.

0090-6778/90/0200-0144$01.00

can impose their own slots using their individual timers and thereby simulate a synchronous system. In particular, the nodes will set their timers to zero at the start of the algorithm and define the mth slot, $m \geq 0$, to begin at time m and to end at time $m+1$. The DBFS algorithm will proceed in exactly the same way as before except that a processor receiving a message with distance counter d waits until the end of the dth slot whereupon it sets its distance to $d+1$, increments the counter and forwards the message. Thus, s will transmit its message with distance counter 0 at global time 0, processors one hop away will receive the message before global time 1 and at that time will set their distance to 1, increment the distance counter and forward the message, and so on. Actually, no counter is needed, since the operation of the algorithm guarantees that *every* message received within the dth slot will have distance counter d. This solution clearly uses $O(|E|)$ messages and $O(D)$ time.

Let us now do away with the start signal. First, we attempt to get rid of the signal by using the distance message itself to inform the nodes of the start of the algorithm. Upon receipt of the distance message, a node will set its timer to zero and, as in the previous case, forward a message with counter d after the end of the dth slot. Recall that we are not lower bounding message delays so a node cannot start its timer at a value greater than 0 without risking the possibility that it will forward the message too early. The algorithm operates correctly (the proof is left to the reader). However, it is easily seen that the time complexity increases to $O(D^2)$, even though the message complexity will remain $O(|E|)$. We observe that the increase in time complexity is because the distance message is delayed by each node by d slots, $d = 1, 2, \cdots, D$. Thus, we will attempt to simulate the start signal by introducing an initialization phase, which consists of node s flooding the network with an initialization message before commencing the DBFS algorithm. The receipt of the initialization message causes the nodes to set their timers to zero, If the initialization message took zero time to propagate through the network, we have succeeded in replicating the start signal. Unfortunately, we can only bound the message delay, thereby ensuring that the starting points of two neighboring nodes differ by less than 1. However, this small difference in initial synchronization causes the algorithm to operate incorrectly. In particular, a processor receiving a message carrying a counter d, cannot just wait until the end of the dth time slot and then forward the (incremented) counter, since a message with a lower counter can reach it later. In order to ensure correct operation, the algorithm must be modified to send more messages upon receipt of a new message with the lower counter. This increases the message complexity even though the time will remain $O(D)$.

Thus, we see that obvious ways of simulating the start signal do not lead to DBFS algorithms that are efficient in both time and messages. A consequence of our synchronization algorithm to be presented in the next section is a DBFS algorithm that achieves both $O(|E|)$ message complexity and $O(D)$ time complexity.

III. Synchronization Algorithms

A. The General Structure

The synchronization algorithms presented in this section have the following structure. As in the algorithm presented at the end of the previous section, they begin with an initialization phase which consists of a flooding of initialization messages. The initiating node, node s sends an initialization message (INIT) to all its neighbors. Each node upon receiving the first INIT message forwards this INIT message to all neighboring nodes except the one from which the INIT message was received. As before, each node resets its timer to zero and starts counting the time. The difference is that the slots will span α units of time (the value of α is to be determined) rather than a single unit of time as before. Thus, the mth slot will begin at time $m\alpha$ and end at time $(m+1)\alpha$. We shall refer to a message sent by a node during slot m as a message of the mth cycle.

In this paper we present two types of synchronization protocols, the *send on start* (SOS) and the *send after delay* (SAD). They differ by the specific point of time in the slot where the nodes sends their messages of the current cycle, the duration of the time slot (α is 2 in the SOS and 3 in the SAD), and the fact that an additional bit must be appended to in each message in the SOS (but not in the SAD).

B. The Initialization Phase

Both the SOS and the SAD synchronizer use the same initialization phase. We give a description of this phase below.

```
On receiving START from the outside world or INIT from x do
  If FLAG = 0 then do
    FLAG ← 1;
    start timer from 0;
    send INIT to all neighbors except x
  end;
end;
```

Note, that *FLAG* can be reset after the reception of INIT from all neighbors.

Let t_i be the global time at which node i resets its timer. The initialization phase has the property that for any two neighboring nodes j and k, the following inequality holds:

$$t_k - 1 < t_j < t_k + 1. \tag{1}$$

This follows from the fact that the message delay is bounded by 1.

C. The Send on Start Synchronizer

In the send on start (SOS) synchronizer, each node sends its messages at the beginning of a time slot, i.e., messages of cycle $m \geq 0$ are sent at local time $m\alpha$. Assume that j and k are neighbors and that k receives j's message of cycle m at global time $\tau(m)_k$. The property of the synchronizer requires that node k receive all messages of cycle m *before* the start of its $m+1$th slot. This implies the following condition:

$$\tau(m)_k < t_k + (m+1)\alpha. \tag{2}$$

However, we know from (1) and the bound on the delay that

$$\tau(m)_k < t_j + m\alpha + 1 < t_k + m\alpha + 2 = t_k + (m+1)\alpha + (2-\alpha). \tag{3}$$

If $\alpha \geq 2$, inequality (2) results from inequality (3), and we are guaranteed that the synchronizer performs correctly. From (1) we find that the message arrival time also satisfies the following lower bound:

$$\tau(m)_k \geq t_j + m\alpha > t_k + m\alpha - 1 = t_k + (m-1)\alpha + (\alpha - 1). \tag{4}$$

For $\alpha \geq 2$, this implies that messages for cycle m will be received by all neighbors *after* the start of slot $m-1$ as well before the start of slot $m+1$. This implies that during slot m a node may receive messages from cycles m and $m+1$ but not from other cycles. Therefore, a node must distinguish only between two consecutive cycles. Stamping the messages with the parity bit of the cycle number suffices for this purpose. Thus, we arrive at the following algorithm.

The SOS Algorithm

```
On TIMER = mα do
  send messages of cycle m with STAMP = (m) mod 2;
end;
On receiving a message stamped with STAMP do
  If mα ≤ TIMER < (m+1)α then do
    If (m) mod 2 = STAMP then consider the received message
for cycle m;
  Else consider the received message for cycle m+1;
  end;
end;
```

D. The Send After Delay Synchronizer

The send after delay (SAD) synchronizer removes the need for the binary stamping of the messages by ensuring that all messages

of cycle m arrive only in slot m. This is achieved at the cost of increasing the value of α by one. In addition, instead of sending the messages of cycle m immediately at the start of slot m, the nodes wait for γ units of time.

We require that all messages sent by node j in cycle m will arrive at its neighbor k in cycle m. This is equivalent to the following condition:

$$t_k + m\alpha < \tau(m)_k < t_k + (m+1)\alpha. \tag{5}$$

We know from the delay bound and (1) that the following two inequalities hold:

$$\tau(m)_k < t_j + m\alpha + 1 + \gamma < t_k + m\alpha + 2 + \gamma = t_k + (m+1)\alpha + (2-\alpha) + \gamma \tag{6}$$

$$\tau(m)_k \geq t_j + m\alpha + \gamma > t_k + m\alpha - 1 + \gamma = t_k + (m-1)\alpha + (\alpha - 1). \tag{7}$$

By choosing $\gamma \geq 1$ and $\alpha \geq \gamma + 2$ it is clear that (6) and (7) imply (5). This means that the minimum values for γ and α are 1 and 3, respectively.

The SAD Algorithm

```
On TIMER = mα + γ do
   send messages of cycle m
end;
On receiving a message do
   If mα ≤ TIMER < (m + 1)α then
      consider the received message for cycle m;
end;
```

We end this section with the following remarks.

1) For the BFS example there is an equivalence between the cycle number in which the distance message is sent and the node distance from the source.

2) If the synchronous algorithm performed does not require FIFO ordering of message reception then neither does our synchronizer. In a non-FIFO environment some messages may arrive before the initialization phase is started. In such a case one of the following options may be taken: 1) consider all those messages received for cycle zero, or 2) consider the first message as a START for the initialization phase and treat this message as if it were received at local time 0.

IV. Handling Inaccurate Timers

As mentioned before, the timers are assumed to be highly accurate and drift between them is typically very small. However, it is useful to have an algorithm that is robust enough to cope with minor rate differences between the various timers in the network. Assume that the fastest timer will count one unit of time after at least $1-\epsilon$ units, while the slowest one will do it after at most $1+\epsilon$. This difference in the timer rates may cause, after some time, a mismatch between the cycles at neighboring nodes. We will compute the smallest possible cycle number when this mismatch may occur and then describe how the nodes resynchronize the timers again.

We assume that the SAD synchronization algorithm is executed; similar results can be derived for the SOS. If we follow the inequalities in (5), (6), and (7) and assume a fast clock in one node and a slow clock in the other, we find that the network will not suffer from any mismatch as long as for every two neighbors j and k the following inequalities hold:

$$t_j + m\alpha(1+\epsilon) + 1 + \gamma(1+\epsilon) < t_k + (m+1)\alpha(1-\epsilon) \tag{8}$$

$$t_j + m\alpha(1-\epsilon) + \gamma(1-\epsilon) > t_k + m\alpha(1+\epsilon). \tag{9}$$

Using (1), for (8) and (9) to be satisfied, it suffices to have the following conditions:

$$2m\alpha\epsilon \leq \alpha(1-\epsilon) - 2 - \gamma(1+\epsilon) \tag{10}$$

$$2m\alpha\epsilon \leq \gamma(1-\epsilon) - 1. \tag{11}$$

In order to maximize m while satisfying both (10) and (11), it is easy to show that γ should be chosen to be

$$\gamma = \frac{\alpha(1-\epsilon) - 1}{2}. \tag{12}$$

Which results that there is no mismatch problem as long as

$$m \leq \frac{\alpha(1-\epsilon)^2 - 3 + \epsilon}{4\alpha\epsilon}. \tag{13}$$

The choice of α determines the value of the bound. To get a feeling of the actual values involved let M be the maximal integer which satisfies (13). If the timers are accurate to within 1/10 of a second in a day this gives a value of $\epsilon = 1/864\,000$. In order for M to be reasonably large, it is necessary to increase α beyond the value of 3 obtained without timer drift. For example, a value of M close to 140 000 can be obtained by setting α to be 8.

When the cycle number exceeds M, there will be a mismatch problem and the timers need to be resynchronized. We now describe the resynchronization procedure. Assume that ϵ is a globally known number (given by the specifications of the timers). The goal of the resynchronization is to bring the network back to the point where the time difference between the reference time at each neighbor pair is less than one unit of time. The resynchronization is identical to the initialization phase, in which a flood of INIT messages brings all timers to a new reference time. The question is when and where this phase may start.

A node will proceed as normal until cycle M. It then will wait till the resynchronization process has completed before proceeding any further. The first cycle after the resynchronization has been completed will be considered to be cycle $M+1$. The resynchronization may be started asynchronously by each node that reaches cycle M after it is sure that all other nodes have reached cycle M as well. As mentioned, the resynchronization takes the form of a node broadcasting an INIT message exactly as in the initialization phase. We must be sure that all nodes in the network (not just adjacent nodes) have completed cycle M, otherwise the resynchronization process may prevent some nodes from completing a lower numbered cycle. In order to determine when all other nodes have completed cycle M two approaches are possible. One is to use a distributed algorithm to get explicit notification from every node that it has completed cycle M. For example, we can use a network wide search such as a propagation of information with feedback (PIF) [4]. A node that reaches cycle M will start a PIF and each other node will send the feedback only when it has also completed cycle M.

Another approach is to use the bounded message delay property of our model. We know that at cycle M (which is still without any mismatch) time slots of the same cycle at adjacent nodes still overlap. Therefore, a node can be sure that all other nodes have completed cycle M after an additional delay of $|V|\alpha(1+\epsilon)/(1-\epsilon)$ [the additional factor is because a node does not know whether its timer is fast or slow]. Thus, a node completes cycle M, waits this additional delay and initiates a new initialization phase. The time of receiving or sending the first INIT message is defined as the beginning of cycle $M+1$ and the messages of this cycle will be sent γ units of time after that. No messages are sent for detecting the termination of cycle M in the network. If knowledge of the network topology exists, then $|V|$ may be replaced by D which is the maximal distance between this node and all other nodes in the network.

V. Discussion

The algorithms presented in this paper represent simple and easy to implement synchronizers for asynchronous bounded delay networks. Both algorithms preserve the order of complexity of synchronous algorithms while only increasing the constant factor. The advent of high speed networks with specialized switching hardware makes the

bounded delay assumption quite realistic and increases the possibility that such synchronizers will be implemented in real networks.

References

[1] B. Awerbuch, "Complexity of network synchronization," *J. ACM*, vol. 32, no. 4, pp. 804–823, Oct. 1985.

[2] I. Cidon and I. S. Gopal, "PARIS: An approach to integrated high-speed private networks," *Int. J. Digital Analog Cabled Syst.*, vol. 1, no. 2, pp. 77–86, April–June 1988.

[3] S. Even, *Graph Algorithms*. Potomac, MD: Computer Science, 1979, pp. 12–18.

[4] A. Segall "Distributed network protocols," *IEEE Trans. Inform. Theory*, vol. IT-29, pp. 23–35, Jan. 1983.