Principles of Distributed Computing
Roger Wattenhofer

Chapter 7

Shared Variables

Section 7.1: Introduction

(Programmers do not like message passing. They like "global" variables; a.k.a. shared memory systems. Shared memory is an amazing and well-studied area within distributed computing, that does not get the share it deserves in this course.)

Definition 7.1 [Shared Memory]: A shared memory system is an asynchronous system that consists of processors that globally share variables. A processor can atomically access a shared variable through a set of predefined operations.

Remarks:

- Processors can also have local variables.
- If the processors operate in synchrony there is a model called PRAM (parallel random access machine).
- Typical operations are read, write, or read-modify-write (read and write in one atomic step – the written value may depend on the read).
- Many of the results in message passing have an equivalent in shared memory (e.g. consensus)
- How can we simulate a shared memory system (or even a single global variable) with a message passing system? See below.

Algorithm 7.2 [Static Location]: The global variable is stored at a node r, the root of a spanning tree in the message passing graph (that is, each node knows its parent in the spanning tree). If a node u initiates a read-modify-write operation it sends a request up the tree; the request is processed by the root r (atomically), and the reply/result is sent down the tree along the same path to the initiating node u. This terminates the operation.

Remarks:

- This works. Instead of a spanning tree, one can use routing.
- But it is not very efficient. Assume that the variable is accessed by a single node v repeatedly. Then we get a high message/time complexity. Instead v could store the variable locally, or at least cache it. But then, in case another node w accesses the variable, we might run into consistency problems.
- Alternative idea: The accessing node should become the new master of the variable. The variable is then like a mobile object. There exist several variants of this idea. The simplest version is a home-based solution like in Mobile IP.

Protocol 7.3 [Home-Based]: A variable has a home base (a node) that is known to every node. If a node accesses the variable, it acquires a lock at the home base, receiving the variable. Future requests are then routed through the home base.

Remark:

- Home-based solutions suffer from the triangular routing problem. If two close-by node access the variable on a rotating basis, all the traffic is routed through the potentially far away home-base.

Section 7.2: Arrow and Friends

Algorithm 7.4 [Arrow]: As in Algorithm 7.2 we are given a rooted spanning tree. Each node identifies one of its neighbor nodes as parent; the root is its own parent node. When a node u wants to acquire exclusive access to the global variable (or object), u sends a "find by u" message to the parent node and sets parent := u. A node w receiving "find by u" message by a node v does the following: a) if w has a parent other than w, w sends "find by u" to its parent and (atomically) sets parent := v. b) if parent = w, w sets parent := v, and sends the variable directly to u (once it is done with its operation).

Remarks:

- When we draw the parent pointers as arrows, in a quiescent moment (where no "find" is in motion) the arrows all point towards the root holding the variable.
- However, what is really great is that the Arrow algorithm also works in a concurrent setting.

Theorem 7.5 [Analysis]: In an asynchronous, steady-state, and concurrent setting, a "find" operation terminates with message and time complexity D, where D is the diameter of the spanning tree.

Lemma 7.6: An edge (u,v) is in one of four states

1) Pointer from u to v (no message on the edge, no pointer from v to u)
2) Message on the move from u to v (no pointers)
3) Pointer from v to u (no message on the edge, no pointer from u to v)
4) Message on the move from v to u (no pointers)

Proof: Initially the system is in state 1. With a message arrival at u the edge goes to stage 2, when the message is received at v we are in 3 – a new message at v then brings the edge back to first 4 and then 1.

Proof Theorem 7.5: Since the "find" message will only travel on a static tree, it suffices to show that it will not traverse an edge twice. Suppose for the sake of contradiction that there is a first "find" message f that traverses an edge e for the second time. The first time f traversed e from node u to v, the second time since we are on a tree in the other direction from v to u. The message f must re-visit the edge e before visiting any other edges because e is the first edge to be traversed twice. Right before f reaches v, the edge e is in state 2 (f is on the move) and in state 3 (it will immediately return with the pointer from v to u). This is a contradiction to Lemma 7.6.

Remarks:

- Finding a good tree is an interesting problem. We like to have a tree with low stretch, low diameter, low degree, etc.
- It seems that the Arrow algorithm works especially well when lots of "find" operations are initiated concurrently. Most of will find a "close-by" node, thus having low message/time complexity. For the sake of simplicity we analyze a synchronous system.

Theorem 7.7 [Concurrent Analysis]: Let the system be synchronous. Initially, the system is in a quiescent state. At time 0, a set S of nodes initiates a "find" operation. The message complexity of all "find" operations is O(log |S| m*), where m* is the message complexity of an optimal (with global knowledge) algorithm on the tree.

Proof (Sketch): Let d be the minimum distance of any node in S to the root. There will be a node u_1 at distance d from the root that reaches the root in d time steps, turning all the arrows on the path to the root towards u_1. A node u_2 that finds (is queued behind) u_1 cannot distinguish the system from a system where there was no request u_1, and instead the root was located at u_1. The message cost of u_2 is consequentially the distance between u_1 and u_2 on the spanning tree. By induction the total message complexity is exactly as if a collector starts at the root and then "greedily" collects tokens located at the nodes in S (greedily in the sense that the collector always goes towards the closest token). Greedy collecting the tokens is not a good strategy in general because it will traverse the same edge more than twice in the worst case. An asymptotically optimal algorithm can also be translated into a depth-first-search collecting paradigm, traversing each edge at most twice. In another area of computer science, we'd call the Arrow algorithm a nearest-neighbor TSP heuristic (without returning to the start/root though), and the optimal algorithm TSP-optimal. It was shown that nearest-neighbor has a logarithmic overhead, which concludes the proof.

Remarks:

- An average request set S on a not-to-bad tree gives usually a much better bound. However, there is an almost tight (log/loglog) worst-case example.
- It was recently shown that Arrow can do as good in a dynamic setting (where nodes are allowed to initiate requests at any time). In particular the message complexity of the dynamic analysis can be shown to have a log D overhead only, where D is the diameter of the spanning tree (for logarithmic trees, it is loglog n though).
- What if the spanning tree is a star? Then with Theorem 7.5 each find will terminate in 2 steps! Since also an optimal algorithm has message cost 1, the algorithm is 2-competitive…? Yes, but because of its high degree the star centre experiences contention… There is no contention-based analysis yet.
- Thought experiment: Assume a balanced binary spanning tree – by Theorem 7.5 the message complexity per operation is log n (also in a dynamic setting). But what about contention?!?
- There are better and worse choices for the spanning tree. The stretch of an edge (u,v) is defined as distance between u and v in a spanning tree. The maximum stretch of a spanning tree is the maximum stretch over all edges. It was recently shown how to construct spanning trees that are O(log n)-stretch-competitive.
- What if most nodes just want to read the variable? Then it does not make sense to acquire the lock. Instead we use caching.

Algorithm 7.8 [Read/Write Caching]:

- Nodes can either read or write the global variable. For simplicity we first assume that reads or writes do not overlap in time (access to the variable is sequential).
- Nodes store three items: a parent pointer pointing to one of the neighbors (as with Arrow), and a cache bit for each edge, plus (potentially) a copy of the variable.
- Initially the variable is stored at a single variable u; all the parent pointers point towards u, all the cache bits are false.

- When initiating a read, a message follows the arrows (this time: without inverting them!) until it reaches a cached version of the variable. Then a copy of the variable is cached along the path back to the initiating node, and the cache bits on the visited edges are set to true.
- A write at u writes the new value locally (at node u), then searches (follow the parent pointers and reverse them towards u) a first node with a copy. Delete the copy and follow (in parallel, by flooding) all edge that have the cache flag set. Point the parent pointer towards u, and remove the cache flags.

Theorem 7.9 [Analysis]: The algorithm above is correct. More surprisingly the message complexity is 3-competitive (at most a factor 3 off the best possible).

Proof: Since the accesses do not overlap by definition, it suffices to show that between two writes we are 3-competitive. The sequence of accessing nodes is $w_0, r_1, r_2, \ldots r_k, w_1$. After w_0 the variable is stored at w_0 and not cached anywhere else. All reads cost twice the smallest subtree T spanning the write and all the reads since each read only goes to the first copy. The write costs T plus the path P from w_1 to T. Since any data management scheme must use an edge in T and P at least once, and our algorithm uses edges in T at most three times (and in P at most once) the theorem follows.

Remarks:
- Concurrent reads are not a problem, also multiple concurrent reads and one write work just fine.
- What about concurrent writes? To achieve consistency writes need to invalidate the caches before writing their value. It is claimed that the strategy then becomes 4-competitive.
- Is the algorithm also time competitive? Well, not really: The optimal algorithm that we compare to is usually offline. This means it knows the whole access sequence in advance. It can then cache the variable before the request even appears!
- Algorithms on trees are often simpler, but have the disadvantage that they introduce the extra stretch factor. In a ring, for example, any tree has stretch n-1; so there is always a bad request pattern.
- In the following we study algorithms that do not restrict to a tree. Of particular interest is the special case of a complete graph. ("Peer-to-Peer" if you wish.)

Section 7.3: Ivy and Friends

Algorithm 7.10 [Pointer Forwarding]: Initially the variable is stored at a root r, and there is a spanning tree pointing towards r (as earlier each node has a parent pointer pointing towards the master of the lock/variable). A node u can acquire the lock of the variable by following the parent pointers until they reach the root. The node u will then become the new root, and the old root will redirect its parent pointer to the new root u.

Remarks:
- If the graph is not complete, routing can be used to find the root.
- Assume that the nodes line up in a linked list. If we always choose the first node of the linked list to acquire the lock, we have message/time complexity n. The new topology is again a linearly linked list. Pointer forwarding is therefore bad in a worst-case.

- If edges are not FIFO, it can even happen that the number of steps is unbounded for a node having bad luck. An algorithm with such a property is named "not fair," or, "not wait-free." (Example: Initially we have the list 4→3→2→1; 4 starts a find; when the message of 4 passes 3, 3 itself starts a find. The message of 3 may arrive at 2 and then 1 earlier, thus the new end of the list is 2→1→3; once the message of 4 passes 2, the game re-starts.)
- There seems to be a natural improvement.

Algorithm 7.11 [Ivy] As pointer forwarding. However, on the path to the root, a message redirects all visited nodes to itself.

Remark:

- Also with this algorithm we might have a bad linked list situation. However, if the start of the list acquires the lock, the linked list turns into a star. As the following theorem will show the search paths are not long on average. Since paths sometimes can be bad we will need amortized analysis.

Theorem 7.12 [Analysis] On average, acquiring a lock has log n steps, where n is the number of processors

Proof: We simplify the proof by assuming that accesses are sequential. We use a potential function argument Let s(u) be the size of the subtree rooted at node u (the number of nodes in the subtree including u itself). The potential of the whole tree is $\Sigma_{\text{all nodes u}} \log(s(u))/2$. A simple calculation (Ginat, Sleator, Tarjan) reveals that the amortized cost of each operation is at most log n (math done on blackboard only).

Remarks:

- Systems guys (the algorithm is called Ivy because it was used in a system with the same name) have some fancy heuristics to improve performance even more: For example, the root every now and then broadcasts its name such that paths will be shortened.
- What about concurrent requests? It works with the same argument as in Arrow. Also for Ivy an argument including congestion is missing (and more pressing, since the dynamic topology of a tree cannot be chosen to have low degree and thus low congestion as in Arrow)
- Sometimes the type of accesses allows that several accesses can be combined into one to reduce congestion higher up the tree. Let the tree in Algorithm 7.2 be a balanced binary tree. If the access to the shared variable is for example "add value x to the shared variable," two or more accesses that accidentally meet at a node can be combined into one. Clearly accidental meeting is rare in an asynchronous model. We can use synchronizers to help meeting a little bit. More on that in a later Chapter.