Chapter 7

Shared Objects

Assume that there is a common resource (e.g., a common variable or data structure), which different nodes in a network need to access from time to time. If the nodes are allowed to change the common object when accessing it, we need to guarantee that no two nodes have access to the object at the same time. In order to achieve this mutual exclusion, we need protocols that allow the nodes of a network to store and manage access to such a shared object.

7.1 Centralized Solutions

A simple and obvious solution is to store the shared object at a central location (see Algorithm 7.1).

Algorithm 7.1 Shared Object: Centralized Solution

Initialization: Shared object stored at root node \( r \) of a spanning tree of the network graph (i.e., each node knows its parent in the spanning tree).

Accessing Object: (by node \( v \))

1. \( v \) sends request up the tree
2. request processed by root \( r \) (atomically)
3. result sent down the tree to node \( v \)

Remarks:

- Instead of a spanning tree, one can use routing.
- Algorithm 7.1 works, but it is not very efficient. Assume that the object is accessed by a single node \( v \) repeatedly. Then we get a high message/time complexity. Instead \( v \) could store the object, or at least cache it. But then, in case another node \( w \) accesses the object, we might run into consistency problems.
- Alternative idea: The accessing node should become the new master of the object. The shared object then becomes mobile. There exist several variants of this idea. The simplest version is a home-based solution like in Mobile IP (see Algorithm 7.2).

7.2 Arrow and Friends

We will now look at a protocol (called the Arrow algorithm) that always moves the shared object to the node currently accessing it without creating the triangular routing problem of home-based solutions. The protocol runs on a precomputed spanning tree. Assume that the spanning tree is rooted at the current position of the shared object. When a node \( u \) wants to access the shared object, it sends out a find request towards the current position of the object. While searching for the object, the edges of the spanning tree are redirected such that in the end, the spanning tree is rooted at \( u \) (i.e., the new holder of the object). The details of the algorithm are given by Algorithm 7.3. For simplicity, we assume that a node \( u \) only starts a find request if \( u \) is not currently the holder of the shared object and if \( u \) has finished all previous find requests (i.e., it is not currently waiting to receive the object).

Algorithm 7.2 Shared Object: Home-Based Solution

Initialization: An object has a home base (a node) that is known to every node. All requests (accesses to the shared object) are routed through the home base.

Accessing Object: (by node \( v \))

1. \( v \) acquires a lock at the home base, receives object.

Remarks:

- Home-based solutions suffer from the triangular routing problem. If two close-by nodes take turns to access the object, all the traffic is routed through the potentially far away home-base.

Theorem 7.4. (Arrow, Analysis) In an asynchronous and concurrent setting, a "find" operation terminates with message and time complexity \( D \), where \( D \) is the diameter of the spanning tree.

Remarks:

- The parent pointers in Algorithm 7.3 are only needed for the find operation. Sending the variable to \( u \) in line 13 or to \( w \)'s successor in line 23 is done using routing (on the spanning tree or on the underlying network).
- When we draw the parent pointers as arrows, in a quiescent moment (where no "find" is in motion), the arrows all point towards the node currently holding the variable (i.e., the tree is rooted at the node holding the variable).
- What is really great about the Arrow algorithm is that it works in a completely asynchronous and concurrent setting (i.e., there can be many find requests at the same time).
Algorithm 7.3 Shared Object: Arrow Algorithm

Initialization: As for Algorithm 7.1, we are given a rooted spanning tree. Each node has a pointer to its parent, the root is its own parent. The variable is initially stored at r. For all nodes \( v \), \( v.\text{successor} := \text{null}, v.\text{wait} := \text{false} \).

Start Find Request at Node \( u \):
1: do atomically
2: \( u \) sends “find by \( u \)” message to parent node
3: \( u.\text{parent} := u \)
4: \( u.\text{wait} := \text{true} \)
5: end do

Upon \( w \) Receiving “Find by \( u \)” Message from Node \( v \):
6: do atomically
7: if \( w.\text{parent} \neq w \) then
8: \( w \) sends “find by \( u \)” message to parent
9: \( w.\text{parent} := v \)
10: else
11: \( w.\text{parent} := u \)
12: if not \( w.\text{wait} \) then
13: send variable to \( u \)  // \( w \) holds var. but does not need it any more
14: else
15: \( w.\text{successor} := u \)  // \( w \) will send variable to \( u \) asap.
16: end if
17: end if
18: end do

Upon \( w \) Receiving Shared Object:
19: perform operation on shared object
20: do atomically
21: \( w.\text{wait} := \text{false} \)
22: if \( w.\text{successor} \neq \text{null} \) then
23: send variable to \( w.\text{successor} \)
24: \( w.\text{successor} := \text{null} \)
25: end if
26: end do

Before proving Theorem 7.4, we prove the following lemma.

Lemma 7.5. An edge \( \{u, v\} \) of the spanning tree 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 more from \( u \) to \( v \) (no pointer along the edge)
3) Pointer from \( v \) to \( u \) (no message on the edge, no pointer from \( u \) to \( v \))
4) Message on the more from \( v \) to \( u \) (no pointer along the edge)

Proof. W.l.o.g., assume that initially the edge \( \{u, v\} \) is in state 1. With a message arrival at \( u \) (or if \( u \) starts a “find by \( u \)” request, the edge goes to state 2). When the message is received at \( v \), \( v \) directs its pointer to \( u \) and we are therefore in state 3. A new message at \( v \) (or a new request initiated by \( v \) then brings the edge back to state 1.

Proof of Theorem 7.4. 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 \( e = \{u, v\} \) for the second time and assume that \( e \) is the first edge that is traversed twice by \( f \). The first time, \( f \) traverses \( e \). Assume that \( e \) is first traversed from \( u \) to \( v \). Since we are on a tree, the second time, \( e \) must be traversed from \( v \) to \( u \). Because \( e \) is the first edge to be traversed twice, \( f \) must re-visit \( e \) before visiting any other edges. 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 7.5.

Remarks:
- Finding a good tree is an interesting problem. We would 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 them will find a “close-by” node, thus having low message/time complexity. For the sake of simplicity we analyze a synchronous system.

Theorem 7.6. (Arrow, 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 \( O(\log |S|) \) 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 initially located at \( u_1 \). The message cost of \( u_2 \) is consequently 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). Greedily collecting the tokens is not a good strategy in general because it will traverse the same edge more than twice in the worst case.
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 would 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-too-bad tree gives usually a much 
better bound. However, there is an almost tight $\log |S|/\log \log |S|$ 
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 partic-
ular 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 (note that for logarithmic trees, the overhead becomes $\log \log n$).
- What if the spanning tree is a star? Then with Theorem 7.4, each find 
will terminate in 2 steps! Since also an optimal algorithm has message 
would become 4-competitive.
- Thought experiment: Assume a balanced binary spanning tree—by 
Theorem 7.4, the message complexity per operation is $\log n$. Because 
a binary tree has maximum degree 3, the time per operation therefore 
traversing each edge at most twice. In another area of computer science, we would 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.

Algorithm 7.7: Shared Object: Read/Write Caching

- Nodes can either read or write the shared object. For simplicity we first 
assume that reads or writes do not overlap in time (access to the object 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 object.
- Initially the object is stored at a single node $u$; 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 
inveting them!) until it reaches a cached version of the object. Then a copy of 
the object 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.

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 object 
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.
Algorithm 7.9 Shared Object: Pointer Forwarding

Initialization: Object is stored at root \( r \) of a precomputed spanning tree \( T \) (as in the Arrow algorithm, each node has a parent pointer pointing towards the object).

Accessing Object: (by node \( u \))
1. follow parent pointers to current root \( r \) of \( T \)
2. send object from \( r \) to \( u \)
3. \( r \).parent := \( u \); \( u \).parent := \( r \); // \( u \) is the new root

Algorithm 7.10 Shared Object: Ivy

Initialization: Object is stored at root \( r \) of a precomputed spanning tree \( T \) (as before, each node has a parent pointer pointing towards the object). For simplicity, we assume that accesses to the object are sequential.

Start Find Request at Node \( u \):
1. \( u \) sends “find by \( u \)” message to parent node
2. \( u \).parent := \( u \)

Upon \( v \) receiving “Find by \( u \)” Message:
3. if \( v \).parent = \( v \) then
4. send object to \( u \)
5. else
6. send “find by \( u \)” message to \( v \).parent
7. end if
8. \( v \).parent := \( u \) // \( u \) will become the new root

7.3 Ivy and Friends

In the following we study algorithms that do not restrict communication to a tree. Of particular interest is the special case of a complete graph (clique). A simple solution for this case is given by Algorithm 7.9.

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 object, we have message/time complexity \( n \). The new topology is again a linear 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 \( 4 \rightarrow 3 \rightarrow 2 \rightarrow 1 \); 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 \( 2 \rightarrow 1 \rightarrow 3 \); once the message of 4 passes 2, the game re-starts.)

There seems to be a natural improvement of the pointer forwarding idea. Instead of simply redirecting the parent pointer from the old root to the new root, we can redirect all the parent pointers of the nodes on the path visited during a find message to the new root. The details are given by Algorithm 7.10. Figure 7.11 shows how the pointer redirecting affects a given tree (the right tree results from a find request started at node \( x_0 \) on the left tree).

Remarks:
- Also with Algorithm 7.10, we might have a bad linked list situation. However, if the start of the list acquires the object, the linked list turns into a star. As the following theorem shows, the search paths are not long on average. Since paths sometimes can be bad, we will need amortized analysis.

Theorem 7.12. If the initial tree is a star, a find request of Algorithm 7.10 needs at most \( \log n \) steps on average, where \( n \) is the number of processors.

Proof. All logarithms in the following proof are to base 2. We assume that accesses to the shared object 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). We define the potential \( \Phi \) of the whole tree \( T \) as \( (V) \) is the set of all nodes)

\[
\Phi(T) = \sum_{u \in V} \frac{\log s(u)}{2}
\]

Assume that the path traversed by the \( i \)th operation has length \( k_i \), i.e., the \( i \)th operation redirects \( k_i \) pointers to the new root. Clearly, the number of steps of the \( i \)th operation is proportional to \( k_i \). We are interested in the cost of \( m \) consecutive operations, \( \sum_{i=1}^{m} k_i \).

Let \( T_0 \) be the initial tree and let \( T_i \) be the tree after the \( i \)th operation. Further, let \( a_i = k_i - \Phi(T_{i-1}) + \Phi(T_i) \) be the amortized cost of the \( i \)th operation.

We have

\[
\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} (k_i - \Phi(T_{i-1}) + \Phi(T_i)) = \sum_{i=1}^{m} k_i - \Phi(T_0) + \Phi(T_m).
\]

For any tree \( T \), we have \( \Phi(T) \geq \log(n)/2 \). Because we assume that \( T_0 \) is a star, we also have \( \Phi(T_0) = \log(n)/2 \). We therefore get that

\[
\sum_{i=1}^{m} a_i \geq \sum_{i=1}^{m} k_i.
\]
Hence, it suffices to upper bound the amortized cost of every operation. We thus analyze the amortized cost \( a_i \) of the \( i^{th} \) operation. Let \( x_0, x_1, x_2, \ldots, x_k \) be the path that is reversed by the operation. Further for \( 0 \leq j < k \), let \( s_j \) be the size of the subtree rooted at \( x_j \) before the reversal. The size of the subtree rooted at \( x_0 \) after the reversal is \( s_k \), and the size of the one rooted at \( x_j \) after the reversal, for \( 1 \leq j \leq k \), is \( s_j - s_{j-1} \) (see Figure 7.11). For all other nodes, the sizes of their subtrees are the same, therefore the corresponding terms cancel out in the amortized cost \( a_i \). We can thus write \( a_i \) as

\[
a_i = k_i - \left( \sum_{j=0}^{k_i} \frac{1}{2} \log s_j \right) + \left( \sum_{j=0}^{k_i - 1} \frac{1}{2} \log (s_j - s_{j-1}) \right)
\]

\[
= k_i + \frac{1}{2} \sum_{j=0}^{k_i - 1} \left( \log (s_{j+1} - s_j) - \log s_j \right)
\]

\[
= k_i + \frac{1}{2} \sum_{j=0}^{k_i - 1} \log \left( \frac{s_{j+1} - s_j}{s_j} \right).
\]

For \( 0 \leq j < k_i - 1 \), let \( a_j = s_{j+1}/s_j \). Note that \( s_{j+1} > s_j \) and thus that \( a_j > 1 \). Further note, that \( (s_{j+1} - s_j)/s_j = a_j - 1 \). We therefore have that

\[
a_i = k_i + \frac{1}{2} \sum_{j=0}^{k_i - 1} \log (a_j - 1)
\]

\[
= \sum_{j=0}^{k_i - 1} \left( 1 + \frac{1}{2} \log (a_j - 1) \right).
\]

For \( \alpha > 1 \), it can be shown that \( 1 + \log(\alpha - 1)/2 \leq \log \alpha \) (see Lemma 7.13). From this inequality, we obtain

\[
a_i \leq \sum_{j=0}^{k_i - 1} \log a_j = \sum_{j=0}^{k_i - 1} \log (s_{j+1} - s_j) = \sum_{j=0}^{k_i - 1} (\log s_{j+1} - \log s_j)
\]

\[
= \log s_k - \log s_0 \leq \log n,
\]

because \( s_k = n \) and \( s_0 \geq 1 \). This concludes the proof.

**Lemma 7.13.** For \( \alpha > 1 \), \( 1 + \log(\alpha - 1)/2 \leq \log \alpha \).

**Proof.** The claim can be verified by the following chain of reasoning:

\[
0 \leq (\alpha - 2)^2
\]

\[
0 \leq \alpha^2 - 4\alpha + 4
\]

\[
4(\alpha - 1) \leq \alpha^2
\]

\[
\log_2 \left( 4(\alpha - 1) \right) \leq \log_2 (\alpha^2)
\]

\[
2 + \log_2 (\alpha - 1) \leq 2 \log_2 \alpha
\]

\[
1 + \frac{1}{2} \log_2 (\alpha - 1) \leq \log_2 \alpha.
\]

**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 7.1 be a balanced binary tree. If the access to a shared variable for example is “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.

**Chapter Notes**

The Arrow protocol was designed by Raymond [Ray89]. There are real life implementations of the Arrow protocol, such as the Aleph Toolkit [Her99]. The performance of the protocol under high loads was tested in [HW99] and other implementations of the Arrow protocol, such as the Aleph Toolkit [Her99]. It has been shown that the find operations of the protocol do not backtrack, i.e., the time and message complexities are within factor \( O \) of \( D \) (see Lemma 7.13).

For \( \alpha > 1 \), it can be shown that \( 1 + \log(\alpha - 1)/2 \leq \log \alpha \) (see Lemma 7.13). From this inequality, we obtain

\[
a_i \leq \sum_{j=0}^{k_i - 1} \log a_j = \sum_{j=0}^{k_i - 1} \log (s_{j+1} - s_j) = \sum_{j=0}^{k_i - 1} (\log s_{j+1} - \log s_j)
\]

\[
= \log s_k - \log s_0 \leq \log n,
\]

because \( s_k = n \) and \( s_0 \geq 1 \). This concludes the proof.

**Lemma 7.13.** For \( \alpha > 1 \), \( 1 + \log(\alpha - 1)/2 \leq \log \alpha \).

**Proof.** The claim can be verified by the following chain of reasoning:

\[
0 \leq (\alpha - 2)^2
\]

\[
0 \leq \alpha^2 - 4\alpha + 4
\]

\[
4(\alpha - 1) \leq \alpha^2
\]

\[
\log_2 \left( 4(\alpha - 1) \right) \leq \log_2 (\alpha^2)
\]

\[
2 + \log_2 (\alpha - 1) \leq 2 \log_2 \alpha
\]

\[
1 + \frac{1}{2} \log_2 (\alpha - 1) \leq \log_2 \alpha.
\]
BIBLIOGRAPHY

Bibliography


