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)

v sends request up the tree
request processed by root r (atomically)
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).

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)

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.

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).

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.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).

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.*

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 r is its own parent. The variable is initially stored at r. For all nodes v, v.successor := **null**, v.wait := **false**.

Start Find Request at Node *u*:

```
do atomically
  u sends "find by u" message to parent node
  u.parent := u
  u.wait := true
end do
```

Upon *w* Receiving "Find by *u*" Message from Node *v*:

```
do atomically
  if w.parent ≠ w then
    w sends "find by u" message to parent
    w.parent := v
  else
    w.parent := v
    if not w.wait then
      send variable to u      // w holds var. but does not need it any more
    else
      w.successor := u        // w will send variable to u a.s.a.p.
    end if
  end if
end do
```

Upon *w* Receiving Shared Object:

```
perform operation on shared object
do atomically
  w.wait := false
  if w.successor ≠ null then
    send variable to w.successor
    w.successor := null
  end if
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 move 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 move 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 $\mathcal{O}(\log |S| \cdot m^*)$ 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 initially 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 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. $\square$

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 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 (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 cost 1, the algorithm is 2-competitive. . . ? Yes, but because of its high degree the star center experiences contention. It can be shown that the contention overhead is at most proportional to the largest degree Δ of the spanning tree, but in a star the degree is linear in the number of nodes.
- Thought experiment: Assume a balanced binary spanning tree—by Theorem 7.4, the message complexity per operation is $2\log n$. Because a binary tree has maximum degree 3, the contention is at most 3.
- 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 shown how to construct spanning trees that are $\mathcal{O}(\log n)$-stretch-competitive.

What if most nodes just want to read the shared object? Then it does not make sense to acquire a lock every time. Instead we can use caching (see Algorithm 7.7).

Theorem 7.8. *Algorithm 7.7 is correct. More surprisingly, the message complexity is 3-competitive (at most a factor 3 worse than the optimum).*

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 object is stored at w_0 and not cached anywhere else. All reads cost twice the smallest subtree T spanning the write w_0 and all the reads since each read only goes to the first copy. The write w_1 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 3 times (and in P at most once), the theorem follows. $\square$

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 inverting 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)

```
follow parent pointers to current root r of T
send object from r to u
r.parent := u; u.parent := u;                    // 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:

```
u sends "find by u" message to parent node
u.parent := u
```

Upon v receiving "Find by u" Message:

```
if v.parent = v then
   send object to u
else
   send "find by u" message to v.parent
end if
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 $4 \to 3 \to 2 \to 1$; 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 $2 \to 1 \to 3$; 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

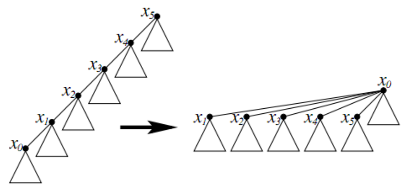

Figure 7.11: Reversal of the path $x_0, x_1, x_2, x_3, x_4, x_5$.

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 Φ 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

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

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 $x_0, x_1, x_2, \ldots, x_{k_i}$ be the path that is reversed by the operation. Further for $0 \leq j \leq k_i$, 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_i} and the size of the one rooted at x_j after the reversal, for $1 \leq j \leq k_i$, 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 ammortized cost a_i. We can thus write a_i as

$$\begin{aligned}
a_i &= k_i - \left(\sum_{j=0}^{k_i} \frac{1}{2}\log s_j\right) + \left(\frac{1}{2}\log s_{k_i} + \sum_{j=1}^{k_i}\frac{1}{2}\log(s_j - s_{j-1})\right) \\
&= k_i + \frac{1}{2}\cdot\sum_{j=0}^{k_i-1}\left(\log(s_{j+1}-s_j) - \log s_j\right) \\
&= k_i + \frac{1}{2}\cdot\sum_{j=0}^{k_i-1}\log\left(\frac{s_{j+1}-s_j}{s_j}\right).
\end{aligned}$$

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

$$\begin{aligned}
a_i &= k_i + \frac{1}{2}\cdot\sum_{j=0}^{k_i-1}\log(\alpha_j - 1) \\
&= \sum_{j=0}^{k_i-1}\left(1 + \frac{1}{2}\log(\alpha_j - 1)\right).
\end{aligned}$$

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

$$\begin{aligned}
a_i &\leq \sum_{j=0}^{k_i-1}\log\alpha_j = \sum_{j=0}^{k_i-1}\log\frac{s_{j+1}}{s_j} = \sum_{j=0}^{k_i-1}(\log s_{j+1} - \log s_j) \\
&= \log s_{k_i} - \log s_0 \leq \log n,
\end{aligned}$$

because $s_{k_i} = n$ 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:

$$\begin{aligned}
0 &\leq (\alpha-2)^2 \\
0 &\leq \alpha^2 - 4\alpha + 4 \\
4(\alpha-1) &\leq \alpha^2 \\
\log_2\left(4(\alpha-1)\right) &\leq \log_2\left(\alpha^2\right) \\
2 + \log_2(\alpha-1) &\leq 2\log_2\alpha \\
1 + \frac{1}{2}\log_2(\alpha-1) &\leq \log_2\alpha.
\end{aligned}$$

□

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. We might be able to use synchronizers (or maybe some other timing tricks) to help meeting a little bit.

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 and variations of the protocol were given in, e.g., [PR99, HTW00].

It has been shown that the find operations of the protocol do not backtrack, i.e., the time and message complexities are $\mathcal{O}(D)$ [DH98], and that the Arrow protocol is fault tolerant [HT01]. Given a set of concurrent request, Herlihy et al. [HTW01] showed that the time and message complexities are within factor $\log R$ from the optimal, where R is the number of requests. Later, this analysis was extended to long-lived and asynchronous systems. In particular, Herlihy et al. [HKTW06] showed that the competitive ratio in this asynchronous concurrent setting is $\mathcal{O}(\log D)$. Thanks to the lower bound of the greedy TSP heuristic, this is almost tight.

The Ivy system was introduced in [Li88, LH89]. On the theory side, it was shown by Ginat et al. [GST89] that the amortized cost of a single request of the Ivy protocol is $\Theta(\log n)$. Closely related work to the Ivy protocol on the practical side is research on virtual memory and parallel computing on loosely coupled multiprocessors. For example [BB81, LSHL82, FR86] contain studies on variations of the network models, limitations on data sharing between processes and different approaches.

Later, the research focus shifted towards systems where most data operations were read operations, i.e., efficient caching became one of the main objects of study, e.g., [MMVW97].

Arrow and Ivy are designed for trees or cliques, respectively. Arrow and Ivy are combinable, as one can choose whether to point to directly to the source or not – the Arvy protocol is the generalization of Arrow and Ivy [KW19]. Recently, other efficient shared object protocols for general networks have been proposed [SBS12].

Bibliography

[BB81] Thomas J. Buckholtz and Helen T. Buckholtz. Apollo Domain Architecture. Technical report, Apollo Computer, Inc., 1981.

[DH98] Michael J. Demmer and Maurice Herlihy. The Arrow Distributed Directory Protocol. In *Proceedings of the 12th International Symposium on Distributed Computing (DISC)*, 1998.

[FR86] Robert Fitzgerald and Richard F. Rashid. The Integration of Virtual Memory Management and Interprocess Communication in Accent. *ACM Transactions on Computer Systems*, 4(2):147–177, 1986.

[GST89] David Ginat, Daniel Sleator, and Robert Tarjan. A Tight Amortized Bound for Path Reversal. *Information Processing Letters*, 31(1):3–5, 1989.

[Her99] Maurice Herlihy. The Aleph Toolkit: Support for Scalable Distributed Shared Objects. In *Proceedings of the Third International Workshop on Network-Based Parallel Computing: Communication, Architecture, and Applications (CANPC)*, pages 137–149, 1999.

[HKTW06] Maurice Herlihy, Fabian Kuhn, Srikanta Tirthapura, and Roger Wattenhofer. Dynamic Analysis of the Arrow Distributed Protocol. In *Theory of Computing Systems, Volume 39, Number 6*, November 2006.

[HT01] Maurice Herlihy and Srikanta Tirthapura. Self Stabilizing Distributed Queuing. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, pages 209–223, 2001.

[HTW00] Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Ordered Multicast and Distributed Swap. In *Operating Systems Review, Volume 35/1, 2001. Also in PODC Middleware Symposium, Portland, Oregon*, July 2000.

[HTW01] Maurice Herlihy, Srikanta Tirthapura, and Roger Wattenhofer. Competitive Concurrent Distributed Queuing. In *Twentieth ACM Symposium on Principles of Distributed Computing (PODC)*, August 2001.

[HW99] Maurice Herlihy and Michael Warres. A Tale of Two Directories: Implementing Distributed Shared Objects in Java. In *Proceedings of the ACM 1999 conference on Java Grande (JAVA)*, pages 99–108, 1999.

[KW19] Pankaj Khanchandani and Roger Wattenhofer. The Arvy Distributed Directory Protocol. In *31st ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), Phoenix, AZ, USA*, June 2019.

[LH89] Kai Li and Paul Hudak. Memory Coherence in Shared Virtual Memory Systems. *ACM Transactions on Computer Systems*, 7(4):312–359, November 1989.

[Li88] Kai Li. IVY: Shared Virtual Memory System for Parallel Computing. In *International Conference on Parallel Processing*, 1988.

[LSHL82] Paul J. Leach, Bernard L. Stumpf, James A. Hamilton, and Paul H. Levine. UIDs as Internal Names in a Distributed File System. In *Proceedings of the First ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 34–41, 1982.

[MMVW97] B. Maggs, F. Meyer auf der Heide, B. Voecking, and M. Westermann. Exploiting Locality for Data Management in Systems of Limited Bandwidth. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 1997.

[PR99] David Peleg and Eilon Reshef. A Variant of the Arrow Distributed Directory Protocol with Low Average Complexity. In *Proceedings of the 26th International Colloquium on Automata, Languages and Programming (ICALP)*, pages 615–624, 1999.

[Ray89] Kerry Raymond. A Tree-based Algorithm for Distributed Mutual Exclusion. *ACM Transactions on Computer Systems*, 7:61–77, 1989.

[SBS12] Gokarna Sharma, Costas Busch, and Srivathsan Srinivasagopalan. Distributed Transactional Memory for General Networks. In *26th International Parallel and Distributed Processing Symposium (IPDPS), Shanghai, China*, May 2012.