Exclusion and Object Tracking in a Network of Processes*

Yih-Kuen Tsay†
Dept. of Information Management
National Taiwan University, TAIWAN

Chien-Chung Huang
Department of Computer Science
Dartmouth College, U.S.A.

Abstract

This paper concerns two fundamental problems in distributed computing—mutual exclusion and mobile object tracking. For a variant of the mutual exclusion problem where the network topology is taken into account, all existing distributed solutions make use of tokens. It turns out that these token-based solutions for mutual exclusion can also be adapted for object tracking, as the token behaves very much like a mobile object. To handle objects with replication, we go further to consider the more general k-exclusion problem which has not been as well studied in a network setting. A strong fairness property for k-exclusion requires that a process trying to enter the critical section will eventually succeed even if *up to* $k-1$ processes stay in the critical section indefinitely.

We present a comparative survey of existing token-based mutual exclusion algorithms, which have provided much inspiration for later k-exclusion algorithms. We then propose two solutions to the k-exclusion problem, the second of which meets the strong fairness requirement. Fault-tolerance issues are also discussed along with the suggestion of a third algorithm that is also strongly fair. Performances of the three algorithms are compared by simulation. Finally, we show how the various exclusion algorithms can be adapted for tracking mobile objects.

1 Introduction

A distributed system stores and manages various shared resources so that they can be conveniently accessed by users of the system. We refer to an instance of any of these resources as an *object*. The mutual exclusion problem is fundamental in such a system, as a shared object typically may be accessed by one user at a time to ensure consistency. Locating an object so as to deliver messages such as operation requests intended for the object is also fundamental; the problem becomes more complicated when the object may move. The mutual exclusion problem has a longer history and is more extensively studied, while object tracking is a central problem in the management of mobile objects which has attracted much attention recently.

Mutual exclusion in a network of processes is different from that in a shared memory system and is solved in a different way. Under the network model, the privilege to exclusively access a shared object, or to enter the critical section, is typically materialized by the possession of a unique token. An algorithm for the mutual exclusion problem essentially needs to address (1) how a request is forwarded to the token holder and (2) how processes change their states as the system evolves to reflect the new location of the token.

The task of locating an object in a distributed system is usually performed by a directory or name service of the system. The relevant directory service may be centralized at a particular server or distributed across a number of servers or even the entire system. Disregarding these variations, certain distributed data structure has to be maintained to keep track of the objects

*Dartmouth Computer Science Technical Report 2007-608

†Email: tsay@im.ntu.edu.tw; post: No. 1, Sec. 4, Roosevelt Rd., Taipei 106, Taiwan; tel: +8862 3366 1189

so that a request can be routed to the intended object. Forwarding a request to the node (or process) where a particular mobile object resides is very similar to forwarding a request to the token holder in a mutual exclusion algorithm. The distributed data structure of a mutual exclusion algorithm intuitively can be adapted as the distributed directory for tracking a mobile object.

To handle objects with replication, we consider the more general k-exclusion problem where the *exclusion* (safety) property requires that at most k processes are allowed in the critical section at any time and the *basic fairness* (liveness) property requires that, if no process stays in the critical section indefinitely, every process trying to enter the critical section will eventually succeed. Mutual exclusion is then the special case of $k = 1$. The k-exclusion problem was first defined by Fischer [6]. Though quite a few algorithms, e.g., [1, 3, 2], have been proposed for a shared memory system, the problem has not been as well studied in a network setting. A stronger fairness property, referred to as **Starvation-Freedom with Concurrency (SFC)** here, requires that a process trying to enter the critical section will eventually succeed even if *up to* $k-1$ processes stay in the critical section indefinitely. SFC implies that the response time of a good solution is insensitive to how long a particular process stays in the critical section (or holds a copy of the object in the context of sharing mobile objects).

We present a brief comparative survey of existing token-based mutual exclusion algorithms, which have provided much inspiration for later k-exclusion algorithms. Building upon the ideas of these token-based algorithms, we then propose two solutions to the k-exclusion problem, the second of which meets the SFC requirement. Fault-tolerance issues are also discussed along with the suggestion of a third algorithm that also meets SFC. Performances of the three algorithms are compared by simulation. Finally, we show how the various exclusion algorithms can be adapted for tracking mobile objects.

Related Work Van de Snepscheut [15] was probably the first to give solutions to the mutual exclusion problem in a network of processes, extending the earlier work for rings by Martin [8]. The main idea was to orient the edges of the network so that they point to the token holder; when the token moves, the directions of the edges are updated accordingly. Raymond [13] proposed another algorithm which turned out to be identical to a restricted version of Van de Snepscheut's where a rooted spanning tree instead of a directed acyclic graph (DAG) is maintained. However, he was able to give a more detailed analysis of the average case message complexity, which is $O(\log N)$.

Naimi *et al.* [10] presented yet another spanning tree-based solution for networks whose topology is a complete graph (or equivalently, for general networks with an underlying end-to-end routing service). Unlike the previous algorithms, the edge set of the tree maintained by their algorithm changes over time. They were also able to derive an average case message complexity of $O(\log N)$ (where the end-to-end transmission of a message is counted as one message).

Demmer and Herlihy [5] proposed the so-called arrow distributed directory protocol for keeping track of mobile objects in a distributed system. They noted the close relationship between distributed mutual exclusion and object tracking. Their main algorithmic technique can be seen as a combination of those of Van de Snepscheut and Naimi *et al.* The preceding four algorithms can be classified as token-based; they introduce distributed data structures that are also applicable to object tracking. We shall review these algorithms in more detail shortly.

Bulgannawar and Vaidya [4] extended the work of Naimi *et al.* by using k dynamic spanning trees to route requests. Their algorithm does not meet SFC. Walter *et al.* [16] generalized the routing topology of Van de Snepscheut to a multi-sink DAG for managing k tokens. They considered a mobile environment where network links may fail. However, their algorithm meets

only a restricted variation of SFC, as we shall explain in a subsequent section.

An early work by Mullender and Vitányi [9] had suggested the relationship between distributed mutual exclusion and object tracking (or name service, in their terminology). They showed that mutual exclusion and name service can be formulated as special instances of the so-called distributed match-making problem. However, the formulations seem to be biased toward particular types of algorithms.

There has been a considerable amount of work on the subject of object location (we use the term object tracking in this paper to emphasize that the object being located may move). PRR (Plaxton, Rajaraman, and Richa) [11], Chord [14], CAN [12], and Tapestry [7] represent a most recent line of such research. These location schemes either do not handle mobility of an object or simply treat it as a combination of object deletion and insertion. Moreover, these schemes can all be classified as *home-based* in that every object is mapped to a fixed home and all messages for an object (or at least the first of messages in the same session) must be routed through its home. The home of an object may be a potential bottleneck of performance. In contrast, the tracking schemes derived from token-based exclusion algorithms do not assign a fixed home to an object.

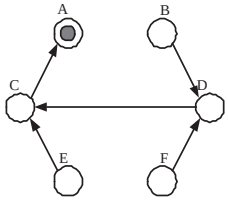
2 Token-Based Mutual Exclusion Algorithms

We briefly review and compare four token-based algorithms for mutual exclusion. Their basic ideas are explained with illustrative diagrams for the ease of comprehension. Part of the material has been moved to the Appendix due to the page limit.

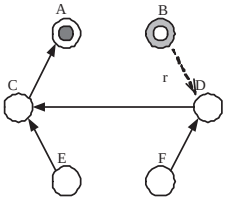
Van de Snepscheut [15] and Raymond [13] The edges of the network are oriented to form a single-sink *directed acyclic graph* (DAG) with the sink holding the token. When a node wishes to enter the critical section, it sends a request along one of the directed paths to the token holder. The next node in the path will either relay the request along one of its outgoing edges or put the request in a local queue. Each node (as an originator or forwarder) permits at most one outstanding request while keeping all others in its local queue. The token is routed in the reverse order along the path that the request has travelled. The direction of each edge incident to a node is updated to point to the node, when the node receives the token; this maintains acyclicity and also ensures that the token holder is the only sink of the entire graph. A diagram illustrating these ideas can be found in the Appendix.

Raymond's algorithm is essentially a restricted version of Van de Snepscheut's where a rooted spanning tree instead of a DAG is maintained. Raymond further considered recoverable node failures. His idea was for a recovering node to restore its state through assistance from its immediate neighbors. However, disregarding the problem of multiple failures, the fault-tolerance measure still leaves processes that are not in the same partition with the token holder to wait probably indefinitely.

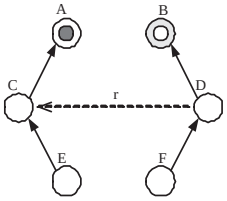
Naimi *et al.* [10] The algorithm of Naimi *et al.* also uses a tree structure, but their tree is for routing requests only and is more dynamic with a changing set of edges. An additional distributed queue tells the current token holder and subsequent holders which node is the next in line waiting for the token (in contrast, Van de Snepscheut's and Raymond's algorithms use separate local queues). Conceptually, the root of the routing tree is also the tail (and sometimes the head as well) of the distributed queue, where some other process may be added (what really happens is more complicated with possibly multiple trees and roots and hence multiple queues



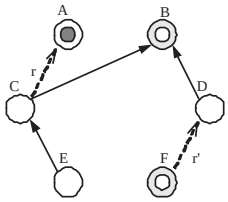
(a) A holds the token and is the root, treating itself as its parent.



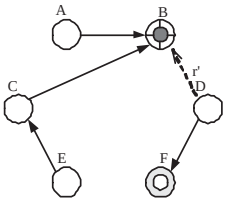
(b) B sends a request r to its parent D and makes itself a new root.



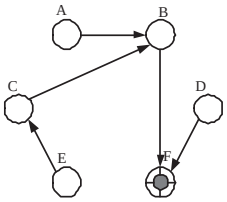
(c) D relays request r to its parent C and makes B its new parent.



(d) C relays request r to its parent A and makes B its new parent. A will add B to the queue when it receives r. F also initiates a request r'.



(e) A releases the token to B (via the assumed routing service) and makes B its new parent. D forwards request r' to B.



(f) B leaves the critical section, sends the token to F, and makes F its new parent.

Figure 1: How the algorithm of Naimi *et al.* works.

being created and merged). They assume that the network is a complete graph, which makes it possible for a node to switch its parent to any other node. Practically, this assumption boils down to working in a general network with an underlying routing service. The "physical link" between each pair of nodes in their model corresponds to the "logical link" realized by the routing service between the pair in the general network.

Initially, the tree is a star-shaped one (any spanning tree would also be suitable) with the token holder as the root and as the parent of every other node. The initial token holder is also the only node, the head, and the tail of the distributed queue. When a node wants to enter the critical section, it sends a request to its parent. It then regards itself as the new root, expecting that, once its request is received by the current root, it will be added to the distributed queue and become the new tail of the queue. Other nodes on the directed path to the current root, upon receiving the request, make the request originator their new parent. As the request travels to its destination, tree edges are removed and added and the tree is temporarily split into two smaller trees. Finally, the current root inserts the request originator behind itself in the queue and also makes the originator its new parent, turning itself into a non-root node and thus merging the two smaller trees back into one. The head of queue, after holding the token for a finite amount of time, will pass out the token to the next node and remove itself from the queue. Every node in the queue eventually will get the token. A scenario can be found in Figure 1.

As shown in the figure, more than one processes may be trying to enter the critical section. Though the overall changes to the tree and the queue may be more complicated, processes behave just as described in the preceding paragraph. A new root may receive some other request even before it is actually added to the queue; it handles the request just as the current root would do. The isolated segment of queue gets hooked back to the distributed queue when the new root is eventually added to the queue. A root allows just one process to be added behind it in the queue. Once such an addition occurs, the root has got a new parent, i.e., the request originator; a second request will be forwarded to the new parent.

Demmer and Herlihy [5] Though originally intended for mobile object tracking, Demmer and Herlihy's algorithm closely resembles that of Naimi *et al.* The main difference is that their tree has a fixed set of edges which is more like the tree in Van de Snepscheut's (restricted tree version) and Raymond's algorithms. They also use a distributed queue for lining up the processes waiting for the token, which is identical to that of Naimi *et al.*; the tree tells where the tail of the distributed queue is.

3 Algorithm A

Algorithm A generalizes the tree version of Van de Snepscheut's algorithm to maintain k tokens. Initially, k tokens are created and arbitrarily distributed among the nodes of the network. Each node records the number of tokens it holds and, for each incident tree edge e, it also maintains a pair (t_e, r_e) of numbers, where t_e indicates "the number of tokens in the subtree that e leads to" and r_e "the number of outstanding requests sent along e". The difference $t_e - r_e$ provides an *estimate* of the number of free tokens in the subtree that e leads to.

When a node wishes to enter the critical section but does not have a free token, it sends a request along a tree edge e such that $t_e - r_e$ is the highest among all incident tree edges. A receiving node of the request that is without a free token passes the request along a *different* tree edge e' with highest possible $t_{e'} - r_{e'}$. The request eventually will reach either a node with a free token or a node where further forwarding would be fruitless (when every $t_e - r_e$ value is zero). In the first case, the free token backtracks along the path that the request has travelled. In the second case, the request is put into a local queue of the last receiving node and will be forwarded when $t_e - r_e$ for some edge e become positive. Further care much be taken to prevent deadlocks, but we omit the details here. It is interesting to note that Algorithm A degenerates into the mutual exclusion algorithm of Van de Snepscheut when k equals 1.

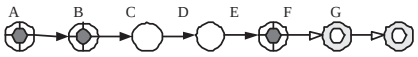
A diagram illustrating the basic ideas of Algorithm A can be found in the Appendix. As the forwarding of a request may stop at a leaf node that happens to be in the critical section and the node might hold the corresponding token indefinitely, Algorithm A fails to meet SFC.

4 Algorithm B

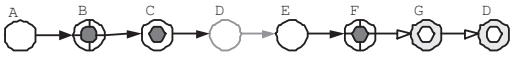
Algorithm B is built upon the ideas of Naimi *et al.* and Demmer and Herlihy. It assumes the existence of an underlying routing service that allows a node to send messages to any other designated node. A distributed *request-routing* tree is maintained. Either the tree of Naimi *et al.* or that of Demmer and Herlihy can be applied, but we assume using the former one. To line up the nodes wishing to enter the critical section, we use a distributed *token-relay* tree instead (a queue is a special case of a rooted tree). A node may have up to two occurrences in the tree, as we shall explain shortly.

Initially, the k tokens are assigned to the only node and the root of the token-relay tree, which is also the root of the request-routing tree. When a request is initiated, it will be routed to the root of the request-routing tree as described before. The root makes the request originator its "(immediate) successor" in the token-relay tree. We use "successor" and "predecessor" (instead of "parent" and "child" or "ancestor" and "descendant") when talking about the token-relay tree. A waiting node enters the critical section when it receives the first token; it passes subsequent tokens (and the first token after it leaves the critical section) to its (immediate) successor.

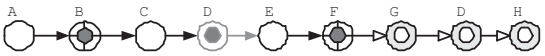
Consider a node that has entered and left the critical section, but is still in the token-relay



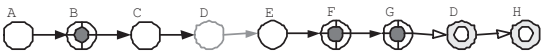
(a) A, B, E are in the critical section. F and G are in the trying section and still waiting for tokens to arrive.



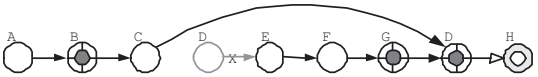
(b) A leaves the critical section and passes the corresponding token to B, which forwards it to C. D initiates a request and becomes the successor of G.



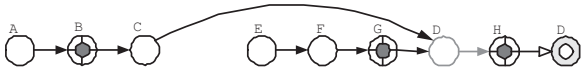
(c) C forwards the token to D. D cannot enter the critical section as discussed in the paper. H initiates a request.



(d) The token is passed from D to E, to F, then to G. G enters the critical section.



(e) G relays the token from F to D. D enters the critical section and also deletes the pointer to E.



(f) D passes the token to H, which enters the critical section. D subsequently initiates yet another request.

Figure 2: How the *token-relay* tree of Algorithm B works (the case of $k = 3$).

tree (because it has not seen all k tokens yet) and wants to enter the critical section again. A tempting idea is to let the node just sit and wait for a token to arrive from its predecessors. However, this will not meet SFC, as its predecessors may refuse to release the tokens, while there might be a free token residing in one of its successors.

Our solution is to let the node join the token-relay tree again (thus allowing a node to have two occurrences in the tree). A waiting node distinguishes its predecessors as from two groups: the *new pre-group* which includes the predecessors (of its newer occurrence) that are also its successors (of its older occurrence) and the *old pre-group* that includes all its other predecessors.

The waiting node will eventually receive a token from either the new pre-group or the old pre-group. In the first case, all nodes in the new pre-group must have entered the critical section and are not starving. Therefore, the waiting node can keep the token and enter the critical section. It can also eliminate its first occurrence from the token-relay tree, making the second occurrence, if any, to be the first and the sole occurrence of the tree; this may be regarded as "transplanting" all subtrees having a path to its first occurrence to its second occurrence. In the second case (the token is from the old pre-group), it passes the token to its successor that is also in the new pre-group to prevent starvation of those nodes in that group. An illustrative scenario is given in Figure 2; the detailed code and its correctness proof can be found in Appendix A.

5 Fault Tolerance: Algorithm C

k-exclusion algorithms such as Algorithm A that use network links directly to route requests or tokens are vulnerable to link failures. The assumption of a routing service, as in Algorithm B, relegates the fault-tolerance responsibility to the underlying routing service. In this section, we discuss how link failures may be handled directly. We first review the work of Walter *et al.* [16] which has tackled the same issue and then propose our third k-exclusion algorithm.

The algorithm of Walter *et al.* can be seen as an extension of Van de Snepscheut's algorithm. A multiple-sink DAG is maintained for routing requests, in which any non-token holding node can follow a directed path to some token holder. As in the algorithm of Van de Snepscheut, the request is forwarded along the directed edges until it arrives at some token; the token backtracks

the very same path and the edges have to be re-directed to ensure that any non-token-holding node always can reach some token holder. When a node detects the failure of some link over which the outstanding request has been sent, it re-routes the request along some other available path.

A node permits at most one outstanding request and keeps others in its local queue. To enhance concurrency, they introduced a technique called "*token forwarding*". The idea is that free tokens are circulated systematically around the network so that they may possibly arrive in regions with high contention.

Their algorithm meets a restricted variation of SFC, where exactly $k-1$ processes stay in the critical section indefinitely. There is an anomalous behavior that they seemed to fail to notice. Consider the following scenario: there exist two distant sinks in the DAG, each of which holds a token. Suppose that some other node initiates a request and the request is routed to one of the two sinks, which unfortunately refuses to release the token. The other sink holds a second free token. Now further suppose that one of its neighbors has exactly one outgoing edge that is connected to the second sink. If the second sink and its neighbor repeatedly attempt to enter the critical section and ask for the free token, then the token will "oscillate" between the two nodes and never have a chance of being routed to other parts of the network by the technique of "token forwarding".

Algorithm C We seek another way of extending the work of Van de Snepscheut. Initially, a *primary* token is assigned to one of the processes in the network. A counter is kept in the primary token to record the number of *secondary* tokens currently in the system so as to guarantee that there are at most $k-1$ of them at a time. The algorithm maintains a directed acyclic graph of the network such that all edges are directed toward the primary token holder. Any process holding either the primary token or a secondary token may enter the critical section.

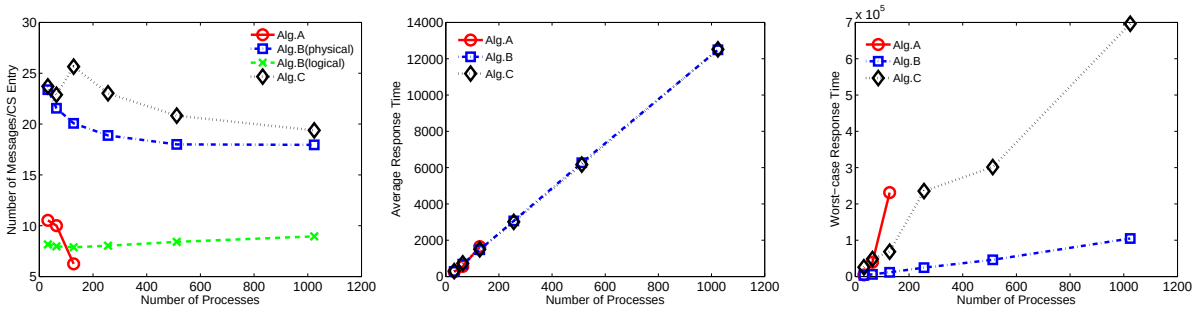
A request is routed along the directed edges until it reaches the primary token holder; any path may be chosen if there exist more than one paths to the primary token. If the primary token is not in use, it is sent to the request originator; the directions of the edges change accordingly as the token moves. Otherwise, the primary token holder generates a secondary token for the request originator or, if the counter has reached $k-1$, puts the request in a local queue. The movement of a secondary token does not change the direction of an edge.

If a node has sent a request through a link that fails, it sends the request again through a second link (if there is any) directed towards the primary token holder; otherwise, it waits until the failed link recovers. The orientation of a recovered link may be determined according to *which of the two end-processes most recently held the primary token* since they detected the last failure of the link. For each failed link, that piece of information may be recorded in the primary token.

When a process leaves the critical section, if the token it holds is primary and the local queue of requests is not empty, it sends the token to the first request originator in the queue and forwards all other requests to the new token holder. If the token is secondary, the process simply returns it to the primary token holder. The latter destroys the token and decrements the secondary token count if there is no pending request in the local queue; otherwise, it forwards the secondary token to fulfill the request. An illustrative diagram for Algorithm C can be found in the Appendix.

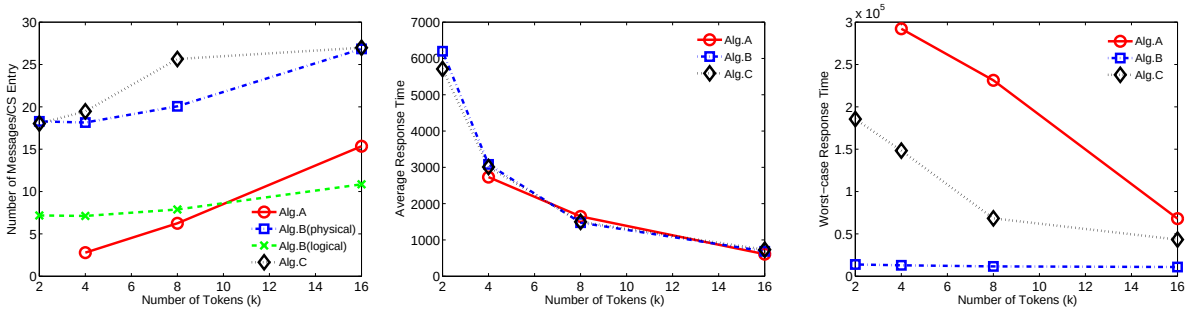
A variant of Algorithm C may be derived by allowing requests to be intercepted by a node that has relayed a secondary token on one of its incoming edges. When the secondary token is returned and arrives at this node, the token can be sent directly to the originator of the intercepted request (without being returned to the primary token holder). Intercepting a request

reduces the number of exchanged messages, but may increase the response time (depending how quickly the secondary token is released by its holder). Moreover, this variation fails to meet SFC because of repeated interceptions.



(a) Average message cost (b) Average response time (c) Worst-case response time

Figure 3: Comparative results with varying numbers of processes; k=8; average time in the critical section = 100.



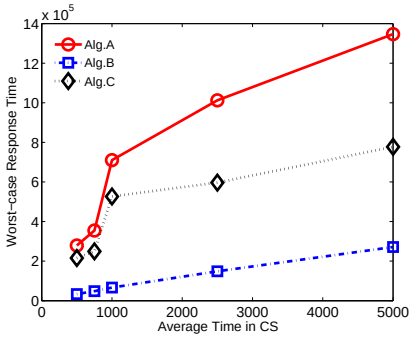
(a) Average message cost (b) Average response time (c) Worst-case response time

Figure 4: Comparative results with varying numbers of tokens; number of nodes = 128; average time in the critical section = 100.

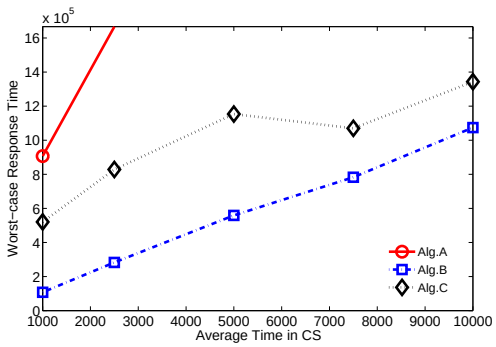
6 Performance Evaluation

We conducted an extensive simulation study to (comparatively) evaluate the message costs and response times of the three algorithms. Simulation experiments were carried out for different scales (numbers of nodes) of random networks with varying settings for the number of tokens and the average time that a node spends in the critical section. Durations of the remainder section and of the critical section are modeled as Poisson processes, while the transmission time of a message over a link is uniformly distributed over $(0, 100]$ (simulation time units). All underlying networks are created at random with a connectivity of 0.05. Performance results with varying network scales are shown in Figure 3; similarly, those with different numbers of tokens (k) and those with regard to average time that a node spends in the critical section can be found in Figure 4 and 5, respectively. When an algorithm causes some nodes to starve within the simulation time, we omit to plot its data.

In terms of message cost, as can be seen from Figure 3(a) and Figure 4(a), Algorithm A is the most efficient. We have measured the counts of messages over both the logical (end-to-end) links and the physical links for Algorithm B, which assumed an underlying routing service. In



(a) Worst-case response time; number of processes= 64; $k = 4$.



(b) Worst-case response time; number of processes= 128; $k = 8$.

Figure 5: Comparative results with varying average times that a node spends in the critical section.

Figure 4(a), the message costs of the three algorithms increase with the number of tokens for different reasons. For Algorithms A and C, higher k allows for more outstanding requests (and secondary tokens) being circulated around the network. For Algorithm B, the cost increases because more tokens are being routed in the token-relay tree. In terms of average response time, the three algorithms are comparably equal, disregarding the network scale or the number of tokens, as can be seen from Figure 3(b) and Figure 4(b).

From Figure 3(c) and Figure 4(c) and the whole Figure 5, one can see that the worst-case response times differ considerably with Algorithm B being the best, while Algorithm A often starves its nodes. The difference can be attributed to whether the algorithms satisfy SFC.

7 Adaptations for Object Tracking

There are many possibilities regarding what a process wants from an object. As our main concern is for a process to find the whereabout of a mobile object, we assume that a requesting process always wants the object to be moved to the process's site and then performs a read or write operation on it. An object tracking solution needs to meet the following requirements:

- (Safety) There is at most one legitimate copy of the object in the system at any time.
- (Liveness) A requesting process will eventually acquire the object.

With the object replaced by a token, the above two requirements become those for a token-based solution to mutual exclusion. The distributed data structure of a token-based algorithm can therefore be adapted as the distributed directory for tracking a mobile object.

An object may sometimes be replicated to enhance availability. We assume that some other mechanism controls the replication of an object and is not part of the object tracking problem; however, the number of copies never exceeds a predefined bound k. The requirements for tracking replicated objects are as follows:

- (Safety) There are at most k legitimate copies of the object in the system at any time.
- (Liveness) A requesting process will eventually acquire a copy of the object assuming up to $k-1$ copies may be held indefinitely by other processes. (An alternative formulation may allow a process to insist on getting a particular copy if that copy is never held indefinitely.)

One way of adapting token-based exclusion algorithms to track k replicated objects is to deploy k copies of a mutual exclusion algorithm and replace a token with a corresponding object replica. This should be more message-efficient than flooding the network. However, there is a problem with this adaptation. If a particular copy that a process tries to acquire is being used, the process either has to wait for the copy to become available or try another copy (which may be in use, too). The first choice may incur undue delay, while the second will cost more messages, probably without any gain. One solution is to have the k copies of algorithm collaborate so that different object replicas can be attempted simultaneously without additional cost, which is exactly what Algorithm A achieves. Another solution is to adapt an algorithm like Algorithm B which fulfills the much desired property of SFC.

Demmer and Herlihy had outlined a solution for tracking replicated objects [5], which is an extension of their algorithm reviewed in Section 2. They assumed that one copy of the object is designated as the *primary* copy. A process wishing to write the object must first acquire the primary copy and then "invalidate" the secondary copies (without actually moving them). A similar solution can be obtained by adapting Algorithm C to incorporate links to secondary copies so that the primary copy holder can actively invalidate secondary copies.

One can also view these solutions as a combination of a mutual exclusion algorithm and a k-exclusion algorithm. The mutual exclusion algorithm is used for acquiring the primary copy, while the k-exclusion algorithm is for acquiring or invalidating a secondary copy. Our algorithms provide alternatives for the needed k-exclusion algorithm.

8 Conclusion

We have presented three solutions to the k-exclusion problem in a network setting, by exploring the ideas of earlier token-based mutual exclusion algorithms. Two of the algorithms satisfy the fairness requirement of Starvation-Freedom with Concurrency. We hope that we have shed some new lights on solutions for tracking mobile objects by deriving them from token-based exclusion algorithms. One distinctive property of the derived solutions is that no fixed home is assigned to an object. These "homeless" tracking schemes, unlike the home-based ones, seem to avoid the performance bottleneck of a home. However, if an object and its replicas require a separate directory structure, the schemes will not scale to a large number of different objects. The merits of such homeless schemes remain to be further studied.

References

[1] Y. Afek, D. Dolev, E. Gafni, and N. Shavit. A bounded first-in-first-enabled solution to the l-exclusion problem. *ACM TOPLAS*, 16(3):939–953, 1994.

[2] K. Alagarsamy and K. Vidyasankar. Fair and efficient mutual exclusion algorithms. In *DISC, LNCS 1693*, pages 166–179, 1999.

[3] J. Anderson and M. Moir. Using k-exclusion to implement resilient, scalable shared objects. In *ACM PODC*, pages 141–150, 1994.

[4] S. Bulgannawar and N. Vaidya. A distributed k-mutual exclusion algorithm. In *ICDCS*, pages 153–160, 1995.

[5] M. Demmer and M. Herlihy. The arrow distributed directory protocol. In *DISC, LNCS 1499*, pages 119–134, 1998.

[6] M. Fischer, N. Lynch, J. Burns, and A. Borodin. Resource allocation with immunity to process failure. In *IEEE FOCS*, pages 78–92, 1979.

[7] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Zhao. Distributed object location in a dynamic network. In *ACM SPAA*, pages 41–52, 2002.

[8] A. Martin. Distributed mutual exclusion on a ring of processors. *Science of Computer Programming*, 5:265–276, 1985.

[9] S. Mullender and P. Vitányi. Distributed match-making. *Algorithmica*, 3:367–391, 1988.

[10] M. Naimi, M. Tréhel, and A. Arnold. A log(n) distributed mutual exclusion algorithm based on path reversal. *Journal of Parallel and Distributed Computing*, 34:1–13, 1996.

[11] C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *ACM SPAA*, pages 311–320, 1997.

[12] S. Ratnasamy, P. Francis, M. Handley, and R. Karp. A scalable content-addressable network. In *ACM SIGCOMM*, pages 161–172, 2001.

[13] K. Raymond. A tree-based algorithm for distributed mutual exclusion. *ACM Transactions on Computer Systems*, 7(1):61–77, August 1989.

[14] I. Stoica, R. Morris, D. Karger, M. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *ACM SIGCOMM*, pages 149–160, 2001.

[15] J. Van de Snepscheut. Fair mutual exclusion on a graph of processes. *Distributed Computing*, 2:113–115, 1987.

[16] J. Walter, G. Cao, and M. Mohanty. A k-mutual exclusion algorithm for wirelesss ad hoc networks. In *ACM POMC*, pages 51–57, 2001.

A Code of Algorithm B and Its Correctness

The detailed code of Algorithm B is given in Figure 6. We prove its correctness in two parts. To not lose the intuition, we will be informal yet rigorous enough so that our proofs permit obvious conversion to more formal ones (for example, in the assertional style) without undue ingenuity.

Before embarking on the proof, we recap the conventions of a typical distributed computing model with reliable asynchronous message-passing communication. The code of Algorithm B and its user processes are collectively referred to as the system or network. Every action as given in the code of Algorithm B is atomic, i.e., only the state before or after an action is observable when we speak of an assertion about states of the system. A variable with subscript i, such as $\mathsf{num\_of\_tokens}_i$, refers to that variable of node i; nodes (processes) are numbered from 1 through n. Messages in transit are part of the state and are observable after having been sent and before having been received. We assume that messages from the same source and to the same destination are delivered in FIFO order. An action, if enabled continuously, will eventually get executed. For instance, Action B1 is enabled when the user makes a transition from *remainder* to *trying* (i.e., it requests to enter the critical section). The action will remain enabled continuously until executed, at which point it becomes disabled.

A.1 Exclusion

The *exclusion* (safety) property requires that at most k processes are in the critical section at any time. It is clear from the code that every transition from *trying* to *critical* (temporarily) removes one token from the system and every transition from *critical* to *exit* eventually gives back the token. To prove the exclusion property, it suffices to show that the sum of the following quantities is exactly k at any time:

t_n: sum of all $\mathsf{num\_of\_tokens}_i$ $(1 \le i \le n)$,

t_m: sum of the c's in all token messages Token(c,j) in transit,

t_c: number of processes in the *critical* section, and

t_e: number of processes in the *exit* section.

The equality $t_n + t_m + t_c + t_e = k$ is obviously true initially (after every node has executed its initialization action) and it is preserved by every action.

VARIABLES:

state: *trying*, *critical*, *exit*, or *remainder*. Transitions from *remainder* to *trying* and from *critical* to *exit* are controlled by "the user" of this code and other transitions by this code (node i).

parent: identifier of node i's parent node in the request-routing tree.

first_next: identifier of successor of node i's *first* occurrence in the token-relay tree.

second_next: identifier of successor of node i's *second* occurrence in the token-relay tree.

predecessor: the identifier of node i's *latest* predecessor in the token-relay tree.

num_of_tokens: number of *free* tokens held by node i.

MESSAGES:

Req(j): request message carrying the identifier of the request originator.

Token(c,j): token message carrying some number of tokens and the identifier of the sender.

Notify(j): notification message to the prospective successor carrying the identifier of the sender.

INITIALIZATION: Initially, node 1 is the root of the request-routing tree and also the single node and the root of the token-relay tree, holding all k tokens.

B0: On being initialized /* Assume state has been initialized to *remainder* by "the user". */

if $i = 1$
then num_of_tokens := k; parent := *nil*;
else num_of_tokens := 0; parent := 1;
first_next := *nil*; second_next := *nil*; predecessor := *nil*;

ACTIONS:

B1: On observing "*remainder* → *trying*"

if num_of_tokens > 0 /* If true, node i must also be the root of the token-relay tree. */
then state := *critical*; num_of_tokens := num_of_tokens − 1;
else send Req(i) to parent; parent := *nil*; predecessor := *nil*;

B2: On observing "*critical* → *exit*"

if first_next = *nil*
then num_of_tokens := num_of_tokens + 1;
else send Token(1,i) to first_next; /* Node i must have just one free token. */
state := *remainder*;

B3: On receiving Token(c,j)

if j = predecessor /* If true, node i is *trying* and the tokens are from the new pre-group. */
then state := *critical*; num_of_tokens := num_of_tokens + c − 1; predecessor := *nil*;
  if second_next ≠ *nil* /* If true, node i has two occurrences in the token-relay tree.*/
  then first_next := second_next; second_next := *nil*;
  else if parent = *nil* **then** first_next := *nil*;
else num_of_tokens := num_of_tokens + c;
if num_of_tokens > 0 **and** first_next ≠ *nil* /* Pass all free tokens to the latest successor if any. */
then send Token(num_of_tokens,i) to first_next; num_of_tokens := 0;

B4: On receiving Req(j)

if parent = *nil*
then send Notify(i) to j;
  if first_next = *nil*
  then first_next := j;
  else second_next := j;
  if num_of_tokens > 0
  then send Token(num_of_tokens,i) to first_next; num_of_tokens := 0;
else send Req(j) to parent;
parent := j;

B5: On receiving Notify(j)

predecessor := j;

Figure 6: Code of Algorithm B for Node i.

A.2 Fairness

We will prove directly the *Starvation-Freedom with Concurrency* (SFC) property, which implies the basic fairness property. The gist of the proof lies in precisely defining the query-routing tree and the token-relay tree, which we have only spoken of conceptually in the description of the algorithm, and associated rank functions that measure the progress of a process trying to enter the critical section.

We start with defining the query-routing tree. One simple definition would be using just the values of parent_i's, but this would force us to reason about a forest of trees most of the time as the system evolves; recall that a node sets its parent to *nil* when it initiates a request, breaking the tree into two smaller trees. (Note: Naimi *et al.* [10] reasoned about this forest, which makes their proof rather operational and harder to follow.) To avoid the complication, we make use also of request messages that are in transit, which help glue the smaller trees together. Precisely, we define the query-routing tree with the set T_r of tree edges as follows:

$$\begin{array}{lll} (i,j) \in T_r & \text{iff} & (1)\ \mathsf{parent}_i = j \text{ or} \\ & & (2)\ \text{“a request message is in transit from node } i \text{ to node } j\text{”} \end{array}$$

It can be shown by the usual assertional argument that T_r (with edge directions ignored) is a spanning tree of the network at any time. For convenience, tree edges of kind (1) are said to be *solid* and tree edges of kind (2) *vanishing*. A node i with an outstanding request Req(i) and the node that will eventually receive the request with $\mathsf{parent} = nil$ reside on two different subtrees connected by the vanishing edge over which Req(i) is in transit. The subtree where node i resides will grow, while the other where the eventual recipient of Req(i) resides will shrink (an appropriate rank function may be defined to reflect this). This is so because when a node j receives a request message Req(i) over a vanishing tree edge (k, j), the vanishing edge is removed from T_r and a solid edge (j, i) is added to T_r (see Action B4). If parent_j was l immediately before the receipt of the request, node j forwards the request Req(i) to node l, making (j, l) a vanishing edge. Since the network is finite, every request will eventually be received by a node with $\mathsf{parent} = nil$.

We now define the token-relay tree. The shape of the token-relay tree depends not only on the values of $\mathsf{first\_next}_i$'s and $\mathsf{second\_next}_i$'s but also on parent_i's. The latter dependency is needed, as we have deliberately chosen not to count the number of tokens that a node has relayed (to simplify the code). A node may have its $\mathsf{first\_next}$ point to some other node, but actually has no further tokens to relay. Such a node must tell if it is the “root” (by checking $\mathsf{parent} = nil$), otherwise it would forward tokens to the process pointed to by its $\mathsf{first\_next}$, an erroneous act.

Let $(i, 1)$ denote the first occurrence of node i and $(i, 2)$ the second occurrence in the tree. When a node, pointed to by the $\mathsf{first\_next}$ or $\mathsf{second\_next}$ of another node, has two occurrences, we need to distinguish which of the occurrences is being pointed to. For this purpose, it is convenient to speak of “the corresponding Req(j) was initiated while $\mathsf{first\_next}_j \neq nil$” when node i with $\mathsf{parent}_i = nil$ receives a request. This can be achieved by superimposing the code to tag Req(j) with a boolean value: the tag is *true* if $\mathsf{first\_next}_j \neq nil$ and *false* otherwise. The set T_t of tree edges of the token-relay tree is defined as follows:

$$\left\{ \begin{array}{lll} ((i,1),(j,1)) \in T_t & \text{iff} & (\mathsf{first\_next}_i = j \text{ and } \mathsf{parent}_i \neq nil) \text{ and } (\mathsf{second\_next}_j = nil \text{ or} \\ & & (\mathsf{predecessor}_j \neq i \text{ and “no Notify}(i) \text{ is in transit from } i \text{ to } j\text{”})) \\ ((i,1),(j,2)) \in T_t & \text{iff} & (\mathsf{first\_next}_i = j \text{ and } \mathsf{parent}_i \neq nil) \text{ and } (\mathsf{predecessor}_j = i \text{ and} \\ & & \text{“the corresponding Req}(j) \text{ was initiated while } \mathsf{first\_next}_j \neq nil\text{”}) \\ ((i,2),(j,1)) \in T_t & \text{iff} & (\mathsf{second\_next}_i = j) \text{ and } (\mathsf{second\_next}_j = nil \text{ or} \\ & & (\mathsf{predecessor}_j \neq i \text{ and “no Notify}(i) \text{ is in transit from } i \text{ to } j\text{”})) \\ ((i,2),(j,2)) \in T_t & \text{iff} & (\mathsf{second\_next}_i = j) \text{ and } (\mathsf{predecessor}_j = i \text{ and} \\ & & \text{“the corresponding Req}(j) \text{ was initiated while } \mathsf{first\_next}_j \neq nil\text{”}) \end{array} \right.$$

When the request Req(j) of a node j is eventually received by a node i with $\mathsf{parent}_i = nil$ (which much occur as proven earlier), node j will be pointed to by either $\mathsf{first\_next}_i$ or $\mathsf{second\_next}_i$ (Action B4) and be added to the token-relay tree. Consider the case of $\mathsf{second\_next}_i$ being set to j and the request Req(j) having been tagged *true*. In this case, $((i, 2), (j, 2))$ will be added to T_t after a slight delay for the message Notify(j) to arrive in node i at which point $\mathsf{predecessor}_j$ becomes i (Action B5). Once a node has been

added to the token-relay tree (as either the first or the second occurrence), it remains to show that the node will eventually receive a token from its predecessor (the node recorded in its predecessor) and enter the critical section (Action B3).

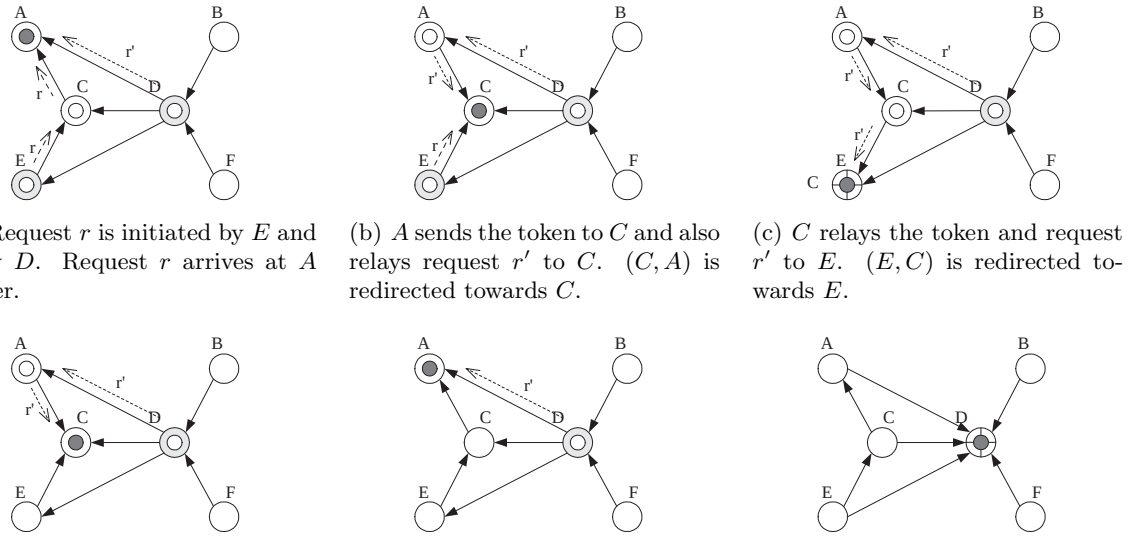
In terms of the token-relay tree T_t, Algorithm B enjoys the following property (simplified):

> If a node i in T_t is in the *trying* section, then every token is either (1) held by an occurrence of node that lies on a path directed towards the latest occurrence of node i or (2) in transit between two nodes that lie on a path directed towards the latest occurrence of node i.

The statement is simplified, as some edges lower in the tree may be formed later than those higher in the tree. A similar and more accurate property may be stated in "stages" as pieces of the tree are connected together. Now, let us consider an arbitrary node i in T_t that is in the *trying* section. If a node eventually leaves the critical section and releases the corresponding token, the token will advance along a path directed towards node i (more precisely, the latest occurrence of node i). Any other free token will also advance along some path directed towards node i. (An appropriate rank function may be defined to measure the progress more precisely.) By induction, node i will eventually receive a token and enter the critical section even if *up to* $k-1$ nodes stay in the critical section (and keep a token) indefinitely.

B Further Details of Token-Based Mutual Exclusion Algorithms

Figure 7 shows a scenario of Van de Snepscheut's algorithm.



(a) Request r is initiated by E and r' by D. Request r arrives at A earlier.

(b) A sends the token to C and also relays request r' to C. (C, A) is redirected towards C.

(c) C relays the token and request r' to E. (E, C) is redirected towards E.

(d) E leaves the critical section and sends the token to C. (C, E) is redirected towards C.

(e) C relays the token to A. (A, C) is redirected towards A.

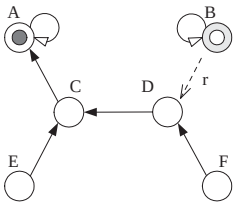
(f) A relays the token to D. (D, A), (D, C), and (D, E) are redirected towards D.

Figure 7: How Van de Snepscheut's algorithm works.

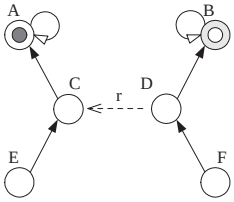
Demmer and Herlihy [5] Though originally intended for mobile object tracking, Demmer and Herlihy's algorithm closely resembles that of Naimi *et al.* The main difference is that their tree has a fixed set of edges which is more like the tree in Van de Snepscheut's (restricted tree version) and Raymond's algorithms. They also use a distributed queue for lining up the processes waiting for the token, which is identical to that of Naimi *et al.*; the tree tells where the tail of the distributed queue is. An underlying routing service is assumed for transmitting the token; as we pointed out earlier, the routing service provides a logical complete communication network that Naimi *et al.* assumed.

The algorithm starts with a spanning tree of the network whose root is the node holding the unique token. Changes to the orientation of the edges occur while requests are processed. When a node wants

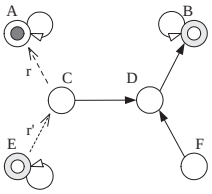
to enter the critical section, it sends a request to its parent and then regards itself as the new root. The algorithm differs from that of Naimi *et al.* in how a node changes its parent. Each of the intermediate nodes on the directed path to the current root, after forwarding the request to the next node, makes the precedent node its new parent (i.e., the direction of the corresponding edge is reversed). Finally, the current root inserts the request originator behind itself in the queue and also makes the precedent node its new parent. When this is done, the direction of the path that the request has travelled is reversed, pointing to the new root.



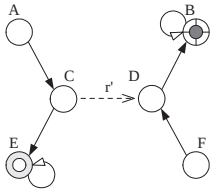
(a) B sends a request r to its parent D and makes itself a new root.



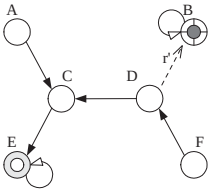
(b) D relays request r to its parent C and makes B its new parent.



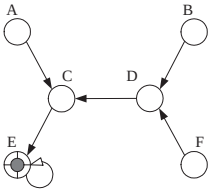
(c) C relays request r to A and makes D its new parent. A will add B to the queue when it receives r. E also initiates a request r'.



(d) A sends the token to B (via the assumed routing service) and makes C its new parent. C relays request r' to D.



(e) D relays request r' to B and makes C its new parent. B will add E to the queue when it receives r'.



(f) B leaves the critical section and sends the token to E (also via the assumed routing service).

Figure 8: How Demmer and Herlihy's algorithm works.

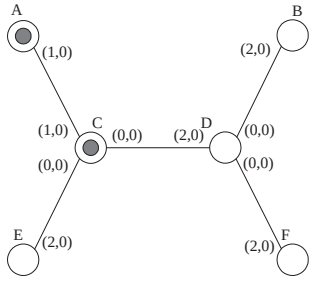
More than one processes may be trying to enter the critical section. But again, like in the algorithm of Naimi *et al.*, although the overall changes to the tree and the queue may be more complicated, processes behave just as described above. Figure 8 illustrates a typical scenario.

C An Illustrative Diagram for Algorithm A

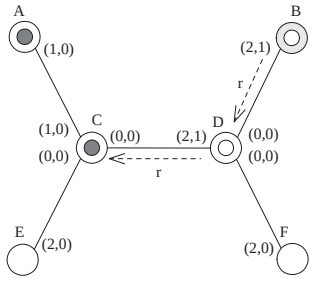
Figure 9 illustrates the basic ideas of Algorithm A.

D An Illustrative Diagram for Algorithm C

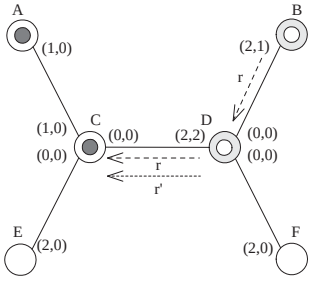
Figure 10 illustrates the basic ideas of Algorithm C.



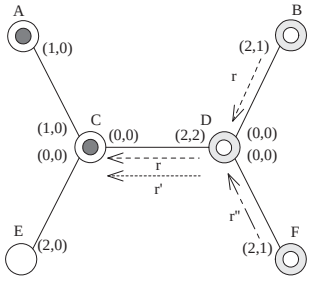
(a) For D, the pair $(2,0)$ on edge $\{C,D\}$ indicates 2 tokens on the subtree $\{C,D\}$ leads to and no outstanding requests from D to that subtree, etc.



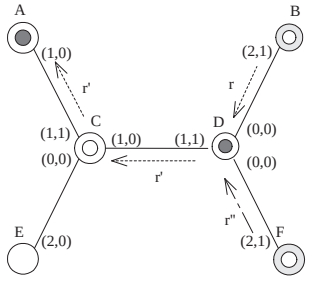
(b) B initiates a request r to D and the pair $(2,0)$ on edge $\{B,D\}$ becomes $(2,1)$. D relays r to C.



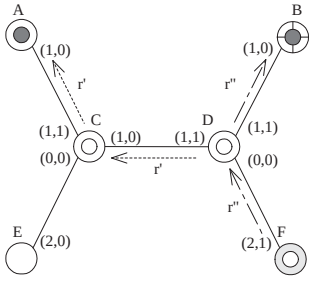
(c) D initiates another request r' to C and the pair $(2,1)$ on edge $\{C,D\}$ becomes $(2,2)$.



(d) F initiates request r'' to D, which puts r'' in its queue.

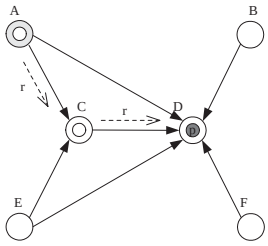


(e) C sends a token to D and the pair $(0,0)$ on $\{C,D\}$ becomes $(1,0)$. It also relays r' to A.

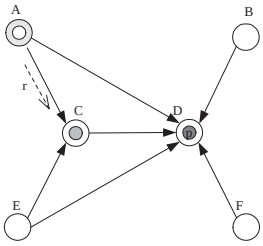


(f) D relays the token as well as r'' to B. B enters the critical section.

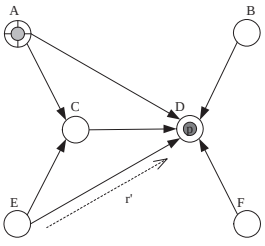
Figure 9: How Algorithm A works (the case of $k = 2$).



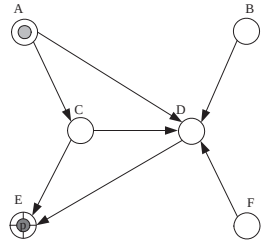
(a) D holds the primary token. A initiates a request r to C, which relays it to D.



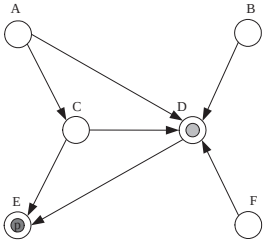
(b) D creates a secondary token and sends it to C.



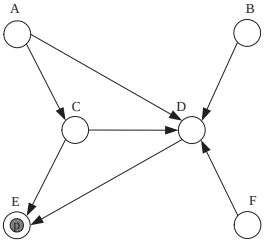
(c) C relays the secondary token to A. E initiates a request r' to D.



(d) D sends the primary token to E. (E,D) and (E,C) are redirected towards E.



(e) A returns the secondary token to D (A may also send it via C).



(f) D relays the secondary token to E.

Figure 10: How Algorithm C works (the case of $k = 2$).