Bounding Lamport's Bakery Algorithm

Prasad Jayanti, King Tan, Gregory Friedland, and Amir Katz

6211 Sudikoff Lab for Computer Science
Dartmouth College, Hanover, NH 03755
prasad@cs.dartmouth.edu
kytan@cs.dartmouth.edu

Abstract. Lamport's Bakery algorithm is among the best known mutual exclusion algorithms. A drawback of Lamport's algorithm is that it requires unbounded registers for communication among processes. By making a small modification to Lamport's algorithm, we remove the need for unbounded registers. The main appeal of our algorithm lies in the fact that it overcomes a drawback of a famous algorithm while preserving its elegance.

1 Introduction

Mutual Exclusion is a classic synchronization problem that has been extensively studied (see [5] for a survey of mutual exclusion algorithms). This problem is described as follows. There are n asynchronous processes with each process repeatedly performing four sections of code: remainder section, entry section, critical section and exit section. It is assumed that no process fails in the entry or exit sections and every process that enters the critical section eventually leaves it. The problem is to design a protocol for entry and exit sections that satisfies the following properties:

Mutual Exclusion: No two processes are in the critical section at the same time.
Starvation freedom: Each process in the entry section eventually enters the critical section.
Wait-free exit: Each process can complete the exit section in a bounded number of steps, regardless of the speeds of other processes.

The following fairness property is also desirable in any mutual exclusion protocol:

Doorway FIFO: The entry section begins with a straight line code called the *wait-free doorway*: a process can execute the doorway in a bounded number of steps, regardless of the speeds of other processes. If process P_i completes executing the doorway before process P_j begins executing the doorway, P_i enters the critical section before P_j.

L. Pacholski and P. Ružička (Eds.): SOFSEM 2001, LNCS 2234, pp. 261–270, 2001.

Lamport's Bakery algorithm is among the best known mutual exclusion algorithms [2]. It is discussed in introductory Operating Systems textbooks [6] and in books on Concurrent Algorithms [1,4,5] and is well-known to the general Computer Science community. The algorithm satisfies all of the above properties, is elegant and has an intuitive appeal: processes assign themselves tokens in the doorway, and enter the critical section in the order of their token numbers. One drawback of Lamport's algorithm is that it requires unbounded registers for communication among processes. It is this drawback that our paper overcomes: with a simple modification, we show how Lamport's Bakery algorithm can be made to work with small bounded registers. Our algorithm adds just two lines and makes a small change to an existing line. Lamport's algorithm and our algorithm are presented in Figures 1 and 2, respectively. (Lines 3 and 8 in Figure 2 are new and line 4 is a slightly modified version of the corresponding line in Figure 1.)

Our algorithm compares with Lamport's algorithm as follows:

Properties: Our algorithm, like Lamport's algorithm, has all properties stated above, including the doorway FIFO property.
Size of registers: Lamport's algorithm requires unbounded registers while ours requires bounded registers, each of size either 1 bit or $\log 2n$ bits. This is the highlight of our algorithm.
Type of registers: Lamport's algorithm requires $2n$ single-writer multi-reader registers. Our algorithm requires the corresponding $2n$ single-writer multi-reader registers and an additional register that we call X. The register X is written by the process in the critical section. Therefore, there is at most one write operation on X at any time. Yet, strictly speaking, it is not a single-writer register because different processes write to it (although never concurrently).
Lamport's algorithm works even if registers are only safe, but our algorithm requires atomic registers.

Algorithms with better space complexity than ours are known: to our knowledge, the best algorithm, from the point of view of size of registers, is due to Lycklama and Hadzilacos [3]. Their algorithm has all properties stated above and requires only $5n$ safe boolean registers. Thus, the appeal of our algorithm is not due to its low space complexity, but because it shows that a small modification removes a well-known drawback of a famous algorithm while preserving its elegance.

1.1 Organization

Lamport's Bakery algorithm (Figure 1), which employs unbounded token values, is the starting point of our work and is described in Section 2. Our approach to bounding token values requires these values to be clustered. In Section 3, we show that Lamport's algorithm does not have the clustering property. In Section 4, we present a modification of Lamport's algorithm that continues to use unbounded tokens, but has the clustering property. Then, in Section 5, we present a bounded version of this algorithm.

2 Lamport's Algorithm

[initialize $\forall i : 1 \leq i \leq n : (gettoken_i :=$ `false`
$token_i := -1)$]

1. $gettoken_i :=$ `true`
2. $S := \{token_j | 1 \leq j \leq n\}$
3. $token_i := max(S) + 1$
4. $gettoken_i :=$ `false`
   for $j \in \{1 \cdots n\} - \{i\}$
5. **wait till** $gettoken_j =$ `false`
6. **wait till** $(token_j = -1) \vee ([token_i, i)] < [token_j, j])$
7. CS
8. $token_i := -1$

Fig. 1. Lamport's Bakery algorithm : P_i's protocol

We first describe Lamport's Bakery algorithm (Figure 1). The algorithm is based on the following idea. When a process gets into the entry section, it assigns itself a token that is bigger than the existing tokens. It then waits for its turn, i.e., until every process with a smaller token has exited the critical section (CS).

Each process P_i maintains two variables—$token_i$ and $gettoken_i$. The purpose of $token_i$ is for P_i to store its token number. A value of -1 in $token_i$ indicates that P_i is either in the remainder section or in the process of assigning itself a token. The variable $gettoken_i$ is a boolean flag. P_i sets it to *true* when it is in the process of assigning itself a token.

We now informally describe the actual lines of the algorithm. P_i sets $gettoken_i$ to *true* to signal that it is in the process of assigning itself a token (line 1). It then reads the tokens held by all other processes (line 2) and assigns itself a bigger token (line 3). Notice that if another process P_j is also computing a token concurrently with P_i, both P_i and P_j may end up with the same token number. Therefore, when comparing tokens (line 6), if the token values of two processes are equal, the process ID is used to decide which token should be considered smaller. Having set its token, P_i sets the flag $gettoken_i$ to *false* (line 4). Lines 1-4 constitute the doorway.

P_i then considers each process P_j in turn and waits until it has "higher priority" than P_j, i.e., until it is certain that P_j does not have (and will not later have) a smaller token than P_i's. This is implemented on lines 5 and 6, and the intuition is described in the rest of this paragraph. Essentially, P_i needs to learn P_j's token value to determine their relative priority. P_i can learn P_j's token value by reading $token_j$ except in one case: P_i cannot rely on $token_j$ when P_j is in the process of updating it (i.e., P_j is on lines 2 or 3). This is because, when P_j is on lines 2 or 3, even though the value of $token_j$ is -1, P_j may be about to write into $token_j$ a smaller value than P_i's token.

Specifically, for each P_j, P_i first waits until P_j's flag ($gettoken_j$) is false (line 5), i.e., until P_j is not in the process of assigning itself a token. After this wait, P_i can be certain of the following fact: if P_j assigns itself a new token before P_i has exited CS, P_j's token will be bigger than P_i's token since (on line 2) P_j will surely read $token_i$. P_i then waits until either P_j is in the remainder section or P_j's token is bigger than its token (line 6). At the end of this wait, P_i can be certain that it has a higher priority than P_j: if P_j is in the entry section or will enter the entry section, P_j's token will be bigger than P_i's and hence P_j will be forced to wait at line 6 until P_i has exited CS. Thus, when the for-loop terminates, P_i is certain that it has higher priority than all other processes. So it enters CS (line 7). When it exits CS, it sets $token_i$ to -1 to indicate that is in the remainder section.

3 Our Approach to Bounding Token Values

In Lamport's algorithm, let max_t and min_t denote the maximum and minimum non-negative token values at time t (they are undefined if all tokens are -1). Let $range_t$ be $max_t - min_t$. Lamport's algorithm is an unbounded algorithm because max_t can increase without bound. What is more pertinent for our purpose of bounding tokens, however, is the fact that $range_t$ can also increase without bound. In Section 3.1 we substantiate this claim that $range_t$ can grow unbounded. Our approach to deriving a bounded algorithm depends on limiting the growth of $range_t$. This approach is described in Section 3.2.

3.1 Unbounded Separation of Tokens in Lamport's Algorithm

Below we describe a scenario to show that the value of $range_t$ can increase without bound. Consider a system of two processes P_1, P_2 in the following state: Both P_1 and P_2 have just completed executing line 4. The values of $token_1$, $token_2$ are respectively either 0, v (for some $v \geq 0$) or v, 0. The range is therefore v.

- P_1 observes that $gettoken_2 = false$ (line 5). P_2 observes that $gettoken_1 = false$ (line 5).
- If $[token_1, 1] < [token_2, 2]$, let $P_{i_1} = P_1$,$P_{i_2} = P_2$. Otherwise, let $P_{i_1} = P_2$,$P_{i_2} = P_1$. Note that the value of $token_{i_2}$ is v. P_{i_1} executes line 6, enters and exits CS, and writes -1 in $token_{i_1}$. P_{i_1} then begins a new invocation of the protocol. P_{i_1} reads v in $token_{i_2}$, computes $v+1$ as the new value for $token_{i_1}$. P_{i_1} stops just before writing $v+1$ in $token_{i_1}$.
- P_{i_2} executes line 6, enters and exits CS, and writes -1 in $token_{i_2}$. P_{i_2} then begins a new invocation of the protocol. P_{i_2} reads -1 in $token_{i_1}$, and writes 0 in $token_{i_2}$. P_{i_2} then executes line 4.
- P_{i_1} writes $v+1$ in $token_{i_1}$ and executes line 4. The range is now $v+1$. The system state now is identical to the system state at the beginning of the scenario, except that the range has increased by 1.

The above scenario took us from a system state with range v to a system state with range $v + 1$. This scenario can be repeated arbitrarily many times, causing the range to increase without bound.

3.2 The Main Idea

Our approach to bounding the token values in Lamport's algorithm consists of two steps. In the first step, we introduce a mechanism that forces the token values to form a narrow cluster, thereby ensuring a small bounded value for $range_t$. The resulting algorithm, which we call UB-Bakery, still uses unbounded tokens, but the tokens are clustered. In the second step, we observe that the token clustering makes it possible to replace integer arithmetic with modulo arithmetic. The resulting algorithm, which we call B-Bakery, achieves our goal of using small bounded token values. The first step is described section 4 and the second step in Section 5.

4 Algorithm with Unbounded and Clustered Tokens

To implement the first step described above, we introduce a new shared variable X which stores the token value of the latest process to visit CS. The new algorithm, UB-Bakery, is in Figure 2. It makes two simple modifications to Lamport's algorithm. First, a process writes its token value in X (line 8) immediately before it enters CS. The second modification is on lines 3 and 4. In addition to reading the tokens of all the processes, P_i also reads X (line 3). P_i then computes its new token value as 1+ (maximum of the values of all processes' tokens and X). No other changes to Lamport's algorithm are needed.

To see the usefulness of X, let us revisit the scenario described in the previous section. There, P_{i_1} read the value v in $token_{i_2}$, but P_{i_2} read -1 in $token_{i_1}$. Consequently, while P_{i_1} set its token to $v+1$, P_{i_2} set its token to 0. This possibility, of one process adopting a large token value and the other a small token value, is what causes the range to increase without bound. In the new algorithm, this is prevented because, even though P_{i_2} might read -1 in $token_{i_1}$, it would find v in X. As a result, P_{i_2}'s token value would be $v + 1$ also. More importantly, the token values of P_{i_1} and P_{i_2} would be close to each other (in this scenario, they would be the same).

Intuitively, the new algorithm ensures that X grows monotonically and all non-negative token values cluster around X. The exact nature of this clustering is stated and proved in the next subsection.

4.1 Properties of UB-Bakery

We now state the desirable clustering properties of UB-Bakery (Theorems 2 and 3), and give proof outlines of how they are achieved. Rigorous proofs of these properties are provided in the full version of this paper.

Observation 1 *The value of X is non-decreasing.*

```
[initialize X := 0;
   ∀i : 1 ≤ i ≤ n : (gettoken_i := false
      token_i := −1) ]

1.    gettoken_i := true
2.    S := {token_j | 1 ≤ j ≤ n}
3.    x := X
4.    token_i := max(S ∪ {x}) + 1
5.    gettoken_i := false
      for j ∈ {1 ··· n} − {i}
6.       wait till gettoken_j = false
7.       wait till (token_j = −1) ∨ ([token_i, i)] < [token_j, j])
8.    X := token_i
9.    CS
10.   token_i := −1
```

Fig. 2. Algorithms UB-Bakery and B-Bakery : P_i's protocol

Proof Sketch: Suppose the value of X decreases. Specifically, let $x_i, x_j, (x_i > x_j)$ be two consecutive values of X, with x_j immediately following x_i. Let P_i, P_j be the processes that write x_i, x_j in X respectively. Thus, P_j immediately follows P_i in the order of their entering CS.

If P_j reads $token_i$ (line 2) after P_i has written x_i in $token_i$ (line 4), then P_j either reads x_i in $token_i$ (line 2) or reads x_i in X (line 3). Thus, x_j, the value of $token_j$ computed by P_j (line 4), must be greater than x_i. This contradicts our assumption that $x_i > x_j$. Therefore, P_j reads $token_i$ (line 2) before P_i writes x_i in $token_i$ (line 4). P_i must subsequently wait until $gettoken_j = false$ (line 6). Since P_j must first write x_j in $token_j$ before setting $gettoken_j$ to *false*, P_i will read x_j in $token_j$ when it executes line 7. Since $x_i > x_j$, P_i will not exit its waiting loop w.r.t. P_j (line 7) until P_j has exited CS and set $token_j$ to -1. This contradicts our assumption that P_i enters CS before P_j. This completes the proof of Observation 1.

□

Theorem 1 (Bounded token range). *At any time t and for any j, let v be the value of $token_j$ and x be the value of X. If $v \neq -1$, then $x \leq v \leq x + n$.*

Proof Sketch: This Theorem asserts that at any time t, the non-negative token values lie in the range $[x, x + n]$, where x is the value of X at t. We now give an informal proof of this Theorem. Suppose Theorem 1 is false. If $v < x$, then P_j has not entered CS at time t. Suppose P_j subsequently writes v in X at time t' (prior to entering CS). The value of X decreases at some time in $[t, t']$ (because $v < x$). This violates Observation 1. If $v > x + n$, then there is some integer a, where $x + 1 \leq a \leq x + n$, such that a is not the token value of any process at t. (This is true for the following reason: Excluding v, there are at

most $n-1$ distinct token values at t, whereas there are n integers in the interval $[x+1, x+n]$.) Since v is the value of $token_j$ at t, and $v > a$, then a must have been the token value of some process P_k at some time before t, This implies that P_k has written a in X at some time t' before t. Since $a > x$, the value of X decreases at some time in the interval $[t', t]$. This again violates Observation 1. This completes the proof.

□

Theorem 2 (Token cluster around X). *Let v be the value of $token_j$ read by P_i executing line 2. Let x be the value of X read by P_i executing line 3. Then, $(v \neq -1) \Rightarrow x-(n-1) \leq v \leq x+(n-1)$.*

Proof Sketch: This Theorem states that any token value v read by P_i on line 2 lies within the range $[x-(n-1), x+(n-1)]$, where x is the value of X read by P_i on line 3. We say that the token values v read on line 2 *cluster* around the *pivot* x. To establish Theorem 2, we first prove two Observations:

Observation 2 *Let t_1 be the time P_i executes line 1 and t_2 be the time P_i executes line 3. Then, any process P_j executes line 8 (writing the value of $token_j$ in X) at most once in (t_1, t_2).*

Proof Sketch: This observation is true for the following reason: Suppose P_j executes line 8 for the first time in (t_1, t_2), and then begins a new invocation of UB-Bakery. Within the interval (t_1, t_2), P_j cannot proceed beyond line 6 of the invocation, where P_j waits for $gettoken_i$ to become false (because $gettoken_i = true$ throughout (t_1, t_2)).

□

Observation 3 *Let t_1 be the time P_i executes line 1 and t_2 be the time P_i executes line 3. Let the value of $token_j$ be v_1 and v_2 at t_1 and t_2, respectively. Let v be the value that P_i reads in $token_j$ on line 2. If $v \neq -1$, then either $v = v_1$ or $v = v_2$.*

Proof Sketch: Suppose $v \neq v_1$. Then P_j must have written v in $token_j$ at some time t, where $t_1 < t < t_2$. Consider the invocation by P_j during which P_j writes v in $token_j$. P_j cannot exit its waiting loop on line 6 w.r.t. P_i at any time before t_2 (because $gettoken_i = true$ throughout (t_1, t_2)). Therefore the value of $token_j$ is v at t_2, i.e. $v = v_2$.

□

Now we continue with the proof sketch of Theorem 2. Theorem 1 establishes that the token range is bounded at any time. Further, the token values are bounded below by the value of X. Observation 2 implies that the value of X increases by no more than $(n-1)$ in (t_1, t_2). Observation 3 says that any non-negative value of $token_j$ read by P_i on line 2 is the value of $token_j$ at either t_1 or t_2. Manipulating the inequalities that result from these assertions gives Theorem 2.

□

Theorem 3 (Token cluster around $token_i$). *Let v_i be the value of $token_i$ (written by P_i on line 4) when P_i is executing line 7. Let v_j be the value of $token_j$ read by P_i when executing line 7. If $v_j \neq -1$, then $v_i-(n-1) \leq v_j \leq v_i+(n-1)$.*

Proof Sketch: This Theorem states that all non-negative token values read by P_i on line 7 cluster around the pivot $token_i$ (which is unchanged throughout the interval during which P_i executes line 7). The proof proceeds as follows: Suppose P_i reads $token_j$ (line 7) at time t. By contradiction, suppose $v_j < v_i - (n-1)$ (resp. $v_i < v_j - (n-1)$) at time t. Then, there is some integer a, $v_j < a < v_i$ (resp. $v_i < a < v_j$), such that a is not the token value of any process at t. (This is true for the following reason: There are at most n distinct token values, whereas there are more than n integers in the interval $[v_j, v_i]$ (resp. $[v_i, v_j]$).) Since v_i (resp. v_j) is the value of $token_i$ (resp. $token_j$) at t, and $v_i > a$ (resp. $v_j > a$), then a must have been the token value of some process P_k at some time before t. Therefore the value of X was a at some time t' before t. Let x be the value of X at t. By Theorem 1, $x \leq v_j$ (resp. $x \leq v_i$). This implies that $x < a$. This in turn implies that the value of X decreases at some time in $[t', t]$, which contradicts Observation 1. This completes the proof.

□

5 Algorithm with Bounded Tokens

In the algorithm UB-Bakery, the values in X and $token_i$ increase without bound. However, by exploiting token clustering (Theorems 2 and 3), it is possible to bound these values. This is the topic of this section.

The main idea is that it is sufficient to maintain the value of X and the non-negative values of $token_i$ modulo $2n-1$. Specifically, let B-Bakery denote the Bounded Bakery algorithm whose text is identical to UB-Bakery (Figure 2), except that the addition operation on line 4 is replaced with *addition modulo* $2n-1$, denoted by $\oplus$, and the operators *max* and $<$ (on lines 4 and 7, respectively) are replaced with max and $\prec$ respectively (these new operators will be defined shortly). Thus, in B-Bakery, we have $X \in \{0, 1, \ldots, 2n-2\}$ and $token_i \in \{-1, 0, 1, \ldots, 2n-2\}$.

Define a function f that maps values that arise in UB-Bakery algorithm to values that arise in B-Bakery algorithm as follows: $f(-1) = -1$ and for all $v \geq 0$, $f(v) = v \, mod \, (2n-1)$. For a set S, define $f(S)$ as $\{f(v) \mid v \in S\}$.

To define the operators max and $\prec$ for B-Bakery, we consider two runs: a run R in which processes execute UB-Bakery algorithm and a run R' in which processes execute B-Bakery algorithm. It is assumed that the order in which processes take steps is the same in R and in R'. Our goal is to define max and $\prec$ for B-Bakery so that processes behave "analogously" in R and R'. Informally this means that the state of each P_i at any point in run R is the same as P_i's state at the corresponding point in R'; and the value of each shared variable at any point in R is congruent (modulo $2n-1$) to its value at the corresponding point in R'. We realize this goal as follows.

– Definition of max: Consider a process P_i that executes lines 2 and 3 in the run R of UB-Bakery algorithm. Let S_{ub} denote the set of token values that P_i reads on line 2 and x_{ub} denote the value of X that P_i reads on line 3. Let $max_{ub} = max\,(S_{ub} \cup \{x_{ub}\})$.
Now consider P_i performing the corresponding steps in run R' of B-Bakery. Let S_b denote the set of token values that P_i reads on line 2 and x_b denote the value of X that P_i reads on line 3.
If processes behaved analogously in runs R and R' so far, we would have $x_b = f(x_{ub})$ and $S_b = f(S_{ub})$. Further, since non-negative values in S_{ub} are in the interval $[x_{ub} - (n-1), x_{ub} + (n-1)]$ (by Theorem 2), if a and b are distinct values in S_{ub}, then $f(a)$ and $f(b)$ would be distinct values in S_b. We want to define the operator max so that $\mathsf{max}(S_b \cup \{x_b\})$ that P_i computes on line 4 is $f(max_{ub})$. This requirement is met by the definition of max described in the next paragraph. (Let $\oplus$ and $\ominus$ be defined as follows: $a \oplus b = (a+b)\,mod\,(2n-1)$ and $a \ominus b = (a-b)\,mod\,(2n-1)$.)
Non-negative values in S_{ub} lie in the interval $[x_{ub}-(n-1), x_{ub}+(n-1)]$. The elements in this interval are ordered naturally as $x_{ub}-(n-1) < x_{ub}-(n-2) < \ldots < x_{ub} < \ldots < x_{ub}+(n-2) < x_{ub}+(n-1)$. Correspondingly, non-negative values in S_b should be ordered according to $x_b \ominus (n-1) < x_b \ominus (n-2) < \ldots < x_b < \ldots < x_b \oplus (n-2) < x_b \oplus (n-1)$. Therefore, if we shift all non-negative values in $S_b \cup \{x_b\}$ by adding to them $n-1-x_b$ (modulo $2n-1$), then the smallest possible value $x_b \ominus (n-1)$ shifts to 0 and the largest possible value $x_b \oplus (n-1)$ shifts to $2n-2$. Then, we can take the ordinary maximum over the shifted values of $S_b \cup \{x_b\}$ and then shift that maximum back to its original value. More precisely, let max be the ordinary maximum over a set of integers. Let $T = S_b \cup \{x_b\} - \{-1\}$. Then, we define $\mathsf{max}(S_b \cup \{x_b\}) = max\,(\{v \oplus (n-1-x_b) \mid v \in T\}) \ominus (n-1-x_b)$. The following Lemma states the desired relationship between max and max that we have established.

Lemma 1. *Consider a run in which processes, including P_i, execute* UB-Bakery. *Let S_{ub} denote the set of token values that P_i reads on line 2 and x_{ub} denote the value of X that P_i reads on line 3. Let $S_b = f(S_{ub})$ and $x_b = f(x_{ub})$. Then $\mathsf{max}(S_b \cup \{x_b\}) = f(max\,(S_{ub} \cup \{x_{ub}\}))$.*

– Definition of $\prec$: Consider a process P_i executing line 7 in run R of UB-Bakery algorithm. Let v_j be the value that P_i reads in $token_j$ and v_i be the value of $token_i$.
Now consider the corresponding step of P_i in the run R' of B-Bakery. Let v'_j be the value that P_i reads in $token_j$ and v'_i be the value of $token_i$.
If processes behaved analogously in runs R and R' so far, we would have $v'_i = f(v_i)$ and $v'_j = f(v_j)$. By Theorem 3, if $v_j \neq -1$ then v_j is in the interval $[v_i-(n-1), v_i+(n-1)]$. We want to define $\prec$ so that $[v'_i, i] \prec [v'_j, j]$ holds if and only if $[v_i, i] < [v_j, j]$ holds. To achieve this, we proceed as before by shifting both v'_i and v'_j by adding $(n-1-v'_i)$ (modulo $2n-1$) and then comparing the shifted values using the ordinary "less than" relation for

integers. More precisely, $\prec$ is defined as follows: $[v'_i, i] \prec [v'_j, j]$ if and only if $[v'_i \oplus (n-1-v'_i), i] < [v'_j \oplus (n-1-v'_i), j]$. The following Lemma states the desired relationship between $\prec$ and $<$ that we have established.

Lemma 2. *Consider a run in which processes, including P_i, execute* UB-Bakery. *Let v_j be the value that P_i reads (on line 7) in $token_j$ and v_i be the value of $token_i$ when P_i is executing line 7. Let $v'_i = f(v_i)$ and $v'_j = f(v_j)$. Then $[v'_i, i] \prec [v'_j, j]$ holds if and only if $[v_i, i] < [v_j, j]$ holds.*

To summarize, B-Bakery, our final algorithm that uses bounded tokens, is identical to UB-Bakery (Figure 2) with the operators $+$, *max* and $<$ replaced with $\oplus$, max and $\prec$ as defined above. The following theorem states our result. Formal proof of this theorem is presented in the full version of this paper.

Theorem 4. *The algorithm* B-Bakery *satisfies the following properties:*

- **Mutual Exclusion:** *No two processes can be simultaneously in CS.*
- **Starvation Freedom:** *Each process that invokes* B-Bakery *eventually enters CS.*
- **Doorway FIFO :** *Let t be the time when P_i writes in $token_i$ when executing line 4. If P_j initiates an invocation after t, then P_i enters CS before P_j.*
- **Bounded Registers:** *Every shared register is either a boolean or has* $\log 2n$ *bits.*

References

1. Attiya, H., and Welch, J.: *Distributed Computing: Fundamentals, Simulations and Advanced Topics.* McGraw-Hill Publishing Company, May 1998.
2. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. Communications of the ACM **17,8** (August 1974) 453-455.
3. Lycklama, E., and Hadzilacos, V.: A first-come-first-served mutual-exclusion algorithm with small communication variables. ACM Transactions on Programming Languages and Systems **13,4** (October 1991) 558-576.
4. Lynch, N.: *Distributed Algorithms.* Morgan Kaufmann Publishers 1996.
5. Raynal, M.: *Algorithms for Mutual Exclusion.* The MIT Press 1986.
6. Silberschatz, A., Peterson, J., and Galvin, P.: *Operating System Concepts.* Addison-Wesley Publishing Company 1991.