Distributed Computing (1989) 3:61–72

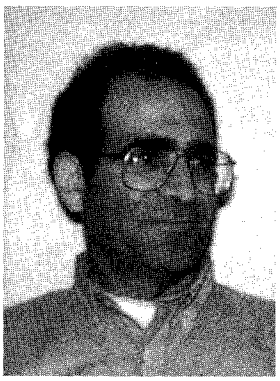
DISTRIBUTED COMPUTING

Choice coordination with limited failure

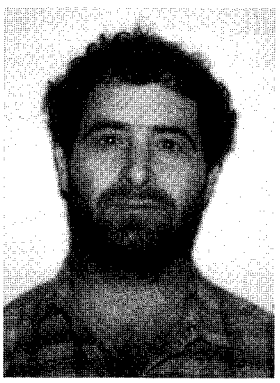
Amotz Bar-Noy[1], **Michael Ben-Or**[2], **and Danny Dolev**[2, 3]

[1] Department of Computer Science, Stanford University, Stanford, CA 94305-2140, USA
[2] Department of Computer Science, The Hebrew University, Jerusalem, Israel
[3] K 53 Almaden Research Center, IBM Research, 650 Harry Road, San José, CA 95120-6099, USA

Michael Ben-Or was born in Jerusalem, Israel, in 1955. He received the Ph.D. degree in mathematics from the Hebrew University, Jerusalem, Israel, in 1982. From 1982 to 1984 he was a C. Weizmann Postdoctoral Fellow at the Laboratory for Computer Science, Massachusetts Institute of Technology. In 1984 he was awarded the Israeli Government Alon Fellowship and he joined the faculty of the Hebrew University where he is currently a Senior Lecturer of Computer Science. His research interests include Algebraic and Combinatorial Complexity, Cryptography, and theoretical aspects of Parallel and Distributed Computations.



Danny Dolev received a B.Sc. in physics from the Hebrew University Jerusalem in 1971, an M.Sc. in applied mthematics from the Weizmann Institute of Science, Israel, in 1973, and a Ph.D. in computer science in 1979. After two years as a post-doctoral fellow at Stanford and a year as a visiting scientist at IBM, he joined the Hebrew University, Jerusalem in 1982, and IBM Almaden Research Center at 1987. His major research interests are distributed computing, reliability of distributed systems, and algorithms.



Amotz Bar-Noy received his B.Sc. degree in mathematics and computer science in 1981, and his Ph.D. degree in computer science in 1987, both from the Hebrew University, Jerusalem, Israel. Since 1987, he is a post-doctoral fellow in the Department of Computer Science at Stanford University. His current research interest is theoretical computer science, in particular, distributed and parallel computing.

Abstract. The Choice Coordination Problem requires n asynchronous processes to reach a common choice of one out of k possible alternatives. Processes communicate via k shared variables. Up to t, $t<n$, of the processes may fail to operate by suddenly quitting the protocol. Rabin (1982) presented lower and upper bounds for the extreme case $t=n-1$. We present deterministic and randomized algorithms for arbitrary t using an alphabet of size $O(t^2)$. A semi-synchronous model is also studied. A reduction to a consensus problem proves the necessity to assume some powerful atomic shared-memory operations.

Key words: Choice coordination – Probabilistic versus deterministic algorithms – Resilient algorithms

* *Offprint requests to:* D. Dolev, K 53 Almaden Research Center, IBM Research, 650 Harry Rd, San Jose, CA 95120-6099, USA

1 Introduction

The problem of breaking symmetry in a distributed system is very basic in distributed and parallel computation. Such problems arise when a set of processes try to distinguish some object in their environment. This paper deals with a typical symmetry-breaking problem called the *Choice Coordin-*

ation Problem (CCP), introduced by Rabin (1982) and described informally as follow:

Given a set of n asynchronous concurrent processes and k shared resources (or alternatives), spontaneously a process, or a set of processes, may start or join an effort to choose an alternative. The choice coordination problem, requires that the processes should choose one and only one of the k possible alternatives. Each process has its own system of names for the various alternatives; therefore the participating processes cannot agree beforehand on the chosen alternative and should communicate in order to agree upon one. Each alternative has an associated variable shared by all participating processes. Processes can communicate only via these k shared variables. The basic atomic step by a process is a *read-modify-write* operation, i.e., in a single indivisible step a process reads a shared variable, makes some internal computation and writes into it, without interference by any other process. As will be shown later, there is no solution for the CCP if *read* and *write* are the only atomic steps. Therefore, the assumption of a stronger atomic step is inevitable.

Two basic parameters measure the quality of a correct solution to the problem: first and most important is the size of the shared variables; i.e., the size of the communication alphabet; the other parameter is the size of the internal memory of each process required by the solution.

Apparently, most distributed algorithms are not resilient to faults, in the sense that unless all processes behave correctly, the algorithm fails. Recently the problem of overcoming faults has been extensively investigated. The goal of our research is to explore the trade-off between the complexity of solving the problem and the number of faults handled.

In the CCP, a *faulty* process is a process that stops functioning before an alternative is chosen or a process that does not start to operate at all. Rabin's solution and lower bound (Rabin 1982) deal with the extreme case where all processes but one may be faulty. In this paper a trade-off between m, the size of the communication alphabet and t, the upper bound on the number of possible faulty processes is studied.

Assuming that each process has a unique *ID* from the range $1, \ldots, n$, Rabin (1982) presented a deterministic algorithm for the case $t=n-1$ that requires $m=n+2$ for any number of alternatives. In the same paper he also proves a lower bound of $(1/2)\, n^{1/3} \leqq m$ for the size of the alphabet. The CCP is not trivial even under the assumption that there are no faulty processes. For this special case (i.e., $t=0$), Artstein (unpublished manuscript) observed that a constant alphabet is sufficient, and presented both probabilistic and deterministic solutions using constant and $O(\log n)$ bits of internal memory respectively. Fischer and Rabin (private communication) presented a deterministic algorithm for the case $t=n-1$ with $m=O(2^n)$ that can be generalized to arbitrary t with $m=O(2^t)$.

In this paper deterministic and probabilistic algorithms for an arbitrary t, $0 \leqq t \leqq n-1$, are presented. The alphabet in both algorithms is of size $m=O(t^2)$. The suggested deterministic algorithm requires $O(2^n \log t)$ bits of internal memory. Another deterministic solution which requires only $O(\log n)$ bits of internal memory, but alphabet size $m=O(t^2 \log n)$ is given. In all the above algorithms the size of m can be reduced by a factor of t, if we assume $2t+1<n$. For the special case where the number of alternatives is three we show that $m=n/2+3$ is sufficient for a deterministic algorithm where $t=(n-1)$. This last result can be generalized to every prime k, with $m=n/(k-1)+3$, under the assumption that processes agree on some cyclic order of the k alternatives.

A practical algorithm for solving the CCP problem, for every t, is suggested by Rabin (1982). That algorithm uses m symbols and terminates correctly with probability $1-\frac{1}{2^{m/2}}$ after at most $O(m)$ steps. Thus, there exists a small probability of an error. Our probabilistic algorithm terminates with probability 1 and always correctly. By combining both algorithms, one gets a correct and practical algorithm that terminates with probability 1.

Rabin's lower bound can be immediately extended to any t, obtaining $m=\Omega(t^{1/3})$. This lower bound does not seem to be tight, but in the sense of the number of bits required by the alphabet our algorithms use the same order of magnitude.

In Sect. 2 we give some preliminaries. In Sect. 3 the case $t=n-1$ is considered. We present a slightly simplified version of Rabin's algorithm that is easier to understand and prove. This section also contains the results for the special case $k=3$ and for k being prime. In Sect. 4 deterministic and probabilistic algorithms for the case $t=1$ are given. The main results for arbitrary t is presented in Sect. 5, for both deterministic and probabilistic cases. A constant size alphabet algorithm is shown in Sect. 6 where it is assumed that the processes are somewhat synchronous. In Sect. 7 we give two proofs to the impossibility result for the CCP in the case where read and write are the only atomic steps. One proof uses a reduction to a consensus problem and the impossibility result of Herlihy

(1988), and the other is an explicit construction of a contradicting run. In the last section we suggest an extension to the "Byzantine" case, and present some open problems.

2 Preliminaries

Denote by $\mathcal{P}=\{p_1, \ldots, p_n\}$ the set of processes that participate in solving the CCP problem, by $\mathcal{A}=\{A_1, \ldots, A_k\}$ the set of the alternatives and by $v_1, \ldots, v_k$ their associated shared variables. Recall that these notations are external and are not known to the processes, otherwise a trivial solution exists.

The asynchronous behavior of processes and the spontaneous nature of the choosing effort can be well described by an arbitrary *schedule* (an adversary) who chooses the next process to make a *step* (read-modify-write). This schedule is described by an infinite sequence of processes from $\mathcal{P}$; $J=p_{j_1}, p_{j_2}, \ldots$. If process p is the τ-th element in the sequence, then p *scans* (reads-modify-writes) one of the shared variables at time τ. The value of the variable, just before being scanned by p is the value *read* by p and the value left in the variable is the value *scanned* by p.

Among the n processes up to t processes might not start or might quit (*die*). The schedule can decide which processes participate, quit or do not start as long as at least $n-t$ processes appear infinitely often in the schedule.

The different local names that each process has for the k alternatives are described by a set of n permutations $\pi_1, \ldots, \pi_n$ of the numbers $1, \ldots, k$. For process p_i the local order of the k alternatives is $A_{\pi_i(1)}, \ldots, A_{\pi_i(k)}$. Process p_i makes its first step by scanning $v_{\pi_i(1)}$ and continues to scan the next variable according to its permutation.

Denote by Σ the communication alphabet, and let m be its size. We assume that Σ contains two special symbols, Φ – the initial value of each shared variable, and e – a value that indicates that the alternative associated with this shared variable was chosen.

A correct *t-resilient* deterministic solution to the problem is an algorithm in which, for any schedule and for any set of n permutations, e is written after a finite number of steps and is written in exactly one of the shared variables. In a correct t-resilient probabilistic solution it is required that e is written in exactly one of the shared variables with probability 1.

The inherent symmetry of the deterministic case implies that processes cannot be identical. Therefore we assume that each process has a unique ID in the range $1, \ldots, n$.

In the case $k=2$, denote by L and R the two alternatives, and by v_l, v_r the corresponding associated variables. The set of n permutations can be replaced by a partition function $\Pi: \mathcal{P} \longrightarrow \{L, R\}$ where $\Pi(p)$ indicates the first alternative process p scans.

In order to demonstrate our solutions it is enough to present them for the case $k=2$, the general solutions are along the same lines. Therefore, for clarity, all the above algorithms are described in detailed only for the case $k=2$. For an arbitrary k we present only the algorithms for $t=n-1$.

3 CCP with $t=n-1$

In this section a slightly different and somewhat easier to understand version of Rabin's (1982) deterministic algorithm is presented since later algorithms are given in a similar way and share some of its ideas. We first present the algorithm for the case $k=2$ with a full proof and then the algorithm for arbitrary k without a proof. At the end of this section we give an algorithm under the assumption that all the processes agree on the same cyclic order between the alternatives.

3.1 *The algorithm for $k=2$*

The idea behind the algorithm is as follows. Every process scans the two variables v_l and v_r in the order determined by Π. A process writes its ID at most once, and only if it reads Φ in its first step. The symmetry is broken since each process has a distinct ID. The first process that makes three steps is, therefore, able to write e at its third step in the variable whose value is smaller (Φ is interpreted as 0).

The alphabet used by this algorithm is $\Sigma=\{\Phi, e, 1, \ldots, n\}$. The local memory of each process consists of three variables ID, w and u. The formal description of the algorithm appears in Fig. 1.

```
Step 1: {move to A_Π(p); read(w)};
        if (w=e) then go to Halt;
        if (w=Φ) then {write(ID); u←ID}
                 else u←w;
Step 2: {move to the other alternative; read(w)};
        if (w=e) then go to Halt;
        if (w=Φ or u<w) then {write(e); go to Halt};
Step 3: move to the other alternative;
        write(e);
Halt:   stop.
```

Fig. 1. The algorithm for $t<n$ and $k=2$

Theorem 3.1. *The algorithm is $(n-1)$-resilient deterministic algorithm.*

Proof. The proof consists of two parts, first the termination, i.e., e will eventually be written, and second the validity, i.e., e will be written at most once. The termination is implied by the fact that every nonfaulty process reaches Step 3 of the algorithm. Therefore, e will be written by the time any process completes three steps. The validity follows from the next lemma. □

Lemma 3.1 (validity). *For every schedule, at most one variable will be set to e by the algorithm.*

Proof. Since every process has a distinct ID it follows that the only case where $v_l = v_r \neq e$ is at the beginning where both are Φ. If a process changes a value that is not Φ it changes it to be e. Therefore if $v_l \neq \Phi$ ($v_r \neq \Phi$) and $v_l < v_r$ ($v_r < v_l$) only $v_l(v_r)$ can be e. The remaining case is where $v_l(v_r)$ was Φ right before being e. In this case $v_r \neq \Phi$ ($v_l \neq \Phi$) and it can never be smaller than $v_l(v_r)$. □

3.2 The algorithm for arbitrary k

The idea of the algorithm for an arbitrary number of alternatives is very similar to that of the previous algorithm. Every process scans all the alternatives according to the permutation π it has for the cyclic order between the alternatives. It writes its ID at most once, only in case it reads Φ in its first step. It replaces every later appearance of Φ by -1. The symmetry is broken, since each process has a distinct ID, each ID is written only once, and at least one ID is written.

The alphabet for this algorithm is $\Sigma = \{\Phi, e, -1, 1, \ldots, n\}$, and the local memory of each process consists of $k+1$ variables: ID and $u_1, \ldots, u_k$. The formal description of the algorithm appears in Fig. 2 for a process whose cyclic order of the alternatives is defined by a permutation π.

Step 1:	{**move to** $A_{\pi(1)}$; **read** (u_1)};
	if $(u_1 = e)$ **then go to** Halt;
	if $(u_1 = \Phi)$ **then** {**write** (ID); $u_1 \leftarrow ID$};
Loop:	**for** $i \leftarrow 2$ **to** k **do**
	{**move to** $A_{\pi(i)}$; **read** (u_i)};
	if $(u_i = e)$ **then go to** Halt;
	if $(u_i = \Phi)$ **then** {**write** (-1); $u_i \leftarrow -1$};
	end;
Decision:	let j be such that $u_j = \max_{1 \leq i \leq k}\{u_i\}$;
	move to $A_{\pi(j)}$;
	write (e);
Halt:	**stop**.

Fig. 2. The algorithm $t < n$ and $2 \leq k$

3.3 Communication alphabet of size smaller than n

Rabin (1982) proved a lower bound on the size of the communication alphabet, he showed that $m \geq \frac{1}{2} n^{1/3}$. The above algorithms may suggest the natural conjecture $m \geq n$. To contradict this conjecture an algorithm for three alternatives with alphabet $\Sigma = \{\Phi, e, -1, 0, \ldots, \lfloor (n-1)/2 \rfloor\}$ is presented.

This algorithm is similar to the previous one with two modifications. The first one is at Step 1, where a process writes $\lfloor (ID-1)/2 \rfloor$ (instead of its ID) and sets u_1 accordingly. The second change is in the decision rule. Since it is guaranteed that each number in the range $[-1, \ldots, \lfloor (n-1)/2 \rfloor]$ is written at most twice there must be either a unique minimum or a unique maximum. Thus, the decision rule for a process is to write e in the variable whose value is this unique extreme.

Note that this algorithm can be generalized to any prime number k, with the additional assumption that all processes scan the alternatives in the same cyclic direction. In this case every process writes $\lfloor (ID-1)/(k-1) \rfloor$. The symmetry is broken by looking at the k cyclic permutations of the values. The fact that k is prime implies that there is a lexicographical maximal permutation unless all the values are equal. In our case at most $k-1$ processes have the same $\lfloor (ID-1)/(k-1) \rfloor$, hence at least two values are different. The size of the alphabet in the general case is $\lceil n/(k-1) \rceil + 3$.

4 CCP without dying

In the previous section the number of faulty processes allowed was $n-1$. Here we consider the other extreme where all the processes are nonfaulty. (Actually it is enough to assume that every process which starts the algorithm will not die before an alternative is chosen.) Both probabilistic and deterministic solutions are presented along the ideas of Artstein (unpublished manuscript).

The basic idea in both algorithms is to enable at most one process to "control" each alternative. Only that process may dynamically change the value of the variable associated with that alternative. Processes will compare bits of their ID's (or coin-flips) in a semi-synchronous manner by writing them into the shared variables. Since the ID's are distinct the symmetry will be broken in the deterministic case. In the probabilistic case it happens with probability 1.

Both algorithms, the deterministic and the probabilistic, are described together. The probabilistic action that substitutes a deterministic one is writ-

```
Step 1: {move to A_{Π(p)}; read (u_1)};
        if (u_1 = e) then go to Halt;
        if (u_1 ≠ Φ) then go to Passive;
        j ← 1;
        {u_1 ← ε_1 (or a coin-flip); write (u_1)};
Loop:   repeat
              {move to the other alternative; read (u_2)};
                (* scanning the second alternative *)
              if (u_2 = e) then go to Halt;
              if (u_2 = Φ or u_2 < u_1) then {write (e);
                go to Halt};
              {move to the other alternative; read (u_1)};
                (* scanning the first alternative *)
              if (u_1 = e) then go to Halt;
              if (u_1 < u_2) then {write (e); go to Halt};
              j ← j+1;
              if (u_1 ≠ −1) then u_1 ← −1
                            else u_1 ← ε_j (or a coin-flip);
              write (u_1);
        end;
Passive: repeat
              {move to the other alternative; read (w)};
        until (w = e);
Halt:   stop.
```

Fig. 3. The algorithm for $t=0$ and $k=2$

ten in parentheses. The alphabet used in the algorithms is $\Sigma=\{\Phi, 1, 0, -1, e\}$, and the internal memory of a process contains two variables u_1 and u_2. In the deterministic case the *ID* is represented by its binary expansion $\varepsilon_1, \ldots, \varepsilon_{\lceil \log n \rceil}$ and a pointer, j, to an arbitrary bit in this representation. The formal description of the algorithm is given in Fig. 3.

Theorem 4.1. *The algorithms are* 0*-resilient.*

Proof. The proof consists of two parts. As in Theorem 3.1, first is the termination, i.e., e will eventually be written, and second is the validity, i.e., e will be written at most once. The proof follows from Lemma 4.3 and Lemma 4.4 □

Since there are only two alternatives at most two processes can read Φ and spend time in the Loop part of the algorithm, we call these processes *active* processes. The loop consists of two parts each corresponds to another alternative. In the case that there are two active processes each one writes the bits from the binary representation of its *ID* (or the coin-flips) only in one of the shared variables. Denote the active process that writes its *ID* in L (respectively, R) by p_l (respectively, p_r).

Let $l_1, l_2, \ldots$ be the sequence of values written by p_l in v_l; and $r_1, r_2, \ldots$ be the similar sequence by p_r in v_r. Define $l_0=r_0=\Phi$. Since an active process writes -1 between the ε_i (or the coin-flips) it follows that both sequences are of the form $\Phi, x_1, -1, x_2, -1, \ldots$, where $x_j \in \{0, 1\}$.

Lemma 4.1 (semi-synchronization). *For every i, if v_l is l_i and $v_r \neq e$ (respectively, v_r is r_i and $v_l \neq e$), then v_r is one of the following: r_{i-1}, r_i, r_{i+1} (respectively, v_l is one of the following l_{i-1}, l_i, l_{i+1}).*

Proof. Let i be the minimal index for which the lemma does not hold. Without loss of generality assume that v_l is l_i and $v_r \neq e$, therefore v_r is r_{i+2}.
– Case $i=0$: before writing r_2 process p_r scans v_l and reads $l_0=\Phi$. However then it writes e in v_l and never returns to v_r; a contradiction.
– Case $i>0$: before process p_r writes r_{i+2} its u_2 and u_1 are l_i and r_{i+1}, respectively. Otherwise $i-1$ would be the minimal index for which the lemma does not hold. Yet $r_{i+1} \neq l_i$ since one of them equals -1 and the other is 1 or 0. It follows from the algorithm that r_{i+2} should be e if written at all, contradicting the assumption that $v_r \neq e$. □

Lemma 4.2 (symmetry breaking). *Suppose $v_l, v_r \in \{0, 1\}$ and $v_l \neq v_r$ at some step. Then the variable which is equal to 1 will not be changed at a later step, while the other one will become e at the next step of any active process scanning that variable.*

Proof. Lemma 4.1 implies that if $v_l, v_r \in \{0, 1\}$ then there exist i such that v_l is l_i and v_r is r_i. Without loss of generality assume that $v_l=1$ and $v_r=0$. In order to change v_l, p_l must see 1 in v_r but this can happen only if v_r is r_{i+2} or r_{i+2} which is impossible by Lemma 4.1. Hence, v_l will not be changed at a later step. From the same reason v_r will not be changed to -1 by p_r. So, either p_r or p_l will set v_r to e. □

Lemma 4.3 (termination). *For any schedule, the symbol e will eventually be the value of either v_l or v_r.*

Proof. Lemma 4.1 implies that for every i, if both l_i and r_i are not e, then there exists a time at which v_l is l_i and at the same time v_r is r_i.
– *The deterministic case.* The odd elements in both sequences are the binary representation of the corresponding *ID*s. Therefore there exists an odd index i for which $l_i=r_i$. By Lemma 4.2, the one that is equal to 0 will become e.
– *The probabilistic case.* The odd elements in both sequences are the results of coin-flips of both processes. Two independent fair coins have different values with probability $\frac{1}{2}$, therefore, with probabili-

ty 1 there exists i such that $l_i \neq r_i$. Again, by Lemma 4.2, e will be written. □

Lemma 4.4 (validity). *For every schedule, it is impossible that both variables are set to e during the run of the algorithm.*

Proof. Assume to the contrary that $l_x = e$ and $r_y = e$ for $x, y \geq 1$. In addition assume that p_l and p_r wrote e in v_l and v_r respectively (the case where p_l and p_r wrote e in v_r and v_l respectively is similar). By Lemma 4.1 $x = y + \delta$, where $\delta \in \{-2, -1, 0, 1, 2\}$.
- Case $x = y$ and x is odd: This implies that $r_{y-1} = l_{x-1} = -1$ and that process p_l scanned $r_{y-2} \in \{0, 1\}$ in v_r before writing e. That event happened after p_l wrote $l_{x-1} = -1$ in v_l. Therefore, process p_r scanned either $l_{x-1} = -1$ or $l_x = e$ after writing $r_{y-1} = -1$. In both cases process p_r would not have written e, contradicting the assumption.
- Case $x = y$ and x is even: Hence, $r_{y-1}, l_{x-1} \in \{0, 1\}$. Lemma 4.2 implies that $r_{y-1} = l_{x-1}$, otherwise the one that is equal to 1 would not be changed at all. Since $r_{y-2} = l_{x-2} = -1$, only l_{x-3} and r_{x-3} could cause them to write e. (If $x = 2$ then there are no l_{x-3} and r_{y-3} and they have no reason to write e). This contradicts Lemma 4.1, because r_{y-3} and l_{x-1} cannot be at the same time in v_r and v_l.
- Case $x = y + 1$ and x is odd (the case $y = x + 1$ is similar): Then the only reason for p_r to write e is if $r_{y-1} = 0$ and $l_{x-2} = 1$, but this contradicts Lemma 4.2 which states that l_{x-2} cannot be changed.
- Case $x = y + 1$ and x is even (the case $y = x + 1$ is similar): The same contradiction follows, because $l_{x-1} = 0$ and $r_{x-1} = r_y = 1$.
- Case $x = y + 2$ ($y = x + 2$ is similar): Then p_l wrote e because it scanned r_{y-1} and l_{x-1}, contradicting Lemma 4.1. □

Remark. The only case in which the above algorithms are not 1-resilient is when all processes make their first step at the same alternative, and the first one to write its bit in that alternative dies. The modification to the algorithms to handle this case is simply adding a special rule for the second step, to go to the Loop part in case a process reads Φ.

5 The general solution

The following algorithms solve the Choice Coordination Problem for any number t of faulty processes, t in the range $[0, \ldots, n-1]$, using alphabet size that is a function of t.

The algorithms are:
1. A probabilistic t-resilient algorithm with an alphabet of size $m = O(t^2)$ and an internal memory of size $O(\log t)$ bits.
2. A deterministic t-resilient algorithm with an alphabet of size $m = O(t^2)$ and an internal memory of size $O(2^n \log t)$ bits.
3. A deterministic t-resilient algorithm with an alphabet of size $m = O(t^2 \log n)$ and an internal memory of size $O(\log n)$ bits.

When $n \geq 2t + 2$ then the alphabet size of any one of the three algorithms can be reduced by a factor of t. All the above algorithms share the same initial part that is described in the next subsection as the initiation algorithm. The purpose of the initiation algorithm is to bound the number of processes which will actively take part in choosing the alternative by $t+2$ and by that decreasing the alphabet's size.

5.1 The initiation algorithm

If we could distribute the *ID*s $1, \ldots, t+1$ to $t+1$ processes and let them choose an alternative, Rabin's algorithm could be used. The following algorithm "entitles" at most $t+2$ processes as *Active* and the other as *Passive*. The Active processes continue in the effort to chose an alternative, whereas the Passive processes just wait until an alternative is chosen. For the second deterministic algorithm this algorithm provides every active process with a "new" *ID*, denoted by *NID*, in the range $[1, \ldots, \lfloor t/2 \rfloor + 1]$, where at most two processes share the same *NID*.

The alphabet used by the initiation algorithm is $\Sigma_1 = \{\Phi, 1, \ldots, \lfloor t/2 \rfloor, 0\}$, and the internal memory of every process consists of the variables *w*, *step* and *NID*. The variable $w \in \Sigma_1$ stands for values read in the shared variables, *step* distinguishes between the first and the second steps, and *NID* is a new number a process receives for the third deterministic algorithm.

The formal description of the initiation algorithm appears in Fig. 4 together with the Passive and the Halt parts of the main algorithm.

Observe that if $t+2$ nonfaulty processes take part in the algorithm, then it results with the value 0 in both alternatives. The initiation algorithm in this case requires the addition of only $O(t)$ symbols to the alphabet of the main algorithm. When we cannot be sure that at least $t+2$ nonfaulty processes will actually start the initiation algorithm the following modification can be used. Each variable is divided into two fields. The initiation algorithm will be run on one field and the rest of the

```
Step 1:  step←1;
         move to A_{Π(p)};
Check:   read(w);
         if (w=e) then go to Halt;
         if (w=Φ) then {write(1); NID←1; go to Active};
         if (1≦w<⌊t/2⌋ then {write(w+1); NID←w+1;
                             go to Active};
         if (w=⌊t/2⌋) then {write(0); NID←⌊t/2⌋;
                            go to Active};
         (* here w=0 or w∉Σ_1 *)
         if (step=1) then {step←2; move to the other
                           alternative; go to Check}
                     else go to Passive;
Active:  MAIN ALGORITHM;
Passive: repeat
           move to the other alternative;
           read(w);
         until (w=e);
Halt:    stop.
```

Fig. 4. The initiation algorithm

CCP algorithm on the other. This partition of variables implies multiplication of the alphabet used by the main algorithms by a factor of t. Thereafter the instruction "read" is used for both cases with the appropriate interpretation.

5.2 *The probabilistic algorithm*

The basic idea behind the probabilistic algorithm is quite simple. Had all processes started at the same alternative they could choose it, however this is not always the case. Our algorithm, in a sense, makes all the processes to "be" in the same alternative. During the algorithm every process chooses

```
Active:   (* at most t+2 processes reach this part
             of the algorithm *)
Step 1:   read(u_1);
          if (u_1=e) then go to Halt;
Step 2:   {move to the other alternative; read(u_2)};
          if (u_2=e) then go to Halt;
          if (u_2>0)
            then {(* zeroing both variables *)
                  write(0); move to the other alternative;
                            write(0);
                  if (coin-flip=1)
                    then move to the other alternative;
                    (* a new alternative is suggested *)
                  go to Step 1};
Step 3:   {move to the other alternative; read(u_3)};
          if (u_3=e) then go to Halt;
          if (u_1=u_3≦t) then {u_1←u_1+1; write(u_1);
                                          go to Step 2};
Decision: if (u_1=u_3=t+1) then {write(ε); go to Halt};
          (* here u_1≠u_3 *)
          {u_1←u_3; go to Step 2};
```

Fig. 5. The probabilistic algorithm, $0 \leqq t < n$ and $k=2$

at random an alternative it *suggests* to be the chosen one. When a process finds out that another process have suggested a different alternative it will flip a coin to resuggest an alternative. With probability one all processes will eventually suggest the same alternative and will choose it.

The alphabet for the probabilistic algorithm is $\Sigma_2 = \{0, 1, \ldots, t+1, e\}$. Every active process has a suggested alternative where the initial suggestion is the alternative it starts at (according to Π). A basic move of a process is three scanning steps: its suggested alternative, the other one and again it's suggested one. In this basic move a process reads into its three local variables u_1, u_2, u_3. By reading a value greater than zero in a second step of a basic move a process realizes that not everyone suggests the same alternative. Therefore it writes 0 in both alternatives and flips a coin to determine its new suggestion. If it reads zero in the second step and it reads the same value in its first and third steps then a process increases this value by 1. The decision is made when this identical value is $t+1$.

Lemma 5.1. *Let x and y be the values of v_l and v_r respectively at some point during the probabilistic algorithm. Then at least $1+\min(x, y)$ active processes have taken part in it.*

Intuitively, assuming $x \leqq y$, the worst case happens when one process wrote $1, \ldots, x$ in one alternative and every $1 \leqq j \leqq y$ was written in the other alternative by a different process.

Proof. The lemma can be stated as follow: Let $v_l = x \geqq 0$ and $v_r = y \geqq 0$ be the values of the alternatives at some time, (it is denoted by (x, y)). Let $(0, 0) = (x_0, y_0), \ldots, (x_{x+y}, y_{x+y}) = (x, y)$ be the intermediate values of v_l and v_r. Assume that for $0 \leqq i \leqq x$, p_i wrote i in v_l and that for $0 \leqq j \leqq y$, q_j wrote j in v_r and denote $P = \{p_1, \ldots, p_r\}$ and $Q = \{q_1, \ldots, q_y\}$. Then:

$$|P \cup Q| \geqq 1 + \min(x, y).$$

If all the p_i's are different or all the q_j's are different then the proof is complete. Otherwise there exist $1 \leqq i < i' \leqq x$ and $1 \leqq j < j' \leqq y$ such that $p_i = p_{i'}$ and $q_j = q_{j'}$.

For every process p_i the last three steps before writing i in v_l are as follows. In the first step it scanned $i-1$ in v_l, in the second step it scanned 0 in v_r and in the third step it wrote i in v_l after reading there $i-1$. Every q_j made also such three steps. Denote by $x_i^1 < x_i^2 < x_i^3$ (respectively, y_j^1

$<y_j^2<y_j^3$) the times when p_i (respectively, q_j) made these three steps.

For every $1\leqq i\leqq x$ and for every $1\leqq j\leqq y$ the following inequalities hold,

$$x_i^2<y_j^3; \quad y_j^2<x_i^3. \tag{1}$$

Otherwise, p_i (respectively, q_j) would have scanned j (respectively, i) in v_r (respectively, v_l) and not 0 as needed.

Inequalities (1) imply that:

$$y_{j'}^2<x_i^3. \tag{2}$$

The assumption that $p_i=p_{i'}$ and $q_j=q_{j'}$ implies that:

$$x_i^3<x_{i'}^2, \tag{3}$$

and that:

$$y_j^3<y_{j'}^2. \tag{4}$$

Using the three inequalities (2), (3) and (4) we get,

$$y_j^3<x_{i'}^2 \tag{5}$$

which contradicts one of the inequalities (1). □

Theorem 5.1. *There is a probabilistic t-resilient algorithm for the CCP with alphabet of size $O(t^2)$.*

Proof. A simple probabilistic reasoning proves the termination of the algorithm. The validity of the algorithm follows from Lemma 5.1 by identifying e as $t+2$ and by using the fact that the number of active processes is bounded by $t+2$. □

5.3 The first deterministic algorithm

The deterministic algorithm is almost identical to the probabilistic one. The only difference is the rule when to suggest a new alternative. Instead of flipping coins each process switches suggested alternatives as a function of t, its ID, and the number of steps it has made so far. The validity proof is identical to the probabilistic one. The termination follows from the way we choose the function to switch suggested alternatives.

The next lemma describes the basic property of the algorithm and provides us the idea behind the definition of the function.

Lemma 5.2. *If a process makes $2t+8$ consecutive steps during which no other process suggests a new alternative e will be written.*

```
Active:  c←1;
Step 1: ...
Step 2: ...
        if (u2>0) then {write (0);
                        move to the other alternative;
                        write (0);
                        if (c=fT(ID)) then
                        {move to the other alternative;
                        c←0}
                        go to Step 1};
Step 3: ...
(* at every step the following instruction is done: *)
c←c+1;
```

Fig. 6. The first deterministic algorithm, $0\leqq t<n$ and $k=2$

Proof. This process needs three steps to identify that both variables are not equal to 0 and to zero them, and $2t+5$ more steps to write e in its suggested alternative. □

For every t define $T=2t+8$, and let $f_T(i):=(2T)^{2^i}$. A process suggests the alternative at which it starts (according to Π) and changes its suggested alternative at the end of the first basic move after it made $f_T(ID)$ steps. We add to the description of the probabilistic algorithm (Fig. 5) the new instructions which appears in Fig. 6.

Lemma 5.3. *Let p be an active process which has just suggested a new alternative. If during $f_T(ID(p))$ consecutive steps that p takes, only processes with smaller ID's suggest an alternative different than p's, then e is written before p suggests a different alternative the next time.*

Proof. The proof is by induction on the number of active processes with ID's smaller than p's one. When p has the smallest ID, then by Lemma 5.2 e will be written because for every ID, $f_T(ID)>T$. Otherwise, let q be the active process having the largest ID smaller than that of p.

If p will make $T'=f_T(ID(q))$ steps while q will make no steps then by the inductive assumption for p, e will be written. Therefore, p can make at most $(T')^2$ steps until q suggests the same alternative. If p will take another $(T')^2$ steps before changing its suggesting alternative the induction assumption can be applied on q. The function f_T implies that if $ID(p)>ID(q)$ then $f_T(ID(p))\geqq 2(f_T(ID(q)))^2$, which completes the proof. □

Let process p be the one with the largest ID among all the *Active* processes. Lemma 5.3 implies that e is written before p changes its first suggested alternative. Thus we proved the following theorem.

Theorem 5.2. *There is a deterministic t-resilient algorithm for the CCP with alphabet of size* $O(t^2)$. □

Note that the size of the internal memory is determined by f_T; and therefore is of size $O(2^n \log t)$ bits.

5.4 The second deterministic algorithm

The following algorithm reduces the size of the internal memory at the expense of requiring a larger alphabet. The algorithm itself is a generalization of the algorithm of Sect. 4. It is composed of three parts: first comes the common initial algorithm; in the second there are $\lfloor t/2 \rfloor + 1$ pairwise competitions among the active processes of the first part who received the same *NID*; and the third is essentially the algorithm of Sect. 3.

The alphabet contains quadruples

$$\Sigma = \{2, 3\} \times \{1, \ldots, t/2+1\} \times \{0, 1\} \times \{0, \ldots, \log n\}.$$

Denote all quadruples with first character i as Σ_i. The subalphabet Σ_i is used in part i of the algorithm. The second item in each quadruple is the *NID* an active process has received in the initiation algorithm. The last two items are a bit from an *ID*'s binary representation, and the bit's location in the expansion, respectively. The role of the first item is to distinguish between the concurrent competitions. The forth item serves as a "semi-synchronizer" for each competition, we cannot use the -1 again because of the concurrent manner the competitions held.

By similar reasoning, to that of the algorithm in Sect. 4, the symmetry is broken. When a process identifies that this happens it picks the quadruple it reads as its *ID*, for the third part of the algorithm. It only changes the first item from 2 to 3. The validity and the termination are guaranteed by similar arguments to those that prove Rabin's algorithm.

In its local memory every active process maintains the following parameters. The number it received during the initiation algorithm (*NID*), its *ID*'s binary representation $(\varepsilon_1, \ldots, \varepsilon_{\lceil \log n \rceil})$, an index denoting the ε it should write next (j) and three variables $u_1, u_2, u_3 \in \Sigma$. If u is a quadruple then denote by $u(j)$, $1 \leq j \leq 4$, the j'th item in the quadruple u. The formal description of this algorithm appears in Fig. 7.

```
Active:   (* at most t+2 processes reach this part of
             the algorithm *)
          j ← 1;
Step 1:   read (u1);
          if (u1 = e) then go to Halt;
          if (u1(1) = 3) then go to Decision;
Step 2:   {move to the other alternative; read (u2)};
          if (u2 = e) then go to Halt;
          if (u2(1) = 3) then {u1 ← u2; go to Decision};
Step 3:   {move to the other alternative; read (u3)};
          if (u3 = e) then go to Halt;
          if (u3(1) = 3) then {u1 ← u3; go to Decision};
          (* here u1, u2, u3 ∉ Σ3 *)
          if (u1 = u2 = u3)
            then {u1(2) ← NID; u1(3) ← εj; u1(4) ← j
                  j ← j+1;
                  write (u1); go to Step 2};
          if (u1 = u3 ≠ u2) then {u1(1) ← 3; write (u1);
                                  go to Decision};
          (* here u1 ≠ u3 *)
          {u1 ← u3; go to Step 2};
Decision: (* here u1 ∈ Σ3 *)
          {move to the other alternative; read (u2)};
          if (u2 = e)  then go to Halt;
          if (u2 < u1) then {write (e); go to Halt}
                       else {move to the other alternative;
                             write (e); go to Halt};
```

Fig. 7. The second deterministic algorithm, $0 \leq t < n$ and $k = 2$

6 The semi-synchronous model

The CCP problem is defined for completely asynchronous processes. For better understanding the inherent complexity of the problem and the possible trade-offs, we study the question in a stronger model. In this section we show that if processes are not completely asynchronous, then there exists an $(n-1)$-resilient algorithm that uses five symbols for both the deterministic and the probabilistic cases.

Definition. A schedule J is Δ-synchronous if every process q that does not appear in a subsequence of J containing $\Delta + 1$ appearances of some process p, does not appear later in J (i.e., q died before the beginning of this subsequence).

Note that a 1-synchronous schedule is just a cyclic queue, where a ∞-synchronous schedule describes a completely asynchronous processes, as in previous sections. The algorithms in this section are for a Δ-synchronous schedules where $1 \leq \Delta < \infty$.

6.1 The probabilistic algorithm

The idea of the probabilistic algorithm is as follow. The processes write the results of coin-flips in the two shared variables until the symmetry is broken

```
procedure READ (u, w, C);
begin
    c←0;
    while (c≦C) do
        {move to the other alternative; read (u')};
          (* scanning the first alternative *)
        if (u'=e) then go to Halt;
        {move to the other alternative; read (w')};
          (* scanning the second alternative *)
        if (w'=e) then go to Halt;
        (* checking if the same pattern is found *)
        if (u'≠u or w'≠w) then {c←0; u←u'; w←w';}
                          else c←c+1
    end
end
```

Fig. 8. The procedure READ (u, w, x)

by having one variable equal to 1 and the other equal to 0. Between every two successive writing a process scans the two variables $\Delta+1$ times. Thus every other nonfaulty process makes at least two steps and therefore reads the contents of both shared variables before they are changed. By using the symbol -1 the processes succeed to achieve the semi-synchronization (as in Sect. 4).

The algorithm uses as a subroutine the procedure "READ (u, w, C)". In this procedure the process scans L and R until it reads the pattern (u, w) C times. During this time it writes nothing and when scanning e it jumps to the Halt position of the main program. This procedure appears in Fig. 8. The parameters u and w and the local variables u' and w' take their values from the alphabet of the main algorithm and the local counter c depends on the bound C sent by the main algorithm.

For the algorithm itself the alphabet is $\Sigma=\{\Phi, e, 1, 0, -1\}$ and the internal memory of a process consists of two variables u_1, u_2 containing the values of v_l and v_r (not necessarily in that order). We present the formal description of the algorithm, Fig. 9, without proving it.

6.2 *The deterministic algorithm*

The algorithm is essentially the deterministic algorithm of Sect. 4. For active processes the algorithm is identical. The only difference is new rules for non-active processes. The rules for active processes in the algorithm of Sect. 4 implies that after $4\log n+1$ steps of one of the active processes e must be written in one of the variables. Therefore the Δ-synchronous schedule enables a process to identify the death of both active processes, by observing that e is not written in any variable after making $x=(\Delta+1)(4\log n+1)$ steps. When a process detects that the active processes died it restarts the algorithm by writing Φ in both variables. The new parts of the algorithm of Sect. 4 are described in Fig. 10. Again we use the subroutine READ (u, w, C) described in Fig. 8. The first usage is with $C=\Delta+1$, in order to assure a process that before changing a value of a shared variables all the other read this value. The second usage is with $C=x$ to enable a process to detect the death of the active processes.

```
Step 1: {move to A_Π(p); read (u_1)};
        if (u_1=e) then go to Halt;
        if (u_1=Φ) then {u_1←coin-flip; write (u_1)}
Step 2: {move to the other alternative; read (u_2)};
        if (u_2=e) then go to Halt;
        if (u_2=Φ) then {write (e); go to Halt}
Loop:   repeat
            READ (u_1, u_2, Δ+1);
            if (u_1<u_2) then {write (e); go to Halt};
            if (u_2<u_1) then {move to the other alternative;
                               write (e); go to Halt};
            (* here, u_2=u_1 *)
            if (u_1≠−1) then u_1←−1
                        else u_1←coin-flip;
            write (u_1);
            {move to the other alternative; read (u_2)};
            if (u_2=e) then go to Halt;
        end;
Halt:   stop.
```

Fig. 9. The probabilistic algorithm for $t=0$, $k=2$ and $\Delta<\infty$

```
Step 1: {move to A_Π(p); u_1←u_2←Φ;}
        READ (u_1, u_2, Δ+1);
        if (u_1≠Φ or u_2≠Φ) then go to Passive;
Active: (* here, u_2=u_1=Φ *)
        {j←1; u_1←ε_1; write (u_1)};
Loop:   (* the same as the algorithm of Sect. 4 *)
Passive: repeat
            READ (u_1, u_2, x);
            if (u_1<u_2) then {write (e); go to Halt};
            if (u_2<u_1) then {move to the other alternative;
                               write (e); go to Halt};
            (* here, u_2=u_1 *)
            {write (Φ); move to the other alternative;
                        write (Φ)};
            go to Step 1;
        end;
Halt:   stop.
```

Fig. 10. The deterministic algorithm for $t=0$, $k=2$ and $\Delta<\infty$

7 Atomic memory operations are necessary

In this section we prove that atomic read and atomic write operations are not sufficient to solve the CCP. We present two proofs. The first, which

does not explicitly expose the difficulties, is a reduction to a consensus problem and then the impossibility is concluded from Herlihy's results (1988). The second proof is an explicit construction of a run that forces e to be written in both alternatives.

In the *consensus problem*, each processes has an input value (assume a binary one) and it is required that all participating processes (that do not stop functioning) agree on the same output. To avoid triviality, there should be a run in which the agreement is on 0, and a run in which the agreement is on 1. A solution to the consensus problem is called *wait-free*, if every process that starts running the algorithm, can complete it independent of any other process being correct or performing any operation.

Theorem 7.1. *Let A be an algorithm solving the CCP with $n=t-1$; then there exists an algorithm to solve the consensus problem. Moreover, the solution is wait-free.*

Proof. Denote the alternatives L and R by A_0 and A_1 respectively. To solve the above consensus problem, we use the partition function to distinguish between processes with input value 0 and those with 1. This is done by defining $\Pi(p)=A_i$ for a process p with input value i. Later on, p runs the CCP protocol A beginning at $\Pi(p)$, as defined above. A process p decides on its input value if e is written in $\Pi(p)$, otherwise p decides on the opposite value.

It is easy to verify that this algorithm solves the consensus problem, because all agree on the same alternative, and therefore decide on the same value. Suppose the decision is i when all have 1 as an input value then the symmetry of the CCP implies that the decision is $1-i$ when all have 0 as an input value. Thus, the requirement of different runs leading to different outputs is met.

The assumption $n=t-1$ implies that every process that starts running the algorithm, completes it without waiting for any other process to perform any action, since all the rest may stop functioning. Thus, the above solution is also wait-free. □

The following result, proved by Herlihy, (1988) implies that any solution to the CCP should use strong memory operation. A similar result can be proved for other memory operations, like *test-and-set*.

Theorem 7.2. *There is no solution to the above consensus problem using atomic read and atomic write operation.*

The requirement $n=t-1$ is not essential, but to improve it, it is required to modify the results of Herlihy and to define a stronger kind of consensus that is out of the scope of this paper. The explicit proof given below is for the case $t=n-1$, but can be extended for $t>n/2$.

Theorem 7.3. *There is no solution for the CCP if the atomic steps are only read operation and write operation.*

Proof. The proof is for the case where $n=3$, $t=2$, and $k=2$ and it uses the terminology from Sect. 2. We show that for every algorithm there exists a schedule and a partition function in which e is written in both alternatives.

Assume to the contrary that A is an algorithm that solves the CCP, where read operation and write operation are the only atomic steps. Let $\mathcal{P}=\{p_1, p_2, p_3\}$ and denote by p^i a sequence of i appearances of p. Look at the following three cases:

1. Run A with the schedule $J_1=p_1^\infty$ and the partition function Π_1, where $\Pi_1(p_1)=L$. Since $t=2$, p_1 must decide after a finite number of steps. Define this number to be x_1 and assume that at the x_1'th step p_1 writes e in $D_1\in\{L, R\}$.
2. Run A with the schedule $J_2=p_2^\infty$ and the partition function Π_2, where $\Pi_2(p_2)=L$. Since $t=2$, p_2 must make a write operation after a finite number of steps. Assume that at its x_2'th step p_2 performs its first write operation, and let it be in $D_2\in\{L, R\}$.
3. In a similar way to the previous case, define J_3, Π_3, x_3, and D_3 for p_3.

The symmetry of the CCP implies that in each one of the three cases above, if $\Pi_i(p_i)$ is changed to R, then D_i will be changed to be the opposite alternative. For the rest of the proof call this property: Property SYM.

Consider the following partition function Π_4, in which processes p_2 and p_3 perform their first write operations at different alternatives. If $D_2\neq D_3$ then $\Pi_4(p_2)=L$ and $\Pi_4(p_3)=L$, otherwise $\Pi_4(p_2)=L$ and $\Pi_4(p_2)=R$. Run algorithm A with the schedule $J_4=p_2^{x_2-1}p_3^{x_3-1}(p_2\ p_3)^\infty$ and the partition function Π_4. Since p_2 does not write before the first step of p_3 and p_3 does not write before the first write operation of p_2, it follows that for both p_2 and p_3 the initial run of J_4 looks like the initial runs of J_2 and J_3 respectively. The definition of Π_4 and Property SYM imply that in this run, p_2 and p_3 make their first write operation in different alternatives. Again, since $t>1$, p_2 and p_3 must decide. Define $J_5=p_2^{x_2-1}p_3^{x_3-1}p_2\,p_3\,J'$ to be a finite prefix of J_4 in which e is written in exactly one

of the alternatives. Let $D_5 \in \{L, R\}$ be that alternative.

Now we define the run that leads to the desired contradiction. The schedule for this run is $J = p_2^{x_2-1} p_3^{x_3-1} p_1^{x_1-1} p_2 p_3 J' p_1$. The partition function is Π, where $\Pi(p_2) = \Pi_4(p_2)$ and $\Pi(p_3) = \Pi_4(p_3)$, and if $D_1 \neq D_5$ then $\Pi(p_1) = L$, otherwise $\Pi(p_1) = R$.

For p_2 and p_3 this run is identical to the run using J_5 and Π_5, because after their first write operation they will erase anything p_1 wrote in both alternatives. This is the crucial point of the proof: while they write, they cannot read and thus, they cannot know about p_1. Furthermore, for p_1 this run is identical to that of Case 1, since p_2 and p_3 do not write before and during its $x_1 - 1$ steps. Before the last operation of p_1 e is written in D_5 and afterwards e is also written in D_1. The choice of Π and Property SYM assure that $D_1 \neq D_5$, and hence, contradiction. □

The proof above explicitly exposes the need for a more powerful atomic operations. The drawback of this explicit proof is its limited scope. It cannot be extended to other atomic operation like test-and-set (as the previous proof).

8 Open questions

In this section we suggest some open questions and an extension to Byzantine behavior of the processes.

Tight Bounds. We first note that our work leaves a gap between the upper bound ($O(t^2)$) and the lower bound ($\Omega(t^{1/3})$) on the size of the alphabet needed for CCP with t faults. In particular – is the right complexity $\Theta(t)$?

Memory – Alphabet Trade-off. The results of the deterministic algorithms in Sect. 5 give an interesting trade-off between the size of the communication alphabet, m, and the size of the internal memory of the individual processes. Decreasing m from $O(t^2 \log n)$ to $O(t^2)$ forced us to increase the size of the internal memory by a large factor. We leave the investigation of this interesting trade-off to further work.

A Conjecture. Throughout our paper, we presented a probabilistic algorithm, and along its lines, a corresponding deterministic one. We note that the validity proofs of both cases were the same. The main difference is the cause of termination. Probabilistic algorithms break the symmetry by coin flips while their deterministic counterpart use the processes different ID's. We managed to translate our probabilistic solutions to deterministic ones without increasing the communication alphabet. We conjecture that any probabilistic algorithm for the CCP that terminates with probability 1, against any adversary, can be translated to a deterministic one without increasing the alphabet size by more than a constant. This conjecture stands in sharp contrast to other examples in asynchronous problems where probabilistic algorithms are strictly stronger than deterministic algorithms (see Fischer et al. 1985 versus Ben-Or 1983).

Byzantine Model. It is easy to see that CCP has no solution if faulty processes can behave in a Byzantine manner. We suggest a different model: Divide each shared variable to several fields, and in an atomic step, a process can read the whole variable but can write in at most one field. A process can write in different fields on different accesses to the variable.

Denote by f the number of fields, then for any Δ-synchronous schedule there is the obvious lower bound of $\Delta \cdot (t/k) \leq f$. In particular, no completely asynchronous solution exists in this model. For the special case of $\Delta = 1$ and $k = 2$, we have a solution with $f = O(t)$ fields. This solution is probabilistic and uses the majority ideas of Ben-Or (1983). Further research in this direction seems promising.

References

Ben-Or, M (1983) Another Advantage of Free Choice. Proc 2nd ACM Symp of Principles of Distributed Computing, 1983

Fischer, MJ, Lynch, NA, Paterson, MS (1985) Impossibility of Distributed Consensus with One Faulty Process. J ACM 32:374–382

Herlihy, MP (1988) Impossibility and Universality Results for Wait-Free Synchronization. Proc 7th ACM Symp of Principles of Distributed Computing, 1988, pp 276–290

Rabin, MO (1982) The Choice Coordination Problem. Acta Inf 17:121–134