自己安定深さ優先トークン伝達方式における故障特権の回避について

木庭 淳
神戸商科大学管理科学科
神戸市西区学園西町 8-2-1
078-794-6161, kiniwa@kobeuc.ac.jp

Abstract

本稿では一般のネットワーク上で深さ優先でトークンを伝達するような自己安定プロトコルにおいて、故障特権を避ける方法について提案する。具体的実現方法としては副トークンと大きな状態空間をもつ状態変数を使用することにより、確率的にほとんどすべての故障特権を防ぐことができる。シミュレーションでは既に提案されている深さ優先トークン伝達方式との比較を行い、ネットワークトポロジと安定化時間の関係、故障数の影響などについて調べた。

自己安定プロトコル、深さ優先トークン伝達、故障特権、大きな状態空間、シミュレーション

Avoiding Faulty Privileges in Self-stabilizing Depth-first Token Passing

Jun Kiniwa
Dept. Management Science, Kobe Univ. of Commerce
8-2-1, Gakuennishi-machi, Nishi-ku, Kobe-shi, Japan
078-794-6161, kiniwa@kobeuc.ac.jp

Abstract

This paper presents a new algorithm and its experimental results for self-stabilizing depth-first token circulation in arbitrary networks. Our algorithm has an additional property of avoiding faulty privileges. Using an auxiliary token and a state with a large state space, almost all faulty privileges can be prevented. Simulation experiments show its performance compared with the previously proposed algorithm. Detailed results on the relation between network topologies and stabilization time, the effects of number of faults are revealed.

self-stabilizing protocol, depth-first token passing, faulty privilege, large state space, simulation

1 Introduction

If some erroneous values occur in tables or bits and programs remain safe, it is called a transient fault. After such a fault, self-stabilizing algorithms try to communicate with neighboring processes and the system eventually converges to some legitimate states. The algorithms just guarantee convergence to a legitimate state and preserving correctness after that. A lot of efforts have been made to get self-stabilizing algorithms with a small number of states. This is because fast convergence can be achieved by small state space. However, it also includes a risk, that is, vulnerability under convergence.

We have already pointed out the problem of "hidden" faulty privileges with small state space [9]. That is, if the state space is too small, the process cannot distinguish it has a true privilege or a faulty privilege. To cope with this problem, we have presented the method of a large state space, which increases the state space and uses some predefined values in legitimate execution. Then we can obtain a local reset algorithm for self-stabilizing mutual exclusion rings.

Our next interest is how our basic strategy of avoiding faulty privileges can be applied to general token passing systems. In arbitrary networks, Huang and Chen have developed a depth-first token circulation algorithm [6], and have later modified it to a uniform one [7]. Our application to general networks is based on their algorithm. Though our method has a wide application to any self-stabilizing problem, its practical implementation in general networks is not straightforward. First, a pointer corruption forces the system to a confusion because a process is connected with several processes unlike rings. Next, the token/signal passing in a depth-first manner has a difficulty in distinguishing whether the token/signal is going forward or backtracking. This is caused by that a token goes out or backtracks through the same process.

Though a lot of self-stabilizing algorithms have been developed, their performance are not known well. In analytical studies, we can evaluate only stabilization time, fault-containment, or some impossibility properties. It is difficult to analyze other behavior due to its distributed control. So it is worth while to examine their behavior by simulation experiments. We are interested in the relation between network topologies and stabilization time, the effect of number of faults, and so forth.

This paper has two major contributions. First, based on the Huang and Chen's method [6], we have developed a depth-first algorithm with avoiding faulty privileges under convergence. Second, we have implemented their and our algorithms into simulation programs using random graphs. Then we investigate their performance with several parameters.

Related Work

Recent papers have tackled the problem of vulnerability under convergence; that is, called fault containment[2, 5], time-adaptive stabilization[10], superstabiliztion[4], or coding systems[5, 12]. The concept of fault containment or time-adaptiveness is to prevent the diffusion of faulty states. This can be achieved by examining neighboring guards, priority scheduling[3], or replica with local voting[10]. Herman and Pemmaraju[5] generalized the use of error detection codes to detect faults. Their idea of increasing the state space is similar to ours. Depth-first token passing in the context of self-stabilization was presented in [6–8]. The methods of Huang and Chen[6] and Johnen,et.al. [8] use a similar idea. There are not so many experimental results in this area to the best of our knowledge. Chang,et.al[1] analyzed an averaged behavior of the Dijkstra's mutual exclusion ring with some simulation results. Masuda,et.al[11] modeled a self-stabilizing leader election algorithm as a Markov chain and evaluated its stabilization time by the Gauss-Jordan method.

2 Model and Basis

First we describe the self-stabilizing model used in our discussion. A network consists of n processes $P = \{p_0, p_1, \ldots, p_{n-1}\}$ of finite state machines connected arbitrarily. The *state model* is assumed, that is, the process p_i can directly read the adjacent states and similarly for auxiliary states. Each process has a program of the internal computation, **if** G **then** A, where G is called a *guard* and A an *action*. If G is true in a process, the process is said to be *enabled*. An *atomic step* consists of reading the neighbor's state, an internal computation, and writing its own state. The global state of all processes is described by its *configuration*, n-dimensional vector of states. For any configuration there must be some process to be enabled (*deadlock-free*).

In the mutual exclusion problem, at most one *privileged* process must be required at any time. In self-stabilizing algorithms, such configurations, called *legitimate*, are contained in every execution (*convergence*). Additionally, once legitimacy is restored, algorithms must keep the property (*closure*). A transient fault may generate erroneous states, where the number of enabled processes may be more than one. If there are several enabled processes, only one process is *activated* at a time by *C-daemon*. The daemon is assumed to be *fair*, that is, every process is activated infinitely often.

Huang and Chen's Method

In Huang and Chen's algorithm [6], called *Huang&Chen* hereafter, each process p_i has an integer variables $C.i \in \{0,1\}$ called `color`, $D.i \in P$ called `descendant` of p_i, and $L.i \in \{0,\dots,|P|-1\}$ called `level`. There are two kinds of programs, for the root and for others. The mutual exclusion condition is guaranteed by the unique path, called *segment*, of the `descendant` pointers originated from the root. The front end process of the segment has a token, meaning the privilege is given. The token is circulated in a depth-first manner, where `color` is used to avoid visiting some process twice in the same round. That is, the new adjacent process with a different color is searched by the front process. The variable `level` is used for eliminating cycles or illegal segments. Since the relation $L.D.i = L.i+1$ does not hold in cycles, the process p_i can find the inconsistency. Then p_i sets the descendant pointer at *Null*, making $D.i$ an illegal root. The illegal root propagates the color *Error* downward, and then sets *Null* upward. The isolated *Error* process changes its color to 0.

3 Algorithm

3.1 Overview of Our Method

We present an algorithm named *AFPG* (Avoiding Faulty Privileges in General networks), where each process has a `color` and a `descendant`, and also has an auxiliary real variable $Safe.i \in [0,1]$. Our *AFPG* also guarantees exactly one token by the segment originated from the root, where the token is circulated in a depth-first manner. The basic idea of a large state space is applied as follows. According to *Huang&Chen*[6], the key variable by which faults can be detected is not `color` but `descendant`. So we increase the domain space of `descendant` to $K \cdot deg(p_r)+1$ states for the root p_r and $K\cdot(deg(p_i)-1)+1$ states for other process p_i, where $deg(p_i)$ denotes the degree of a process p_i. Let K be a sufficiently large constant and let $j\cdot K$, called a *base* for an integer $j \in B_i = \{1,2,\dots,deg(p_i)-1\}$ ($B_r = \{1,2,\dots,deg(p_r)\}$), indicate a *child* (more distant from the root than p_i) process of p_i. If $j=0$ (corresponding to one addition), it is called *Null*. Since each process other than the root has a parent if it has a token, one is subtracted from its degree. The correspondence between the value and the process can be defined arbitrarily, e.g., in a clockwise order. Then if `descendant` takes on non-base value, the process can detect the fault by itself. Note that any non-base value can be considered as a corrupted one, and any base value as a legitimate one in almost every case.

When an activated process finds its descendant has some non-base state, it copies the non-base value. Since such an operation is propagated upward (to the root), the segment is broken at the time. Then isolated states are reset to *Null* one after another. This amount of stabilization time is reduced from [6] because their algorithm first propagates *Error* color downward. If there is a process other than the root which has no parent but has a descendant, called a *illegal root*, it is colored by *Stop*. The color *Stop* is propagated over the (*illegal*) segment, rooted by the illegal root, unless non-base copying is reached, meaning that the segment is "dead" and will be removed. After the copying operation arrives at the root, it circulates `safe` signal (auxiliary token) in a depth-first manner. The `safe`, usually has some randomized value, also uses two predefined bases, 0 and 1. When the neighboring process p_k has $Safe.k = 0$, the activated process p_i sets at $Safe.i = 0$ meaning that the `safe` is going forward. On the other hand, p_k has $Safe.k = 1$, then the `safe` is backtracking.

Before the formal description of our method, we should state our notation. Let $D.i$ be the `descendant` of p_i, $C.i$ the `color` of p_i. Let $D.D.i$ be the `descendant` of `descendant` of p_i, and so forth. Additionally, let $P.i$ be the `parent` of p_i, $NB.i$ the set of p_i's neighbors, $NP.i$ the number of p_i's parents. If $D.i$ points to some base, we denote $D.i = base$. If the base corresponds to p_k, we express $base_k$. If $D.i$ points to non-base, we say $D.i = nonbase$. Let *Stop* and *Wait* be some specified non-base values. We sometimes denote the root (index) as r.

3.2 Formal Description of *AFPG*

We describe the formal description of our *AFPG* in the appendix.

4 Correctness

In this section, we briefly describe proof sketches based on the following assumption.

Assumption 1 *The faulty state of $D.i$ (resp. $Safe.i$) takes on any integer (resp. real) value over the interval $[0, K \cdot (deg(v_i) - 1)]$ or $[0, K \cdot deg(v_r)]$ (resp. $[0, 1]$) with equal probability. Any two faulty state occurs independently.* □

Lemma 1. *Our algorithm AFPG is deadlock-free.*

Proof. Suppose that a deadlock occurs, that is, there is no enabled process. More precisely, (1) There is no illegal root/segment, otherwise the *IllegalRoot.i* would be true for such p_i. Even if $NP.i = 0$ is true, the illegal root/segment is colored *Stop.* (2) There is no non-base `descendant`, otherwise the *ResetNull.i* would be true for such p_i. (3) Every process has the same color or there is no (legal) segment, otherwise *Token.i* may be true. In the former case, *BToken.i* holds and *search*(i) begins. In the latter case, *Token.r* is true if $Safe.r \neq 0$. So we suppose that $Safe.r = 0$. Then *SCandidate.i* ($i = r$) is true and a legal segment generates; a contradiction. □

To prove the convergence property, we use the *pseudo legitimate states* through which the system stabilizes.

$$PLS1 \equiv (|IllegalRoots| = 0) \wedge (|Cycles| = 0)$$
$$PLS2 \equiv PLS1 \wedge (|IsolatedNodes| = 0)$$
$$PLS3 \equiv PLS2 \wedge (\forall i, j : C.i = C.j \neq Stop),$$

where *Cycles* is a set of cycles formed by descendant paths, and the other sets described above are defined as

$$IllegalRoots = \{i \mid i \neq r, NP.i \neq 1, D.i = base, C.i \neq Stop\}$$
$$IsolatedNodes = \{i \mid D.i = nonbase,\ NP.i = 0\}.$$

Lemma 2. *PLS1 eventually holds.*

Proof. (1) Suppose that there exists some $D.j = nonbase$, where the process j is reached from the root by the `descendant` path. After the *SpreadUp.i* rule is repeatedly applied, the *ResetNull.r* inevitably holds and the circulation of `safe` signal begins. For other illegal segments, there are two cases whether the ends of the segments have base or non-base `descendant` pointers. If it had a non-base `descendant` pointer, the *SpreadUp.i* rule would be repeatedly applied and would reach the illegal root. Otherwise, the illegal root would be colored by *Stop* and all the processes on the segment would be colored by *Stop* by the *IllegalRoot.i* and the *SpreadDown.i* rules. It means that their activities are "stopped" until the `safe` signal comes. In any case, the number of $|IllegalRoots|$ decreases. Thus every illegal root is eventually eliminated. Other rules do not affect the number of $|IllegalRoots|$.

(2) Suppose that the corrupted $D.j$ reachable from the root takes on some other base.

For other illegal segments, they will be eliminated or "stopped" as above. The segment with the (true) root continues token passing. If the segment reaches an illegal segment and *SpreadUp.i* rule is being applied, it reaches the root and the circulation of `safe` signal begins. If the segment reaches an "stopped" segment, the color *Stop* is corrected by the action of *Token.i*. Thus such segments will be eliminated.

Even if there were cycles by the `descendant` pointers, $NP.i$ would become two when the legal segment reaches some process on a cycle. Then the cycle is broken by the *ResetNull.i* rule. Thus every cycle or illegal root is eventually eliminated. □

Lemma 3. *PLS2 eventually holds.*

Proof. For the processes in *IsolatedNodes*, the *ResetNull.i* rule is applied. After all the illegal segments are eliminated, the *ResetNull.i* rule monotonically decreases the number of $|IsolatedNodes|$. Other rules do not affect the number of $|IsolatedNodes|$. □

Lemma 4. *After circulating the* `safe` *signal, PLS3 holds.*

Proof. The `safe` signal is at first set to 0 by the root (*ResetNull.r*) and is circulated. If there are some neighboring non-base state, setting the `descendant` pointer is delayed ($D.i := Wait$) until the non-base value is reset. Hence all the states are reset after the `safe` returning. If there are neither non-base states ($\neg NonBase.i$) nor unvisited processes ($\neg SCandidate.i$) with base states, `safe` is reset to 1. Then the predicate *BSafePass.i* plays a backtracking role. After it returns to the root, *Safe.r* changes from 0 to 1. Through the `safe` signal circulation, the color is unified (*SafePass.i*). Thus the PLS3 holds. □

After PLS3 holds, there is no segment. Since $Safe.r = 1$ and $D.r = Null$, *Token.r* is true. Therefore a legitimate configuration is restored.

Lemma 5. *Let n be the number of processes, δ the minimum degree, and p the fault probability. Then the probability that some faulty privileges exist is at most* $\frac{(n+1)p}{2K(\delta-1)}$.

Proof. For each process p_i ($i \neq r$), it must have exactly one parent and $D.i = Null$ to get a faulty privilege. Since the neighboring two processes cannot do at the same time, at most $\lceil \frac{n}{2} \rceil \leq \frac{n+1}{2}$ processes are faulty. In addition, each process has $D.i = Null$ with probability $p \cdot \frac{1}{K(\delta-1)+1} < p \cdot \frac{1}{K(\delta-1)}$ greater than the one for p_r. Multiplying them derives the probability. □

From lemmas above, the next theorem follows.

Theorem 1. *Our algorithm AFPG eventually stabilizes with avoiding almost all faulty privileges.* □

5 Experimental Results

5.1 Simulation Model

To evaluate *AFPG*, we execute simulation experiments and compare it with the *Huang&Chen*. The simulation programs are implemented by *C* language. Our simulation model has an underlying arbitrary network, where each process corresponds to a process. The network is generated by our random graph routine, which selects the largest connected component from randomly connected processes with probability p. Each process is activated at random by the C-daemon. Then the only enabled process can change its state. The mean interval time between activations, *ROUND*, is exponentially distributed with mean 0.2. If the activated process has a token, it gets into the critical region. The time staying in the critical region, *CRITICAL*, is normally distributed with mean 1 and variance $CSIGMA = 0.3$. On average, *fnum* faults occur with variance $FSIGMA = 3$. The mean interval time of fault occurrences, *FROUND*, is also exponentially distributed with mean 20. As shown in the following results, the averaged stabilization time may beyond 20. Thus faults sometimes occur when the system has not been restored from the previous damage yet.

We execute three experiments varying parameters, the number of entire processes, *nsize*, the probability of connecting a pair of processes, *prob*, and the mean number of faults at a

Constant	Value	Meaning
K	1000	Large constant for a base $j \cdot K$
$CRITICAL$	1	Mean staying time in the critical region
$CSIGMA$	0.3	Variance of staying time in the critical region
$ROUND$	0.2	Mean interval time between activations
$FSIGMA$	3	Variance of number of faults
$FROUND$	20	Mean interval time between faults

Table 1. Constants

Parameter	Range	Standard	Meaning
nsize	8 — 26	20	Number of processes
prob	0.3 — 0.75	0.3	Probability of connecting a pair of processes
fnum	1 — 19	2	Mean number of faults

Table 2. Parameters

time, *fnum*. Experiment 1 examines the effects of network size, *nsize*, varying from 8 to 26. Experiment 2 examines the effects of the density of graph, *prob*, varying from 0.3 to 0.75. Experiment 3 examines the effects of damages, *fnum*, varying from 1 to 19. When varying a parameter, we keep other parameters standard values shown in Table 2. The number of convergences is 100 for a generated network, and 100 kinds of networks are generated. The constants and parameters are summarized in Tables 1 and 2.

5.2 Results

In Fig.s 1–3, the left hand side one shows the mean stabilization time, and the right hand side one shows the rate of faulty privileges. The results of Experiment 1 are depicted in Fig. 1. It is obvious that our *AFPG* outperforms *Huang&Chen*. The mean stabilization time of *AFPG* is less than a half of *Huang&Chen*.

The results of Experiment 2 are depicted in Fig. 2. It seems that the mean stabilization time reaches a peak for dense networks. (for large *prob* values). Since the number of edges is saturated for $prob > 0.5$, the mean stabilization time grows in proportion to the number of edges. It is coceivable that there can be more choices for corrupted descendant values in *Huang&Chen*. In contrast, there is no such choices in our *AFPG*. The slight increase in *AFPG* is caused by the length of the segment and frequent *Wait* operation because of many corrupted adjacent processes.

The results of Experiment 3 are depicted in Fig. 3. When the mean number of faults is $fnum = 1$, the mean stabilization time between the two methods is not so different. However, as the number of faults increases, the performance of *Huang&Chen* sharply gets worse. Then it reaches a peak when $fnum = 5$, twice our *AFPG*. It can be seen that it is bounded by the length of the segment. That is, the performance of *Huang&Chen* is affected if processes on the segment is corrupted. Even if the processes out of the segment are hit by a fault, no restoring operations are necessary. Since our standard network size is $nsize = 20$, the probability that the segment is broken is determined by the number of faults. Thus there is no effect on the performance even if too many faults occur.

Concerning the rate of faulty privileges, our *AFPG* always keeps less than 0.3 %. In Fig.3, the rate of faulty privileges in *Huang&Chen* sharply increases as the mean number of faults grows. This is reasonable because descendant pointers arbitrarily direct other processes or *Null* in *Huang&Chen*. Hence the rate of faulty privileges is in proportion to the number of faults.

6 Concluding Remarks

In almost every case, the mean stabilization time of our *AFPG* is shorter than that of

Huang&Chen. This is because there is no need for *Error* coloring phase in our *AFPG*. There are two reasons for the unnecessary. First, each process can detect its descendant faulty in almost every case. Thus copying the corrupted value is equivalent to breaking the segment rapidly. Second, the livelock risk, pointed out in [6], can be removed by our *Stop* coloring.

References

1. E.J.H.Chang, G.H.Gonnet, and D.Rotem, "On the costs of self-stabilization," *Information Processing Letters*, 24:311–316, 1987.
2. S.Ghosh and A.Gupta, "An exercise in fault-containment: Self-stabilizing leader election," *Information Processing Letters*, 59:281–288, 1996.
3. S.Ghosh and X.He, "Fault-containing self-stabilization using priority scheduling," *Information Processing Letters*, 73, 145–151, 2000.
4. T.Herman, "Superstabilizing mutual exclusion," *Distributed Computing*,13,1:1–17, 2000.
5. T.Herman and S.Pemmaraju, "Error-detecting codes and fault-containing self-stabilization," *Information Processing Letters*, 73, 41–46, 2000.
6. S.T.Huang and N.S.Chen, "Self-stabilizing depth-first token circulation on networks," *Distributed Computing*, 7: 61–66, 1993.
7. S.T.Huang and L.C.Wuu, "Self-stabilizing token circulation in uniform networks," *Distributed Computing*, 10: 181–187, 1997.
8. C.Johnen, G.Alari, J.Beauquier and A.K.Datta, "Self-stabilizing depth-first token passing on rooted networks," In *Proceedings WDAG97 Distributed Algorithms 11th International Workshop*, Springer-Verlag LNCS1320: 260-274, 1997.
9. J.Kiniwa, "Avoiding almost all faulty privileges in almost linear convergence time," In *Proceedings of 5th Japan-Korea Joint Workshop on Algorithms and Computation*, 33–40, 2000.
10. S.Kutten and B.Patt-Shamir, "Time-adaptive self-stabilization," In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, 149–158, 1997.
11. H,Masuda, Y.Tsujino and N.Tokura, "A self-stabilizing leader election algorithm on the multi-access channel," *Transactions of IEICE*, J80-D-I, 1, 1-10, 1997.
12. IL.Yen, "A highly safe self-stabilizing mutual exclusion algorithm," *Information Processing Letters*, 57:301-305, 1996.

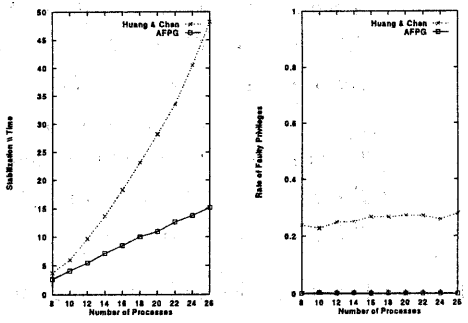

Fig. 1. Varying Number of Nodes

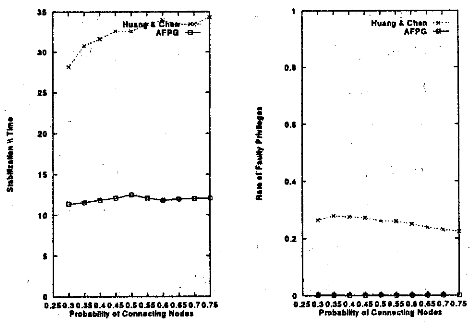

Fig. 2. Varying Connecting Probability

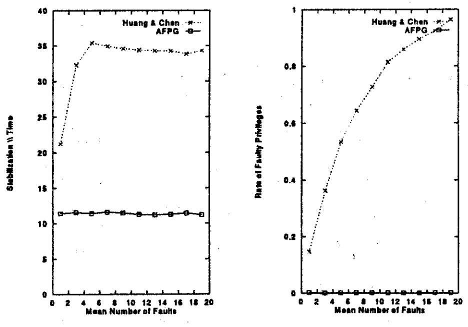

Fig. 3. Varying Number of Faults

The token passing rules and their predicates are described as follows.

$$\begin{aligned}
\mathit{Token.r} &\equiv (D.r = \mathit{Null}) \wedge (\mathit{Safe.r} \neq 0)\\
\mathit{Token.r} &\rightarrow C.r := (C.r+1) \bmod 2;\;\; \mathit{Safe.r} := \mathit{randomize}();\;\; \mathit{search}(r)\\
\mathit{Token.i} &\equiv (i \neq r) \wedge (NP.i = 1) \wedge ((D.i = \mathit{Null}) \vee (C.i = \mathit{Stop}))\\
&\quad \wedge (C.P.i \neq C.i) \wedge (\mathit{Safe.P.i} \neq 0, 1)\\
\mathit{Token.i} &\rightarrow C.i := C.P.i;\;\; \mathit{Safe.i} := \mathit{randomize}();\;\; \mathit{search}(i)\\
\mathit{BToken.i} &\equiv (D.i = \mathit{base}) \wedge (D.D.i = \mathit{Null}) \wedge (C.i = C.D.i) \wedge (\mathit{Safe.i} \neq 0)\\
\mathit{BToken.i} &\rightarrow \mathit{search}(i)
\end{aligned}$$

The error handling rules and their predicates are as follows.

$$\begin{aligned}
\mathit{IllegalRoot.i} &\equiv (i \neq r) \wedge (NP.i = 0) \wedge (D.i = \mathit{base}) \wedge (C.i \neq \mathit{Stop})\\
\mathit{IllegalRoot.i} &\rightarrow C.i := \mathit{Stop}\\
\mathit{SpreadDown.i} &\equiv (C.P.i = \mathit{Stop}) \wedge (C.i \neq \mathit{Stop})\\
\mathit{SpreadDown.i} &\rightarrow C.i := \mathit{Stop}\\
\mathit{SpreadUp.i} &\equiv (D.i = \mathit{base}) \wedge (D.D.i = \mathit{nonbase})\\
\mathit{SpreadUp.i} &\rightarrow D.i := D.D.i\\
\mathit{ResetNull.r} &\equiv (D.r = \mathit{nonbase}) \vee (\mathit{Safe.r} = 0)\\
\mathit{ResetNull.r} &\rightarrow D.r := \mathit{Null};\;\; \mathit{Safe.r} := 0;\;\; \mathit{SafeCirculate}(r)\\
\mathit{ResetNull.i} &\equiv (i \neq r) \wedge (((NP.i = 0) \wedge (D.i = \mathit{nonbase})) \vee (NP.i > 1) \vee (D.i = r))\\
\mathit{ResetNull.i} &\rightarrow D.i := \mathit{Null}\\
\mathit{SafePass.i} &\equiv (NP.i = 1) \wedge (D.i \neq \mathit{nonbase}) \wedge (\mathit{Safe.P.i} = 0) \wedge (\mathit{Safe.i} \neq 0, 1)\\
\mathit{SafePass.i} &\rightarrow \mathit{Safe.i} := 0;\;\; C.i := C.P.i;\;\; \mathit{SafeCirculate}(i)\\
\mathit{BSafePass.i} &\equiv (D.i = \mathit{base}) \wedge (D.D.i = \mathit{Null}) \wedge (\mathit{Safe.i} = 0) \wedge (\mathit{Safe.D.i} = 1)\\
\mathit{BSafePass.i} &\rightarrow \mathit{SafeCirculate}(i)
\end{aligned}$$

procedure *search*(i)

$$\begin{aligned}
\mathit{Candidate.i} &\equiv (\exists k \in NB.i - r : (C.k \neq C.i) \wedge ((D.k = \mathit{Null}) \vee (C.k = \mathit{Stop})))\\
\mathit{Candidate.i} &\rightarrow D.i := \mathit{base}_k\\
\neg \mathit{Candidate.i} &\rightarrow D.i := \mathit{Null}
\end{aligned}$$

procedure *SafeCirculate*(i)

$$\begin{aligned}
\mathit{NonBase.i} &\equiv (\exists k \in NB.i - r : (D.k = \mathit{nonbase}))\\
\mathit{NonBase.i} &\rightarrow D.i := \mathit{Wait}\\
\mathit{SCandidate.i} &\equiv (\mathit{Safe.i} = 0) \wedge (\exists k \in NB.i - r : (\mathit{Safe.k} \neq 0, 1) \wedge (D.k = \mathit{base}))\\
\mathit{SCandidate.i} &\rightarrow D.i := \mathit{base}_k\\
\neg \mathit{NonBase.i} &\\
\wedge \neg \mathit{SCandidate.i} &\rightarrow D.i := \mathit{Null};\;\; \mathit{Safe.i} := 1
\end{aligned}$$