State Checksum and Its Role in System Stabilization

Chin-Tser Huang
Dept of Computer Science and Engineering
University of South Carolina at Columbia
huangct@cse.sc.edu

Mohamed G. Gouda
Department of Computer Sciences
The University of Texas at Austin
gouda@cs.utexas.edu

Abstract

Although a self-stabilizing system that suffers from a transient fault is guaranteed to converge to a legitimate state after a finite number of steps, the convergence can be slow if the harmful effects of the fault are allowed to propagate into many processes in the system. Moreover, some safety properties of the system may be violated during the convergence. To address these problems, we propose in this paper the concept of a state checksum -- a redundancy that can be added to the state of a self-stabilizing system so that some classes of faults become visible to the system, and the system can limit the propagation of their harmful effects, and maintain its safety properties during the convergence. To make these concepts concrete, we discuss the case study of a token ring and show how to use fault-detecting and fault-correcting checksums to detect visible faults, limit the propagation of their harmful effects, and ensure that the safety properties of the ring are maintained during the convergence from these faults.

1. Introduction

Assurance is an increasingly desirable requirement in system design. The objective of system assurance is to provide the guarantee that the system can recover from or tolerate the occurrence of heterogeneous classes of faults. Self-stabilization, which was first introduced by Dijkstra in 1974 [5], is one way to provide system assurance. A self-stabilizing system is designed in such a way that starting from any arbitrary state, it is guaranteed that the system will converge to a legitimate state after a finite number of steps without any interference from a centralized supervisor. This also means that a self-stabilizing system that suffers from any transient fault is guaranteed to converge to a legitimate state after a finite number of steps. However, even with such desirable property, a self-stabilizing system still has the following two shortcomings when it is recovering from a fault. First, if the harmful effects of the fault are allowed to propagate into many processes in the system, then the convergence can be complicated and slow. Second, some safety properties of the system may be violated during the period of convergence.

In this paper, we propose to add some redundancy called state checksums to a self-stabilizing system to overcome the aforementioned two shortcomings. A state checksum can make some classes of faults "visible" to the system as follows: if a process in a system suffers from a visible fault, then its state checksum becomes inconsistent with its state that is corrupted by the fault. Therefore, the process that suffers from a visible fault and all of its neighboring processes can detect the occurrence of the visible fault from the inconsistency between the state checksum and the corrupted state value of this process, and can contain the harmful effects of the fault in this process until the corrupted state value is corrected.

Arora and Kulkarni ever proposed the concept of multitolerance in [2], which provides a theoretical basis for designing systems that can tolerate different classes of faults in different ways. As a matter of fact, the addition of state checksums to a self-stabilizing system is aimed to make the system multitolerant in the following way. On one hand, if the fault that occurs to the system is visible to the state checksum, then the process that is suffering from the fault will detect the occurrence of the fault, and will correct the corrupted state value before it is accessed by other neighboring processes. On the other hand, if the fault is not visible to the state checksum, then the system can still converge to a legitimate state via self-stabilization, although the two aforementioned shortcomings may still exist.

The rest of this paper is organized as follows. In Section 2, we explain the concept of state checksum in some detail. In the next four sections, we illustrate the use of state checksums using the case study of stabilizing token ring proposed by Dijkstra in [5]. We first review the original stabilizing token ring in Section 3, then show how to apply fault-detecting state checksums on the token ring in Section 4, how to remove harmful effects of visible faults on the token ring in Section 5, and how to apply fault-correcting state checksums on the token ring in Section 6. We draw some concluding remarks in Section 7.

2. State Checksum

Before getting into the details of state checksum, we first give an overview of the model we use in this paper. A system is composed of a number of processes that are connected in a directed graph. A state of a system is defined by one value for each variable in each process in the system. A local state of a process is defined by one value for each variable in the process. The legitimate states of a self-stabilizing system can be defined as satisfying the following two conditions. The first is the *closure property*: if the system is currently in a legitimate state, then executing any enabled action will bring the system to another legitimate state. The second is the *convergence property*: starting from any arbitrary state, any execution of a sequence of actions will bring the system to a legitimate state in finite number of steps [1].

State checksum is a redundancy that is added to every process in a system such that starting from a legitimate state of the system, if the local state of one process is corrupted by some classes of faults, the state checksum of this process will become inconsistent with the corrupted local state of this process, thus the fault can be detected by checking the local state against the state checksum. These classes of faults that can be detected with a state checksum are called visible faults. In contrast, if the local state of one process is corrupted by some other classes of faults, the state checksum of this process may remain consistent with the corrupted state, thus the fault cannot be detected by checking the local state against the state checksum. These classes of faults that cannot be detected with a state checksum are called invisible faults.

The purpose of using state checksums is two-fold. First, state checksums can be used to contain the "harmful effects" of visible faults. The state checksum of a process indicates whether this process is suffering from a visible fault or not. Therefore, by checking the state checksum of a neighboring process, a process can judge whether its neighboring process is currently suffering from a visible fault or not, and if so, can avoid accessing the variables of its neighboring process before the values of its corrupted variables are corrected. In this way, the visible fault will not propagate into other processes and the harmful effects of the visible fault, if any, will be contained in the processes that suffer from the visible fault. Second, state checksums can lead the system to fast convergence from visible faults. Although a self-stabilizing system can converge to a legitimate state by itself, the convergence will become faster with the help of state checksums because the fault does not propagate to make the situation complicated.

In this paper, we are concerned about two types of state checksums: fault-detecting and fault-correcting. Fault-detecting state checksums can be used to detect faults that are visible to them so that these faults will not propagate into other processes and the harmful effects of these faults will be contained. An example of adding fault-detecting checksum to a process is by adding one parity bit to represent the number of 1's in all bits of an important variable: for an important variable x, if the total number of 1's in all bits of x is odd, then the parity bit is set to 1; otherwise the parity bit is set to 0. If a fault corrupts an odd number of bits in variable x, then the parity bit will become inconsistent with variable x, and thus this fault is visible. On the other hand, if a fault corrupts an even number of bits in variable x, then the parity bit remains consistent with variable x, and thus this fault is invisible. The parity of variable x can be computed using a parity function PRT. More specifically, for a variable x and its parity bit c,

$\mathrm{PRT}(x) \neq c$ if a visible fault occurs to variable x,
$\mathrm{PRT}(x) = c$ if no fault occurs to variable x or an invisible fault occurs to variable x

Fault-correcting state checksums contain extra information such that it not only can be used to detect visible faults, but can also be used to correct the variables that are corrupted by visible faults to their original values. An example of adding fault-correcting checksum to a process is by using redundant variables to store the value of an important variable: for an important variable x, introduce two more variables y and z of the same size as x. When the process needs to update the value of variable x, it will write the new value to y and z at the same time. When the process needs to access the value of variable x, then instead of just reading the value of x, it will use a majority function MJR to read all three of x, y, and z, and return the majority value at every bit position. More specifically,

MJR(x, y, z) = for the i^{th} bit, where $1 \leq i \leq$ number of bits in variable x,
i^{th} bit = 1 if at least two of i^{th} bit in x, i^{th} bit in y, and i^{th} bit in z are 1,

i^{th} bit = 0 otherwise

From the definition of the majority function MJR, it can be seen that if at every bit position at most one of x, y, and z is corrupted, then the fault is visible and the function MJR returns the correct value regardless of any corrupted bit in x, y, or z. In this case, we can say that the corruption is indeed masked by the majority function because the majority function always returns the correct value. We can also say that the system that uses the majority function *snap-stabilizes* [4] when suffering from visible faults, because the system in fact remains in a legitimate state. However, if at any bit position at least two of x, y, and z are corrupted, then the fault is invisible and the system has to converge through stabilization.

3. A Stabilizing Token Ring

In this section, we give a review of the classic example of stabilizing token ring proposed by Dijkstra in [5]. We first present a protocol of the token ring using a variation of Abstract Protocol Notation introduced in [7]. Then, in the subsequent three sections, we will modify the protocol to illustrate the different uses of state checksums.

Consider n processes that are indexed from 0 to n-1 and are placed in a ring; i.e. for process i, $0 \le i \le n-1$, the left neighbor of process i is process (i-1) mod n, and the right neighbor of process i is process (i+1) mod n. Each process has a variable x whose range is from 0 to n-1. With the exception of process 0, each process i, $1 \le i \le n-1$, has one action whose guard is whether the value of its variable x.i is not equal to the value of its left neighbor's variable x.(i-1). If the guard is true, then the action is enabled: process i executes its critical section and then sets x.i to be equal to the value of x.(i-1). For process 0, it has one action whose guard is whether the value of its variable x.0 is equal to the value of process (n-1)'s variable x.(n-1). If the guard is true, then the action is enabled: process 0 executes its critical section and then sets its variable x.0 to (x.0 + 1) mod n. Process 0 and processes 1..n-1 can be defined as follows.

```
process 0
var x.0  :    0..n-1
begin
      x.0 = x.(n-1) →   critical section;
                        x.0 := x.0 +_n 1
end

process i : 1..n-1
var x.i  :    0..n-1
begin
      x.i ≠ x.(i-1) →   critical section;
                        x.i := x.(i-1)
end
```

The legitimate states of the token ring can be defined as follows: in a legitimate state there exists an i, where $0 \le i \le n-1$, such that for every $j \le i$, x.j = x.0, and for every $j > i$, x.j = (x.0 – 1) mod n. The closure and convergence properties of this system are as follows. First, the execution of an enabled action in a legitimate state will bring the token ring into another legitimate state. Second, regardless of the initial state, the token ring is guaranteed to enter a legitimate state after a finite number of steps. These properties can be easily verified from the protocol.

Note that one important property of the token ring is that it satisfies mutual exclusion. It can be seen in that in each legitimate state there is only one enabled action in all the processes, which means at any time only one process can be executing its critical section. If process i is the one which is executing its critical section, then after process i finishes the execution of its action, the action in process (i + 1) mod n is enabled and process (i + 1) mod n gets the right to execute its critical section. This is why it is called a token ring.

It has been verified that in the worst case it can take up to $O(n^2)$ for the token ring to converge to a legitimate state. A proof can be found in [6].

4. The Token Ring with Detecting Checksums

Although a self-stabilizing system that suffers from a fault can always converge to a legitimate state through stabilization, the convergence can be very slow. This is because the faults are allowed to propagate into many other processes, which can make the convergence complicated. In order to limit the propagation of the faults, fault-detecting checksums can be added to a self-stabilizing system.

For example, consider the scenario in which each variable x in processes 0..n-1 in the token ring has the value 0. This is a legitimate state of the token ring. Suppose the variable x.1 in process 1 is corrupted by a fault and its value becomes 4. The action in process 2 becomes enabled and the corrupted value 4 is copied into variable x.2. This corrupted value can keep propagating into processes 3, 4, and so on. Although process 1 can later correct the value of x.1 by copying from the value of x.0, it will take several steps for subsequent processes to correct all the propagated values. If there are more faults, the convergence can become more complicated.

To limit the propagation of the faults in the token ring, we can add to each process i, where $1 \leq i \leq n-1$, a parity bit c that represents the parity of variable x. When process i, where $1 \leq i \leq n-1$, finds that parity bit c.i is inconsistent with variable x.i, process i detects that a visible fault has occurred to variable x.i. To overcome this fault, process i should recover the value of x.i by copying the value of x.(i-1) from its neighbor process (i-1). On the other hand, process i, where $2 \leq i \leq n-1$, will check whether the parity bit c.(i-1) is consistent with x.(i-1), to avoid copying the value of a corrupted x.(i-1) and let this corrupted value propagate into x.i. Every time process i, $1 \leq i \leq n-1$, copies the value of x.(i-1) into x.i, it will reset the parity bit c.i to make it consistent with x.i. For process 0, it will also check whether the parity bit c.(n-1) is consistent with x.(n-1), to avoid copying the value of a corrupted x.(n-1) and let this corrupted value propagate into x.0. Note that we do not add a parity bit to process 0 to avoid the deadlock in which each process is suffering from a visible fault. Process 0, process 1, and processes 2..n-1 in the token ring can be modified to include the fault-detecting checksum as follows.

```
process 0
var x.0  :    0..n-1
begin
    x.0 = x.(n-1)  ∧  c.(n-1) = PRT(x.(n-1)) →
        critical section;
        x.0 := x.0 +_n 1
end

process 1
var x.1  :    0..n-1,
    c.1  :    0..1
begin
    x.1 ≠ x.0  ∨  c.1 ≠ PRT(x.1) →
        critical section;
        x.1 := x.0;
        c.1 := PRT(x.1)
end

process i : 2..n-1
var x.i  :    0..n-1,
    c.i  :    0..1
begin
    (x.i ≠ x.(i-1)  ∨  c.i ≠ PRT(x.i))  ∧  c.(i-1) =
PRT(x.(i-1)) →
        critical section;
        x.i := x.(i-1);
        c.i := PRT(x.i)
end
```

We have seen in the last section that the stabilizing token ring can converge from an arbitrary state to a legitimate state in $O(n^2)$ steps. By contrast, if fault-detecting checksums are added to the processes in the token ring and all the faults suffered by the processes in the token ring are visible faults, then the token ring can converge fast to a legitimate state in just O(k) steps, where k is the number of processes that suffer from a visible fault. This is because in the case of the token ring with detecting checksums, the faults do not propagate into other processes, and each of the processes suffering from the fault just needs to execute its action once to correct its corrupted state. However, if any one of these faults is an invisible fault, then this invisible fault may propagate into other processes, and the token ring may still have to converge through stabilization.

5. Removing the Harmful Effects of Visible Faults

Although fault-detecting state checksums can be used to detect visible faults and let the system converge fast to a legitimate state, the visible faults may have some harmful effects that cannot be cured even if the system converges to a legitimate state. In this case, it is desirable that additional cares are taken to remove the harmful effects caused by visible faults during the period of convergence.

For example, in the protocol of the token ring with detecting checksum presented in the last section, multiple processes may be executing their critical sections at the same time if multiple processes suffer from visible faults at the same time. This will violate the mutual exclusion property of the token ring.

To ensure that the mutual exclusion property is not violated, we can replace the "critical section" statement in each process i, where $1 \leq i \leq n-1$, by the following "if .. then .. fi" statement:

```
if  x.i = x.(i-1) -_n 1  ∧  c.i = PRT(x.i) →
      critical section
fi
```

With this replacement, a process will first check whether it is suffering from a visible fault before executing its critical section. Therefore, although multiple processes may be executing their actions at the same time, it is guaranteed that only one process, which is a process not suffering from any visible fault, will be executing its critical section.

6. The Token Ring with Detecting and Correcting Checksums

We have shown that with fault-detecting checksums added to the processes in a self-stabilizing system, convergence from visible faults will become faster and safety properties of the system will be maintained. However, there is no guarantee that the corrupted variables will be restored to the values that they held prior to the corruption when the system converges to a legitimate state. If in some cases it is desired that each corrupted variable should be corrected to the value held prior to the corruption, then correcting checksums should to be used.

For example, consider the scenario in which each variable x in processes 0..n-1 in the token ring has the value 0. This is a legitimate state of the token ring. Suppose the variable x.0 in process 0 is corrupted by a fault and its value becomes 3, and the variable x.1 in process 1 is corrupted by a fault and its value becomes 1. The detecting checksum in process 1 can be used to detect the fault (because the total number of 1's in x becomes odd from even) and the token ring can later converge to a legitimate state, in which each variable x in processes 0..n-1 in the token ring has the value 3. However, this value is not the same value held by each variable x prior to the occurrence of the fault.

To get the correct value of x.0 that is corrupted by a visible fault, we can introduce two redundant variables y.0 and z.0 into process 0 to store the value of x.0. To represent numbers from 0 to n-1, each of x.0, y.0, and z.0 needs to be $\lceil \log n \rceil$ bits long. The majority function MJR introduced in Section 2 can be applied on x.0, y.0, and z.0 to return a result that is also $\lceil \log n \rceil$ bits long. This result contains the majority value at every bit position. The protocol in the last section needs to be modified as follows. For process 0, the guard of its action becomes whether the value of MJR(x.0, y.0, z.0) is equal to the value of x.(n-1), and whether the parity bit c.(n-1) is consistent with x.(n-1). If the guard is true, then the action is enabled: process 0 executes its critical section and then sets its variables x.0, y.0, and z.0 to (x.(n-1) + 1) mod n. For process 1, instead of accessing the value of x.0, it will access the value of MJR(x.0, y.0, z.0). Note that we only add the fault-correcting checksum to process 0, because other processes can get the correct value from process 0. Process 0, process 1, and processes 2..n-1 in the token ring can be modified to include the fault-correcting checksum as follows.

```
process 0
var x.0 :   0..n-1,
    y.0 :   0..n-1,
    z.0 :   0..n-1
begin
    MJR(x.0, y.0, z.0) = x.(n-1) $\wedge$ c.(n-1) = PRT(x.(n-1)) $\rightarrow$
        critical section;
        x.0 := x.(n-1) $+_n$ 1;
        y.0 := x.(n-1) $+_n$ 1;
        z.0 := x.(n-1) $+_n$ 1
end

process 1
var x.1 :   0..n-1,
    c.1 :   0..1
begin
    x.1 $\neq$ MJR(x.0, y.0, z.0) $\vee$ c.1 $\neq$ PRT(x.1) $\rightarrow$
        if  x.1 = MJR(x.0, y.0, z.0) $-_n$ 1 $\wedge$ c.1 = PRT(x.1) $\rightarrow$
            critical section
        fi;
        x.1 := MJR(x.0, y.0, z.0);
        c.1 := PRT(x.1)
end

process i : 2..n-1
var x.i :   0..n-1,
    c.i :   0..1
begin
    (x.i $\neq$ x.(i-1) $\vee$ c.i $\neq$ PRT(x.i)) $\wedge$ c.(i-1) = PRT(x.(i-1)) $\rightarrow$
        if  x.i = x.(i-1) $-_n$ 1 $\wedge$ c.i = PRT(x.i) $\rightarrow$
            critical section
        fi;
        x.i := x.(i-1);
        c.i := PRT(x.i)
end
```

According to the definition of the function MJR, if at every bit position at most one of x.0, y.0, and z.0 is corrupted, then the fault is visible and the function MJR returns the correct value regardless of any corrupted bit in x.0, y.0, or z.0. Since the majority function returns the correct value, the token ring remains in a legitimate state. However, if at any bit position at least two of x.0, y.0, and z.0 are corrupted, then the fault is invisible and the token ring has to converge through stabilization.

7. Concluding Remarks

In this paper, we introduce the concept of state checksum and how it can be used to achieve fault containment. We use the case study of Dijkstra's stabilizing token ring to illustrate the functions and usage of state checksums. The conclusion that can be drawn from our presentation of state checksum is as follows: if all the faults that the processes in the system are suffering from are visible to the state checksum used in the system, then the system can converge fast to a legitimate state, and all the safety properties are maintained during the convergence from these faults. However, if any one of these faults is invisible to the state checksum used in the system, then the system can still converge through stabilization, but the convergence can be slow, and the safety properties of the system might be violated during the convergence from these faults. This way, we make the system multitolerant to visible and invisible faults. Indeed, the techniques described in this paper can be generalized and applied in every stabilizing system to preserve stabilization properties of the system when the system suffers from visible faults.

It is worthy to note that the problem of multitolerance design in token ring has been investigated by Arora and Kulkarni in [2]. However, we note that there are three key differences between their design and our design. First, their design is based on the assumption that corruption of the state of a process is detectable, which means that the corrupted state is detected by the process itself before any action inadvertently accesses that state (however they did not specify how the corrupted state is detected). By contrast, our design does not require explicit detection of the corrupted state, because inconsistency between a state and its checksum already indicates that the state is corrupted. Second, they introduce two special values $\perp$ and $\top$ to represent corrupted states, while our design uses extra variable(s) to store the state checksum. Third, their design does not allow self-stabilization, while our design allows the system that suffers from invisible faults to converge through stabilization.

An interesting question to ask is that given a system equipped with state checksum, if it is already revealed to us that the system is currently in a faulty state, can we determine whether the fault is visible or invisible to the state checksum? The answer is yes, because if the state checksum is inconsistent with the state, then the fault is visible to the state checksum, and if the state checksum is consistent with the state, then the fault is invisible. If the fault is visible and the checksum in the system is a fault-correcting checksum, then the faulty state can be corrected.

References

[1] Anish Arora, Mohamed G. Gouda, "Closure and Convergence: A Foundation of Fault-Tolerant Computing", IEEE Transactions on Software Engineering, Vol. 19, No. 11, November 1993.

[2] Anish Arora, Sandeep S. Kulkarni, "Component Based Design of Multitolerant Systems", IEEE Transactions on Software Engineering, Vol. 24, No. 1, January 1998.

[3] Anish Arora, Sandeep S. Kulkarni, "Detectors and Correctors: A Theory of Fault-Tolerance Components", Proceedings of the 18th International Conference on Distributed Computing Systems, 1998.

[4] Alain Bui, Ajoy K. Datta, Franck Petit, Vincent Villain, "State-Optimal Snap-Stabilizing PIF in Tree Networks", Proceedings of the Fourth Workshop on Self-Stabilizing Systems, 1999.

[5] Edsger W. Dijkstra, "Self-stabilizing Systems in Spite of Distributed Control", Communications of the ACM, Vol. 17, No. 11, November 1974.

[6] Shlomi Dolev, Self-Stabilization, MIT Press, 2000.

[7] Mohamed G. Gouda, Elements of Network Protocol Design, John Wiley & Sons, 1998.