Byzantine Agreement–
Algorithms and Applications

A Thesis

Presented to

The Faculty of the Department of Computer Science

University of Houston–University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Yi Zhao

December, 1987

ACKNOWLEDGEMENTS

It is a pleasure to express my sincere gratitude and deep appreciation to my advisor Dr. Farokh Bastani for his guidance, advice and patience during the work towards this thesis.

My sincere thanks are due to my committee members Dr. Ernst Leiss and Dr. Tiee-Jian Wu for their comments and criticism.

I also would like to thank Mr. Dar-Ren Leu for his many suggestions and help.

Byzantine Agreement—
Algorithms and Applications

An Abstract of a Thesis

Presented to

The Faculty of the Department of Computer Science

University of Houston–University Park

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Yi Zhao

December, 1987

ABSTRACT

All participating processes in a distributed system often have to reach agreement of some kind. The presence of malicious faults can cause a faulty process to send conflicting messages to different processes, making it difficult for nonfaulty processes to reach agreement. This problem, often called Byzantine agreement, Byzantine generals problem, or interactive consistency, is important in both the theory and practice of distributed computing and has been the subject of intense research in recent years.

This thesis presents a comprehensive study of this problem and its various applications. The Mostly Byzantine agreement is proposed as a less restrictive model for process-control applications. It can achieve a high degree of fault tolerance with a relatively low cost.

TABLE OF CONTENTS

CHAPTER 1

Introduction

Computer systems are used in a great variety of applications today. As more and more complex applications have been automated and we have become increasingly dependent on computing systems, the reliability of computers has become a major concern. At one extreme, a wrong output may cause minor inconvenience; at the other end of the spectrum, where human lives and/or vast sums of money may be at stake, no failures can be tolerated. Enhancing the reliability of computing systems is not a new topic; since the 1950's fault tolerance techniques have been adopted in order to achieve highly reliable hardware operation. Considerable effort has been devoted to the development of techniques to tolerate failures in hardware as well as software.

The motivation of this thesis is to study techniques to cope with a type of failure that is often overlooked—namely, a faulty component which sends conflicting information to different parts of the system. The problem of coping with this type of failure is known as the Byzantine generals problem, or Byzantine agreement. The major part of this thesis is devoted to a study of various algorithms to solve the Byzantine generals problem and the application of these algorithms to fault tolerance.

1.1 Fault Tolerance

Following the terminology suggested in [AND 82], the reliability of a system is usually characterized by a function which is the probability that no failure of the system would occur in a certain period of time. A failure of a system is an externally observed event that occurs when the behavior of the system deviates

from its specification. An erroneous state is an internal state which would lead to a failure. An error is part of an erroneous state and caused by faults which are the presence of defects in a component or in the design of the system.

The most straightforward way to construct reliable computing systems would be to use only reliable components, and put them together following only correct designs. In practice, however, one often has to try to achieve reliability despite the unreliability of the hardware and software components. So, this simple approach, aiming at fault avoidance and removal, can not guarantee that the overall system is free from faults and must be complemented by techniques of fault tolerance, which attempts to intervene and prevent occurring faults from causing system failures.

In the early days of computer development, hardware was a dominating factor both in the cost and complexity of a computer system. Early research in fault tolerance focussed on hardware faults, for example, circuit testing and fault diagnosis. Hardware faults can be transient or permanent. A transient fault exists in a system for only a limited period of time (less than some threshold) and disappears from that system spontaneously from the system after that period. A transient fault may be intermittent if it appears more than once. A permanent error is a fault which is present for a period longer than a threshold. Unlike a transient fault, which is usually due to physical deterioration, a permanent fault is often the result of some design or manufacturing errors, such as misunderstanding of specifications or incorrect connections of circuits.

As software has gained a dominating position, software faults have also become the major factor contributing to the failure of computing systems. Unlike hardware, software is not physical and not subject to deterioration with time. So software faults are usually permanent and are the result of design errors. Such design errors often arise from some exceptional cases where special attention is required. Sometimes it may not be possible to precisely classify whether a fault is a

hardware or a software one. For example, in some data structure oriented computer systems, the faults in that data structure can induce either hardware or software faults, or both. Often, a failure of such a kind of system can not be merely counted as a hardware or software fault; rather it is a combination of both faults.

The faults discussed here are assumed to be unmalicious in the sense that they are caused by some random process, either due to design errors or physical deterioration. Generally, a fault can be malicious with a purpose to prevent the system from meeting its specification. We will discuss those kinds of faults in later chapters.

All fault tolerant systems depend on the effective deployment and utilization of redundancy to compensate for the effect of faults in the system. By redundancy, we mean extra time or extra components which would not be required in a system free from faults. Time redundancy requires extra executions of the same computation, perhaps using different methods. Component redundancy involves the use of extra physical components, such as memory cells, bus lines, functional modules, etc., to provide the extra information needed to guard against failures. Generally, four phases can be identified for a fault tolerant implementation to prevent faults from causing to system failures. They are error detection, damage assessment, error recovery, and fault treatment and continued system services. Though most faults may not be directly detected, they can often introduce some detectable erroneous states. When an error is detected, the first step is to assess the extent to which the system state is damaged. It is desirable to limit the effect of fault to one area of the system, thus, preventing contamination of other areas. Error recovery attempts to transform the current erroneous system state into a well defined and error free state from which the system operation can continue. Techniques include masking of faults, retrying of operation reconfiguration of the system, and isolation of faulty components. In the fault treatment and continued service phase, techniques are used

to enable the system to continue to provide the services required by its specification, by ensuring that the fault is not repeated soon. For example, the system can start from the roll back point, the failed module can be replaced, etc.

Nearly all hardware fault tolerance techniques employ only component redundancy. Usually, hardware redundancy provides no recovery mechanism, rather it tries to mask the error. One frequently used method is to replace a single module by several identical or functionally equivalent modules. These modules are assumed to perform the same task and their outputs are sent to a voting circuit that would select the majority as the result. Errors can be detected as long as there is at least one properly functional component. Errors can be masked as long as a majority of them are correct. One classical example is the so called Triple Modular Redundancy (TMR), where three identical modules are provided to tolerate one failure. Another approach is to use structurally redundant components. One fine example is an error correcting code used in the memory system. In this scheme several redundant bits are used to summarize information about the data bits and a simple computation would indicate the existence and position of error in data bits.

Software fault tolerance techniques include both time and component redundancy. Error recovery is a kind of time redundancy. Recovery can be backward, restoring the computation to a prior correct state of the system regardless of the current erroneous state, or forward, manipulating some parts of the current state to produce a new one which would hopefully be error free. Recovery block is a well studied recovery scheme. A recovery block consists of a primary module, an acceptance test and a set of alternatives which are supposed to fulfill the same task as the primary module, but in a probably less efficient way. Before entering the block, a recovery point is set up to save all the state information that might be changed during the execution of the primary module or any of the alternatives. The primary module is executed first and the result directed to an acceptance test that would

check its correctness. If the result is acceptable, the whole block is finished. Otherwise the recovery point is restored and the first alternative is invoked, running like the primary module, and so on, until either the acceptance test is passed or all the alternatives have been tried. In the latter case, a failure is declared. An example of component redundancy is the N-version programming scheme. In this scheme, N different programs are developed to fulfill a single task. Each program can run independently (perhaps in parallel) and the results are compared to invoke appropriate actions.

1.2 Overview of the Thesis

The main theme of this thesis is to investigate techniques for tolerating arbitrary faults, especially malicious faults, in a distributed computing system. In such a system, every processor has its own memory and clock, and processors can communicate with each other through various communication lines. Some of the motivations to build distributed systems are to share resources, to speed up computation, to enhance reliability and to facilitate communication, etc.

While it is difficult to program a distributed system, to tolerate faults in such a system is even harder. Many techniques of fault tolerance in a distributed system assume a model in which a process fails only cleanly, by just terminating, without sending any bad data to corrupt other processes in the system. However, this assumption is not generally true. In general, the behavior of a faulty process is unpredictable. In the extreme case, it might intentionally subvert the effort of other processes. In this thesis, we are especially interested in Byzantine failure. In this fault model, the faulty process can send conflicting message to other processes, or unfaithfully relay other processes' messages.

In chapter 2, we will present a comprehensive survey of Byzantine agreement.

This problem serves as the theoretical basis of techniques to tolerate malicious faults. Suppose in a distributed system, every process has some initial value. The goal of Byzantine agreement is to enable all reliable processes to reach a decision in spite of the arbitrary malicious behavior by some faulty processes, such that all reliable processes should reach the same decision and if all reliable processes have the same initial value, they should all decide on that value. While easy to formulate, the problem is surprisingly hard to solve.

Chapter 3 presents a survey of applications of Byzantine agreement, such as clock synchronization, fail-stop processor, state machine, and distributed database system. Essential to these applications is reliable broadcasting, which can be easily implemented by using solutions to Byzantine agreement. We shall see that for some applications, Byzantine agreement presents an elegant and powerful solution, while for other applications like distributed data processing the power is limited.

In Chapter 4, we propose a variation of Byzantine agreement, namely the Mostly Byzantine agreement. Unlike Byzantine agreement, which requires reaching agreement every time, the Mostly Byzantine agreement does not. Rather, agreement is an overall property. In the Mostly Byzantine agreement, there can be only a finite number of disagreements in an infinite number of attempts to reach agreement. We present a general framework for Mostly Byzantine agreement using the techniques of self-stabilization and two algorithms with different convergence speeds.

Chapter 5 summarizes this thesis and outlines some research areas.

CHAPTER 2

Byzantine Agreement

2.1 Introduction

In a distributed system, each participating process often has to reach agreement of some kind. If all processes are reliable, or all faulty processes die cleanly, a simple round of message exchanges will enable all reliable processes to reach agreement. But when faulty processes can have arbitrary failures, such as sending conflicting messages to others, the problem gets very difficult. This problem, reaching agreement in the presence of malicious faults, is often called Byzantine agreement, Byzantine generals problem, or interactive consistency problem.

Imagine that several divisions of a Byzantine army, each commanded by its own general, are camped outside an enemy city. The Byzantine generals have to reach a common agreement on whether or not to attack the enemy city at dawn. It is crucial that all the generals agree since an attack by only some divisions would result in defeat. The generals can communicate with each other only by messengers running from camp to camp. Further, suppose that messengers will not be caught by the enemy or tell lies. (We shall see later why this assumption is reasonable.) Unfortunately, some generals are traitors, trying to prevent the loyal ones from reaching agreement by sending conflicting messages to other generals. This is exactly the situation we encounter in distributed systems.

To state the problem more formally, consider a set of directly connected processes $p_1, p_2, \cdots, p_n$, some of which may be faulty. Each process p_i has an initial value v_i. The goal is that, after a number of message exchanges, each process p_i has to decide one value d_i that satisfies [PEA 80]:

(1) *Agreement:* Every reliable process should decide on the same value, i.e., $d_i =$

d_j, for any two reliable processes p_i and p_j.

(2) *Validity:* If all reliable processes have the same initial value v, then for every reliable process p_i, $d_i = v$.

The messages used are two-party, non-authenticated, or oral messages, for which we make the following assumptions [LAM 82] [PEA 80] [DOL 82]:

A1: Every message that is sent is delivered correctly.

A2: The sender of a message can be identified.

A3: The absence of a message can be detected.

By A1, a faulty process can not change the message sent by another process. A2 makes it impossible for a faulty process to introduce spurious messages under the name of another process. The combination of A1 and A2 prevents a faulty process from interfering with the communication between two other processes. A3 ensures that a faulty process can not prevent agreement by simply not sending messages to some processes. We assume that links in the network are free from failures, since it is indistinguishable, in case of corruption or omission of a message, whether the sender or the link delivering the message is faulty. This is also the reason why we assume that the messengers for the Byzantine generals would not be caught by the enemy or tell lies.

In the discussion of Byzantine agreement, we will also use a slightly modified model of this problem. In this model, instead of each process having its own initial value, only one process, the commander, has a value. All other processes will obtain the value of the commander. The values obtained by processes must satisfy [LAM 82]:

(1) *Agreement:* Every reliable process should get the same value.

(2) *Validity:* If the commander is non-faulty, then all reliable processes should get the commander's value.

In this version of the problem, after sending its value to all the other processes,

the commander will not participate in the procedure any more. In the original version, no special assumption is made about the commander. In particular it could be a possibly unreliable data source that sends a value to each of the n processes in the system before the procedure begins and all processes will participate in the whole procedure. These two versions are essentially identical. For an algorithm for the latter one to solve the first version, we can repeatedly run the algorithm by letting each process act as the commander in turn, resulting in each reliable process having the same vector holding the private values of all the processes in the system. To solve the latter version by using a solution to the original version, we can simply add an initial round in which the commander sends its value to every other process.

In this chapter, we will present a comprehensive survey of various algorithms for Byzantine agreement. The algorithms we study are primarily for Byzantine agreement for binary values and in Section 2.5 we shall see how a binary value solution can be extended to the multivalue case. The processes in the system are supposed to be fully connected and the messages are non-authenticated and synchronized, unless explicitly stated. A discussion of partially connected networks, authenticated messages, and asynchronous communication can be found in Section 2.6, which will address some related topics of Byzantine agreement.

2.2 Impossibility Results

Here, we will discuss some impossibility results concerning the Byzantine agreement (generals) problem without formally proving them. Detailed proofs can be found in [FIS 82], [DOL 83] and [LAM 82].

The first result is about the redundancy required in the system. If we define redundancy as $(n-1)/t$, where n and t are the number of processes and the number of faulty processes respectively, this theorem states that a redundancy of at least 3

is required to reach Byzantine agreement.

Theorem 2.1. *Byzantine agreement can not be assured for n processes, of which at most t are faulty, if $n < 3t + 1$.*

Just to gain some intuitive feeling about the difficulty of Byzantine agreement and why at least a redundancy of 3 is required, consider the scenario in which there are three processes, A, B, and C, of which, say C is faulty. Maliciously, C is trying to prevent A's and B's effort to obtain consistency. In particular, C can send messages to A, suggesting that C's private value is 0 and B is faulty. Similarly, C's messages to B may suggest to B that C's private value is 1 and A is lying. If C lies consistently, A will not be able to tell whether B or C is faulty. To insure validity, A will thus have to select 0 as C's private value. A similar argument shows that B must select 1 as C's private value, thus defeating the agreement requirement. On the other hand, if the reliable processes are programmed to select one value in case of conflicting values, we can always construct scenarios to defeat validity. Thus, we have shown that there is no solution to the three generals, one traitor Byzantine generals problem. By reducing the $3t$ generals, t traitors Byzantine generals problem to the one traitor case, we can get the above result.

In synchronous systems, it is customary to measure time complexity in terms of rounds consisting of message exchanges and computation steps, such that during a round every process can communicate with all others in the system. The messages sent by a process in a given round can not depend on the messages it receives during the same round. An important result about the lower bound of complexity of any solutions to Byzantine agreement states that at least $t + 1$ rounds are required to reach agreement.

Theorem 2.2. *Byzantine agreement can not be assured within t or fewer rounds for n processes and t faulty ones, provided $n > t + 1$.*

Here, we require that there are at least two reliable processes, since it would be trivial to reach agreement when the number of reliable processes is fewer than two.

2.3 Deterministic Algorithms

In an early paper on Byzantine generals problem, Lamport, *et al.* [LAM 82] gave a recursive solution. The algorithm is not efficient in complexity and difficult to program; however, it served as an existence proof that agreement can be reached through $t+1$ rounds of message exchanges. The algorithm uses a majority function that will be used by each process to decide a value for a given set of values it receives from all processes in the system (including itself). For those sets that do not have a majority, a predefined default value is used instead. The algorithm $BG(t, c, L)$ (Fig. 2.1), which is essentially a solution to the modified version of the Byzantine agreement problem, would return to the caller the selected commander's value, given that t is bounded by the number of faulty processes, c is the commander, L is the set of processes excluding the commander, and $t \leq |L|$. During the process, only the initial call of $BG(t, c, L)$ implies that there are at most t faulty processes in the set $\{c\} + L$, and in the successive recursive call BG(t', c', L'), $t' < t$, there may be more than t' faulty processes in the set $\{c\} + L'$! Byzantine agreement can be obtained by the method discussed before, letting each process act as the commander in turn.

The recursive procedure $BG(t, c, L)$ involves $n-1$ separate executions of the procedure $BG(t-1, c', L')$, each of which invokes $n-2$ executions of $BG(t-2, c'', L'')$, in turn. Thus, a process may send many separate messages to each other processes. One way to distinguish among these different messages is for each process i to prefix its index i to each of the message it sends in step (2). As the recursion unfolds,

procedure $BG(t, c, L)$;
begin
(1) The commander c sends its value to every other process;
(2) **if** $t > 0$ **then**
for every process i other than c, let v_i be the value it receives from the commander, or $DEFAULT$ if it receives no value. Processes i acts as the commander c in the procedure $BG(t-1, i, L-\{i\})$, sending the value v_i to each process in $L-\{i\}$.
(3) For each process $i \in L$, let $V = \{v_1, \ldots, v_{c-1}, v_{c+1}, \ldots, v_n\}$, where each $v_j, j \neq i$, is the value process i receives from j in step (2) (using $BG(t-1, j, L-\{j\})$, if $t > 0$, or $V = \{v_i\}$, otherwise. In both case, v_i is the value process i receives directly from the commander c. Process i returns the majority of the set V as the decided value for $BG(t, c, L)$.
end;

Fig. 2.1 A recursive algorithm for Byzantine generals problem.

$BG(0, c, L)$ will be called $(n-1)(n-2)\cdots(n-t)$ times to send a value prefixed by a sequence of t processes' name, which can be encoded in length $O(\log t)$ bits. So the total number of message bits exchanged will be in the order of $(\log t)(n-1)(n-2)\cdots(n-t)$, which is exponential to t, the number of faulty processes. The correctness of the algorithm BG is established by Theorem 2.3.

Theorem 2.3. *For at most t faulty processes, algorithm $BG(t, c, L)$ solves the Byzantine generals problem, if $|L + \{c\}| > 3t$.*

To prove the theorem, we first prove the following two lemmas, which establish the agreement and validity requirements, respectively.

Lemma 2.1. *For any t and at most k faulty processes, $BG(t, c, L)$ satisfies the validity condition if $|L + \{c\}| > 2k + t$.*

PROOF. We will prove, by induction on t, that every reliable process will get commander c's value, if c is reliable. By condition A1, every process will get the value correctly from the commander, so validity is true for $t = 0$. We assume that it is true for $t-1$, and prove it for t.

In step (1), the reliable commander sends its value to all 3t other processes in the system. Every reliable processes will apply $BG(t-1, i, L')$, with $L' = L - \{i\}, |L'| = |L| - 1 > 2k + (t-1)$. By induction hypothesis, reliable process i would get $v_j = v$ for each reliable process j. Since there are at most k faulty processes, a majority of $2k+t$ values received would be equal to v. So, process i obtains $v = majority\{v_1, v_2, \ldots, v_{c-1}, v_{c+1}, \ldots, v_n\}$ in step (3). ∎

Lemma 2.2. *For at most t faulty processes, $BG(t, c, L)$ satisfies the agreement condition, if $|L + \{c\}| > 3t$.*

PROOF. If the commander c is reliable, Lemma 2.1 guarantees the validity condition; therefore the agreement is satisfied. So we just need a proof for the case that the commander is faulty. We prove it by induction on t. For $t = 0$, this is trivially true. We assume it is true for $t - 1$, and show it is also true for $t > 0$.

There are at most t faulty processes, and c is one of them, so there are at most $t - 1$ faulty processes in L. In step (2), each process i acts as the commander in $BG(t-1, i, L - \{i\})$. Since $|L - \{i\}| > 3t - 1 > 3(t-1)$, by induction hypothesis, every reliable process will get the same set $\{v_1, v_2, \ldots, v_{c-1}, v_{c+1}, \ldots, v_n\}$, and therefore, obtain the same majority value in step (3), proving the lemma in case of a faulty commander. ∎

To understand the algorithm better, we informally describe how the algorithm works from a non-recursive point of view. Essentially, the algorithm consists of two parts, expanding and shrinking the witness tree. The witness tree is expanded level by level, each level corresponding to one round of message exchanges. A witness tree for process p_i in a t faulty processes, n processes Byzantine generals problem is a tree which has one node at level 1, $n - 1$ nodes at level 2, $(n-1)(n-2)$ nodes

at level 3, $\cdots$, $(n-1)(n-2)\cdots(n-t)$ nodes at level $t+1$. Level 1 represents the messages that p_i gets at round 1. Let p_0 be the commander, and let all other processes be $p_1, \cdots, p_{n-1}$, each node in the witness tree for p_i can be labeled as $v_{0i_1 i_2 \ldots i_k}, 0 \leq k \leq t$, with v_0 being the root. $v_{0i_1 i_2 \ldots i_k}$ is interpreted as the value that p_{i_k} tells p_i that $p_{i_{k-1}}$ told p_{i_k} that $\cdots$ p_{i_1} told p_{i_2} that p_0 told p_{i_1}.

To build a witness tree, each process sends all the values at the leaf nodes of its current witness tree and expands its witness tree by one level using the messages it receives from all other processes. For example, the $v_{0i_1 i_2 \ldots i_k}$ node of p_i would be expanded by using the values of $v_{0i_1 i_2 \ldots i_k}$ nodes of all the p_j's such that $j \notin \{0, i_1, i_2, \ldots, i_k\}$, resulting in new nodes $v_{0i_1 i_2 \ldots i_k j}$'s. After the witness tree is built, each process can get its agreement value by shrinking the tree level by level from bottom to top until only one node is left, taking the value of that node as the agreement value. To shrink a witness tree of depth k, $k > 1$, the value of each $k-1$ level node is replaced by the majority of the values of that node and all its k level children which are also leaf nodes. Fig. 2.2 shows how 2 witness trees for reliable processes p_1 and p_2 might be expanded in a 2 faulty processes, 7 processes scenario in which the commander p_0 and process p_6 are faulty. Specifically, p_0 sends value 0 to odd numbered processes and value 1 to even numbered processes. p_6 lies to even numbered processes in round 2 and to odd numbered processes in round 3. Fig. 2.3 shows the shrinking of the witness trees constructed in Fig. 2.2. It also shows that agreement is actually satisfied since both p_1 and p_2 have the same set of values before applying the last majority function.

As we pointed before, the algorithm uses messages "witnessed" by up to t processes, with the combination of such t-witness messages in the order of exponential to t in total. So this method is very expensive in term of the message bits used, although it reaches the $t+1$ rounds lower bound for any solution to this problem. Also, it is very hard to program this algorithm even if we can devise a non-recursive

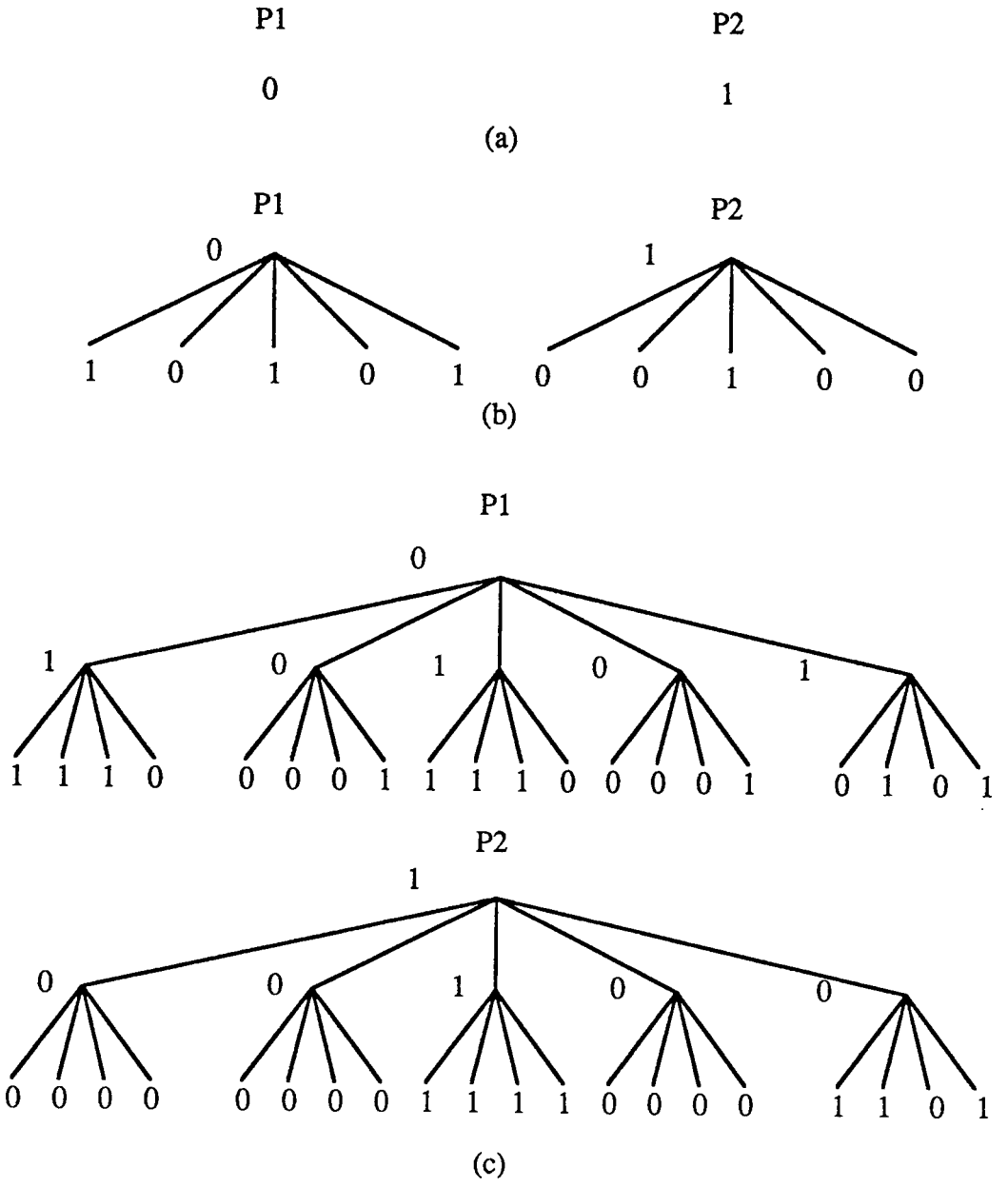

Fig. 2.2 The construction of witness tree for two processes (a) after the first round. (b) after the second round. (c) after the third round.

version. The reason lies in the fact that no structure is imposed on the information exchanged in the algorithm in which every process just simply broadcasts everything it receives, and applies certain decision functions to get the final result [LYN 82].

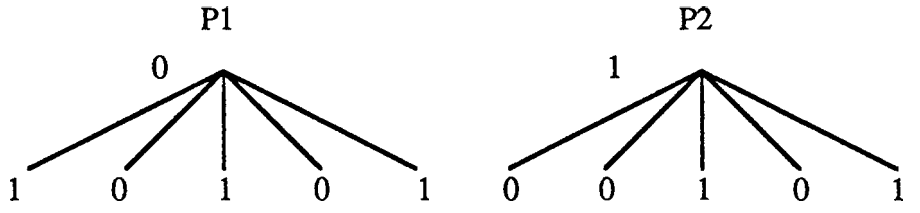

Fig. 2.3 The shrinking of the witness trees in Fig. 2.2 by one level. Note the difference with Fig. 2.2 (b).

Dolev and Strong [DOL 82b] gave the first algorithm that needs only a polynomial amount of message bits. By cleverly summarizing the information and sending only what is relevant, they got a solution using $4t + 4$ rounds and $O(n^4 \log n)$ message bits. The messages used can be thought of as 1-witness messages, since each process can confirm another process's initial value by consulting what other processes get from that process. Lynch *et al.* [LYN 82] developed the ideas of Dolev and Strong and gave another solution, using $2t + 4$ rounds and $O(tn + t^3 \log t)$ message bits. The algorithm presented below is taken from [LYN 82] with some minor modifications.

Let the number of processes be $n > 3t$, with each process having an index ranging from 1 to n. The only message items are process indices and a special value $'*'$, i.e., the set of message items is $I = \{'*', 1, 2, \ldots, n\}$. Messages are sets of message items, thus, each $M_i \in 2^I$. Also, we define $LOW = t + 1$, and $HIGH = 2t + 1$.

Each process p maintains the witness set $W_x(p)$ each for message item x, and the confirmed process set $C(p)$. $W_x(p)$ is defined as:

$$W_x(p) = \{j \mid p \textit{ has received message item } x \textit{ from process } j\}$$

Letting $w_x(p) = |W_x(p)|$, we define

$$C(p) = \{k \mid w_k(p) \geq HIGH\}$$

and $c(p) = |C(p)|$.

The algorithm (Fig. 2.4) is organized in $t + 2$ epochs. Trivially, we suppose that there is an epoch 0 in which each process gets its initial value. Each epoch consists of two rounds of message exchanges. In the first round, a process will send a '*' to every other process (including itself) if it initiates in that epoch. A process p initiates

I1. in epoch 0, if it has initial value 1, or

I2. in epoch $e > 0$, if it has $c_{e-1}(p) \geq LOW + e - 1$, and has not initiated before.

Here, we use $W_{x,e}(p), C_e(p)$ to refer to the set $W_x(p), C(p)$ at the end of epoch e. In the second round, a process will broadcast the index i if it witnesses (directly or indirectly) the initialization of process i. A process p commits in epoch e, if $c_e(p) \geq HIGH$, agreeing on value 1.

```
procedure IC(t, p);
begin
  for e := 0 to t + 2 do
  begin
round1:
          broadcast('*') if p initiates in epoch e;
          receive '*'s, if any;
          W_*(p) := {j | p receives '*' from j in epoch e};
round2:
          for x := 1 to n do
            if x ∈ W_*(p) or W_x(p) ≥ LOW then broadcast(x);
          receive messages;
          for every message item received do
            if x is sent by process j then W_x(p) := W_x(p) + {j};
          for x := 1 to n do
            if w_x(p) ≥ HIGH then C(p) := C(p) + {x}
  end;
  if c(p) ≥ HIGH then ⟨decide v_p = 1⟩
  else ⟨decide v_p = 0⟩
end;
```

Fig. 2.4 Algorithm using polynomial message bits and $2t + 4$ rounds.

Theorem 2.4. *Algorithm IC solves the Byzantine agreement problem, if* $n > 3t$.

To prove the theorem, we prove the following lemmas which establish the cor-

rectness of the algorithm. All the lemmas refer to the computation of at most t faulty process and $n > 3t$ processes. We denote T as the set of all reliable processes.

The first lemma states that $C(p)$ and $W_x(p)$ increase monotonically.

Lemma 2.3. *Let $0 < e \leq e' \leq t+2$, then $C_e(p) \subseteq C'_e(p)$, and $W_{x,e}(p) \subseteq W_{x,e'}(p)$, for all x, p.*

PROOF. The proof is trivial, since in the algorithm, no statement takes any element away from $C(p)$ and $W_x(p)$. ∎

Lemma 2.4 says that whenever a reliable process initiates, it is confirmed by all reliable processes one epoch later.

Lemma 2.4. *Let $i \in T$, if i initiates in epoch e, $0 \leq e < t = 2$, then $i \in C_{e+1}(j)$, for all $j \in T$.*

PROOF. Since i initiates in e, it announces its initialization in the first round of epoch $e+1$, and all reliable processes will witness its initialization by broadcasting i to every other process in the second round. Thus any $j \in T$ will get message i from all the reliable processes, so $|W_i(j)| \geq T \geq 2t+1$, which implies that $i \in C_{e+1}(j)$. ∎

The next lemma shows whenever all reliable processes initiate, all reliable processes will commit one epoch later.

Lemma 2.5. *For $0 \leq e < t+2$, if all $i \in T$ initiate in epoch e, then all $j \in T$ commit in epoch $e+1$.*

PROOF. For any $j \in T$, by Lemma 2.4, $i \in C_{e+1}(j)$ for any $i \in T$. The lemma follows since $|C_{e+1}(j)| \geq |T| \geq HIGH$. ∎

The following lemma says that whenever a process is confirmed by another reliable process, it will be confirmed by all the reliable processes one epoch later.

Lemma 2.6. *For $0 < e < t+2, i, j \in T$, if $k \in C_e(i)$, then $k \in C_{e+1}(j)$.*

PROOF. Since $k \in C_e(i)$, it has been witnessed by at least $HIGH$ processes, at least LOW of which are reliable. So every reliable process has seen at least LOW witnesses of k. By the algorithm, every reliable process will broadcast k in the second round of epoch $e+1$, so $|W_{k,e+1}(j)| \geq |T| \geq HIGH$, and the lemma follows. ∎

Next, we show that when a reliable process initiates in some epoch e, $e > 0$, then all the reliable processes will get initiated one epoch later.

Lemma 2.7. *For $0 < e < t+2$, if $i \in T$ initiates in epoch e, then all $j \in T$ will initiate in epoch $e+1$, provided they have not initiated before.*

PROOF. Since i initiates in epoch $e, e > 0$, it can only be initiated by rule I2, so $c_e(i) \geq LOW + e - 1$. By Lemma 2.6, $C_e(i) \subseteq C_{e+1}(j)$, for all $j \in T$, and by Lemma 2.4, $i \in C_{e+1}(j)$. Since i had not initiated before, so no reliable process has ever witnessed it, i.e., $w_{i,e}(h) < LOW$, for any $h \in T$, which implies that $i \notin C_e(h)$. In particular, $i \notin C_e(i)$. So $c_{e+1}(j) = |C_e(i) + \{i\}| = c_e(i) + 1 \geq LOW + e$. By I2, j initiates in epoch $e+1$, if it has not initiated before. ∎

The following lemmas form the backbone for Theorem 2.4. They show that as soon as LOW reliable processes have initiated, an avalanche of initialization begins which result in all reliable processes initiating and committing two epochs later.

Lemma 2.8. *For $0 < e < t = 2$, if there is a set of reliable processes A, $|A| = LOW$, such that all $i \in A$ has initiated at the end of epoch e, then all $j \in T$ will get initiated at the end of epoch $e+1$.*

PROOF. Let e' be the least number epoch such that all $i \in A$ have initiated at the end of epoch e', we show that all $j \in T$ will get initiated at the end

of epoch $e' + 1$. If $e' = 0$, the $c_1(j) \geq |A| = LOW + 1 - 1$, thus all j get initiated in epoch 1. If $e' > 0$, then at least one $i \in T$ gets initiated in e', but not in $e' - 1$. By Lemma 2.7, all the non-initiated reliable processes will initiate in epoch $e' + 1$, proving the lemma. ∎

Lemma 2.9. *For $0 \leq e \leq t$, if there is a set A, $A \subset T$ and $|A| = LOW$, such that all $i \in A$ have initiated at the end of epoch e, then all $j \in T$ commit at the end of epoch $e + 2$.*

PROOF. By Lemma 2.8, all $j \in T$ will be initiated at the end of epoch $e + 1$, thus all $j \in T$ commits at the end of $e + 2$, following lemma 2.4. ∎

Now, we can prove Theorem 2.4 by showing that algorithm IC satisfies agreement and validity properties.

Lemma 2.10. *Either all $i \in T$ commit in epoch $t + 2$, or no $i \in T$ commits in epoch $t + 2$.*

PROOF. If there is a set A, $A \subset T$ and $|A| = LOW$, such that all $j \in A$ get initiated at the end of epoch t, then all $i \in T$ would commit at the end of epoch $t + 2$, by Lemma 2.9. If there is no such a set, then no $k \in T$, not initiated before, would initiate in epoch $t+1$, or $t+2$, since $c(k) < t+LOW$ in these two rounds, which implies that the confirmed reliable processes are at most t. Thus, for any $i \in T$, $c_{t+2}(i) \leq 2t < HIGH$, which means that it is impossible for i to commit. ∎

Lemma 2.11. *Let $i \in T$,*

(a) if all $j \in T$ has initial value 0, then $v_i = 0$.

(b) if all $j \in T$ has initial value 1, then $v_i = 1$.

PROOF. (a) We prove by induction on e that no $j \in T$ will ever initiate in epoch e. This is trivially true for $e = 0$. Now, suppose that no $j \in T$

initiates in epoch e or less and there is some $k \in T$ initiates in epoch $e+1$, then, by I2, $c_{e+1}(k) \geq LOW$. It follows that there is at least one $h \in T$, such that $w_{h,e+1}(k) \geq HIGH$, this can happen only if h get initiated in epoch e or before, a contradiction.

(b) All reliable processes initiate in epoch 0, thus all $i \in T$ commit at the end of epoch 1, by Lemma 2.5. ∎

Thus, we have shown that after $t + 2$ epochs, Byzantine agreement can be assured. The following theorem states that the bound of $t + 2$ epochs is tight for this algorithm.

Theorem 2.5. *The Byzantine agreement can not be assured by IC through fewer than $t + 2$ epochs.*

PROOF. Scenarios can be constructed to show that faulty processes maliciously send conflicting messages to let some, but not all, reliable processes commit in epochs ranging from 1 to $t + 1$. The construction is tedious, so we omit it here. ∎

Since $|I| = n + 1$, each message item can be coded by $O(\log n)$ bits. In this algorithm, each process sends at most n messages in each epoch, and each message contains potentially, but no more than, $n+1$ message items. So totally, there would be $O(n^2(t+2)(n+1)\log n) = O(t^4 \log t)$ message bits. Compared with the original algorithm by Lynch, the algorithm presented here would save half of the message bits. On the other hand, if we make a process i send a message item at most once to other processes, both algorithm would use the same amount of message bits, that is $O(n^2(n+1)\log n) = O(t^3 \log t)$.

The initialization, or the initial value of a reliable process is actually witnessed by the other processes once, so this algorithm can be thought of as a 1-witness algorithm. Recall that the recursive algorithm uses $t + 1$ rounds of message ex-

changes and exponential message bits. A question arises concerning the relations between the message structure, the number of rounds, and the amount of message bits. Another question is whether $t + 2$ epochs, or $2t + 4$ rounds is an optimum value for an algorithm using 1-witness messages. We, as well as other researchers, have been trying, unsuccessfully, to reduce the number of rounds, hoping to remove the factor 2.

2.4 Randomized algorithms

We have seen that the number of rounds used to reach agreement in deterministic algorithms is obtained based on the worst case analysis, while the average expected number of of rounds is appreciably less. The reason is that the worst case analysis presupposes a particular probability distribution on the space of instances of a problem, while the relative frequency of instances of such problem may be changing in an appreciable and unmanageable way [RAB 76] [KAR 77]. The sample of instances actually appearing in a given application is often statistically biased in a manner not confirming to the assumption made in the analysis of our algorithms. In contrast to deterministic algorithms, randomized algorithms do not assume anything about the distribution of the instances of the problem to be solved. Rather, they incorporate randomization into the algorithms.

In a randomized Byzantine agreement algorithm, each process can use randomly chosen numbers in the course of reaching agreement. The property and efficacy of those randomly chosen numbers do not depend on the random behavior assumption of the faulty processes [RAB 83]. Usually, there are two kinds of such algorithms. One is to achieve Byzantine agreement with a small probability of error. More precisely, given $\varepsilon > 0$, we say that a randomized algorithm is a $1 - \varepsilon$ reliable Byzantine agreement algorithm, if the reliable processes will reach

Byzantine agreement with probability of at least $1 - \epsilon$. Another way is to reach Byzantine agreement without error after an expected number, some constant c, of rounds of message exchanges.

In Ben-Or's algorithm [BEN 83], each process tosses a coin independently until a large number of individual outcomes coincide. This algorithm works well when the redundancy is high but when the redundancy is low, the number of rounds and message bits may be exponential to n, the number of processes in the system.

Rabin [RAB 83] produces a global coin toss using Shamir's technique for sharing secrets [SHA 79]. In Shamir's method of sharing a secret S among n participants, every k or more cooperating participants can reconstruct S, but fewer than k participants can not do so. In this scheme, before the Byzantine agreement algorithm is first started, a trusted dealer, D, a reliable process, would randomly and independently choose a sequence of bits, $b^1, b^2, \ldots, b^m$. The dealer computes n pieces secret item $p_1^m, p_2^m, \ldots, p_n^m$, using b^m, authenticates all the p_i^m with digital signatures, and then distributes the secret item, s_i^m, the digitally signed p_i^m, to process p_i. During the m-th lottery, $Lottery(m)$, each p_i requests $2t$ other processes' secrets, s_j^m, and at least t of them will answer. Using these (at least) $t+1$ pieces of secret, p_i can reveal the secret b^m. Fig. 2.5 is an algorithm given by Perry [PER 85], which is essentially the same as Rabin's except that it does not use authenticated communication.

In the algorithm, process p_i invokes $Exchange(R)$, resulting in $message_i^{R-1}$ being sent to all processes. $c = \lceil (n+2t)/2 \rceil$ for $n \geq 6t+1$ asynchronous processes, or $c = \lceil (n+t)/2 \rceil$ for $n \geq 3t+1$ synchronous processes. The $Lottery(R)$ procedure enables process i to get to know the R-th secret $secret_i^R$. Invoking $Lottery(R)$ after $Exchange(R)$ guarantees that s^R is not revealed prematurely, even in the asynchronous case. $LastRound$ is determined by a concurrently running procedure $CloseFinish$ which detects whether agreement has been reached in a certain round. $message_i^{LastRound}$ will be the decision value for process i. The algorithm can reach

```
procedure Decision(R);
begin
  if Count_i^R(0) > Count_i^R(1)
    then majority := 0;
    else majority := 1;
  if Count_i^R(majority) > c
    then message_i^R := majority;
    else message_i^R := secret_i^R
end;

procedure RandomBA1;
begin
  R := 0;
  while (R ≠ LastRound) do
  begin
    R := R + 1;
    Exchange(R);
    Lottery(R);
    Decision(R)
  end
end;
```

Fig. 2.5 A randomized Byzantine agreement algorithm that reach agreement with probability $1 - 2^{-R}$ in R rounds.

agreement in R rounds with probability $1 - 2^{-R}$. If all the reliable processes have the same initial value, the algorithm terminates in the first round with $message_i^1$ being equal to that value for all reliable process i. For a detailed proof, see [PEA 85].

Chor and Coan [CHO 85] gave another randomized algorithm which terminates in an average $O(t/logn)$ rounds for $n \geq 3t + 1$ synchronous processes (Fig. 2.6). In their algorithm, a small group of g processes is assigned the task of coin tossing at each epoch. Thus, it differs from Rabin and Perry's centralized tossing by a trusted dealer. Each process in this selected group tosses its own coin and broadcasts its result. Every process takes the majority of those tosses as the correct coin toss. If more than half of the tossing processes are faulty, they can bias the coin toss maliciously. However, if fewer than half are faulty, all the reliable processes in that group can produce the same toss with a sufficiently high probability if the group size, g, is relatively small, which enables all reliable processes to reach agreement

```
procedure RandomBG2;
begin
  current := init_value;
  for e := 1 to ∞ do
  begin
round1 :
          broadcast message (e, 1, current)
          receive message (e, 1, *)
          if for some v, Count(e, 1, v) ≥ n − t
            then current := v;
            else current :='?';
          if Group(p) = e mod ⌊n/g⌋
            then Toss := Toss_Coin();
            else current :='?';
round2 :
          broadcast message (e, 2, current, toss);
          receive message (e, 2, *, *);
          ANS := value v ≠'?' such that Count(e, 2, v, *) is largest;
          num := Count(e, 2, ANS, *);
          if num ≥ n − t then ⟨decide v_p = ANS⟩
          else if num ≥ t + 1 then current := ANS
          else current := majority tosses from the group e mod ⌊n/g⌋
  end
end;
```

Fig. 2.6 A randomized Byzantine agreement algorithm with an average $O(t/\log n)$*rounds*.

in one more round.

In this algorithm, if all the reliable processes have the same initial value, they would decide on that value in the second round of first epoch. The algorithm certainly would terminate since for any epoch e, there is at least one value which would cause all the reliable processes to decide in the second round of epoch $e+1$. Agreement is guaranteed by the fact that if one reliable process decides on a value in epoch e, all other reliable processes would decide on that value in epoch $e+1$. This also makes it possible to detect the termination of the algorithm. If the process failures are uniformly distributed and the size of a group is 1, agreement is expected to be reached in 4 epochs, or 8 rounds.

Both Rabin/Perry's and Chor's algorithms use only $O(n^2)$ message bits.

2.5 Multivalue Byzantine Agreement

Except for the recursive algorithm (Fig. 2.1), all the algorithms presented so far can only solve binary valued Byzantine agreement problem. In this section, we show how to extend the binary valued solution to a multivalued one.

The simplest way to do so is to encode the original data in a binary bit string and run multiple copies of the binary solution in parallel, as suggested in [LYN 82]. While theoretically sound, this approach suffers from the great amount of message bits exchanged (b times of what a binary solution has, where b is the length of the binary string). Further, it may introduce undefined values in case not all strings of a certain length are meaningful. This is especially true in some real time environments.

In [TUR 84] and [PER 85], a clever method is used in which the binary solution is extended to a multiple one by prepending two rounds of message exchanges. The prepended two rounds will produce, before the start of the binary Byzantine agreement algorithm, a binary value for each process that serves as the initial binary value for that process. Fig. 2.7 is a solution, for $n \geq 3t + 1$, proposed by Turin and Coan. It is very similar to that of Perry. In the algorithm, $BinaryBA$ is any binary Byzantine agreement solution with $Alert_i$ as the initial value and $Final_i$ as the decision value for process p_i.

It is not hard to see that if all the reliable processes have the same initial value, then none of them will get perplexed, i.e., $Perplexed_i = 0$ for every reliable process p_i. So all their $Alert_i$'s are 0, thus all the $Final_i$'s, making them decide on that value.

So, what we need to show is that in case every reliable process p_i gets $Final_i = 0$, all the reliable processes will get the same majority value. In this case, at least one reliable process has $Alert_i = 0$, which implies that there are at least $2t + 1$ unperplexed processes for process p_i, the majority of them, at least $t + 1$ processes, is reliable. These unperplexed reliable reliable processes are also in the

```
procedure MultiBA;
begin
round1 :
        broadcast message (Init_i);
        receive messages (Init_*);
        S_i := {j|Init_i ≠ Init_j};
        if |S_i| > ⌈(n − t)/2⌉ then Perplexed_i := 1;
        else Perplexed_i := 0;
round2 :
        broadcast message (Perplexed_i);
        receive messages (Perplexed_*);
        P_i := {j|Perplexed_j = 1};
        if |P_i| ≥ n − 2t then Alert_i := 1
        else Alert_i := 0;
  BinaryBA;
  if Final_i = 0 then ⟨decide v_i = majority({j|j ∉ P_i})⟩
  else ⟨decide v_i =' default value'⟩
end;
```

Fig. 2.7 A multivalue Byzantine agreement solution for process p_i.

unperplexed set of every other reliable process. So the majority of every reliable process's unperplexed process set are reliable. We claim that all those unperplexed processes have the same initial value. Thus every reliable process will get the same decision value when some majority function is applied to the values of processes in its unperplexed set.

To see why all the unperplexed reliable processes have the same initial value, consider a set A of reliable processes, such that all the processes in A have the same initial value and all reliable processes having that value belong to A. $|A|$ must be greater than $(n-t)/2$, since otherwise all the processes in A are perplexed, and so is every process not in A, which is a contradiction to the fact that there are at least $t+1$ unperplexed reliable processes. Further, suppose that a reliable process $j \notin A$ is unperplexed and has a different value. But this is impossible since $|A| > (n-t)/2$, which implies that j is perplexed! Thus all unperplexed reliable processes belong to A.

2.6 Related Topics

2.6.1 Authenticated Message. If in addition to A1–A3, every message must be signed before it is sent, we add the following

A4 (a): A reliable process's signature can not be forged, and any alteration of the contents of its signed message can be detected.

(b): Any process can verify the authenticity of another process's signature.

A4 restricts a faulty process's ability to lie by making a reliable process's signature unforgeable by a faulty process. Thus, a message signed by a reliable process may either be relayed correctly or absorbed by a faulty process. Here, we make no assumption about a faulty process's message. In particular, a faulty process's message can be altered and signature be forged.

With authenticated messages, it it not necessary to have at least $3t+1$ processes to cope with t faulty processes. Actually, no requirement is needed on the number of processes in the system. But generally, we still assume that at least $t+2$ processes are required in case of t faulty processes, since it is trivial when there are fewer than $t+2$ processes. But $t+1$ rounds are still required to assure agreement [DOL 82b]. Fig. 2.8 is an algorithm given in [LAM 82]. Here, we assume that processes are named by $0, 1, \ldots\ n$, and process 0 acts as the commander.

The validity property is obvious, since no other message is received by reliable processes. For reliable processes i and j, $i \neq j$, we have $V_i = V_j$, since every $v \notin V_i$ will be sent to i by j, and vice versa; so the agreement holds.

2.6.2 Asynchronous Byzantine Agreement. So far, we have been mainly concerned with synchronous Byzantine agreement. In this model, it is assumed that there is a finite bounded delay on the operation of the processes and their intercommunications. Thus, unannounced process death as well as long delay are considered to be faults.

In asynchronous systems, this assumption no longer holds since in such systems,

```
procedure AuthenticatedBG;
begin
  V_i := φ;
  while more_messages do
  begin
    receive message (m);
    if m is of the form v : 0 then
    begin
      V_i := {v};
      send v : 0 : i to every processes other than o and i
    end
    else
      if m is of the form v : 0 : j_1 : ... : j_k and v ∉ V_i then
      begin
        V_i := V_i + {v};
        if k < t then send v : 0 : j_1 : ... : j_k : i to every processes
          other than 0, j_1, ..., j_k, i
      end
  end
  (decide v_i = majority(V_i))
end
```

Fig. 2.8 Algorithm using authenticated messages.

a very slow process can not be distinguished from a dead process. Our algorithms, except for that of Rabin/Perry with $c = \lceil (n+2t)/2 \rceil$ for $n \geq 6t+1$, do not work with asynchronous systems. The reason is that there is no predefined finite bound on the time of message exchange due to the uncoordinated progress of each individual process. If there are t faulty processes, each process can only be guaranteed to receive $n-t$ messages. If some predefined default value is used for the slow, possibly dead, processes, even some reliable processes may behave inconsistently to other reliable processes, making the problem even harder. It is shown in [DOL 86] and [FIS 85] that no deterministic algorithm exists to assure exact agreement in a finite number of rounds, even with only one faulty process. However, exact agreement can be reached with any given probability $p < 1$ after a finite number of rounds [RAB 83] [BRU 85] [PER 85]. Another kind of agreement, approximate agreement, can also be reached after a finite number of rounds.

2.6.3 Weak Byzantine Agreement. If, in the second version of Byzantine agree-

ment, we replace the validity requirement by:

Weak Validity: If all the processes are reliable, then every process obtains the value of the commander.

We get a weaker version of the Byzantine generals problem [LAM 83]. An instance of the weak Byzantine generals problem is the transaction commit problem for a distributed database system. In such a system, the transaction coordinator acts as the commander and the all the participating sites act as the generals. The transaction coordinator issues a command indicating whether a transaction should be committed or aborted. To achieve consistency among the participating sites, all sites should agree on the whether to commit or abort the transaction (agreement), but the failure of any site is allowed to abort the transaction (weak validity).

Lamport has shown that at least 3 redundancy is required to reach weak Byzantine agreement. A solution to an ordinary Byzantine agreement certainly solves the weak one. But the weaker validity requirement suggests that simpler and less costly algorithms may exist.

2.6.4 Approximate Agreement. If the set of values V is not finite, but instead is drawn from an interval of real space, such as in clock synchronization, it is more desirable to reach approximate agreement satisfying the following requirements [DOL 86]:

(1) *agreement:* All non-faulty processes eventually halt with output values that are within ε of each other.

(2) *validity:* The values output by each reliable process must be in the range of the initial values of the reliable processes.

Solutions to approximate Byzantine agreement usually consist of a series of message exchanges. After each round of exchange, each process produces a new message item for the next round, hoping that the new message item will be some-

what "closer" to the desired value. The number of rounds is determined by the initial range, the desired ranged, and the way to produce new items. The key to those solutions is the way to generate new items. In [DOL 86], an averaging function $f_{k,t}(V)$ is applied to V, the set of message items received by a process during some round, to generate the next message. Given a set V of real numbers, some of which may be identical, $f_{k,t}(V)$ is obtained by first eliminating the highest and lowest t elements, resulting in a set U, $|U| = m = |V| - 2t$, and $u_0 \leq u_1 \leq \ldots \leq u_{m-1}$ are the elements of U, then taking the mean value of the smallest element of U and every kth element thereafter, i.e., $f_{k,t}(V) = (u_0 + u_k + \ldots + u_{jk})/c$, where $c = \lfloor (m-1)/k \rfloor + 1$ and $j = c - 1$. For example, let $V = \{2,2,3,4,5,6,7,8,8,9\}$, $f_{2,2}(V) = (3 + 5 + 5)/3 = 5$, where $U = \{3,4,5,6,7,8\}$, $c = 3$, and $j = 2$.

```
procedure ApproximateBA(t, ε);
begin
  broadcast message (Init_i);
  receive messages (Init_*) into V_i;
  v_i := f_{t,t}(V_i);
  c := ⌊(n − 2t − 1)/t⌋ + 1;
  H_i := ⌊log_c(max(V_i) − min(V_i))/ε⌋ + 1;
  for h := 2 to H_i do
  begin
    broadcast message (v_i);
    receive messages (v_*) into V_i;
    v_i := f_{t,t}(V_i);
  end
  broadcast message (⟨v_i, halt⟩);
  output (v_i);
end
```

Fig. 2.9 An algorithm for approximate agreement in synchronous system with $n \geq 3t + 1$.

The algorithm of Dolev *et al.* is shown in Fig. 2.9. H_i is the process p_i's estimated number of rounds to reach agreement. For those processes terminating earlier, having broadcasted $\langle v, \text{halted} \rangle$ to other processes, the terminating value v will be used as the message broadcasted in the rounds after termination. A similar algorithm for the asynchronous case is also given for $n \geq 5t + 1$, except that no

predefined timeout is used. Each process stops collecting values when it has received messages from $n - t$ processes, and the averaging function is $f_{2t,t}$.

2.6.5 Early Stopping of Algorithms. In deterministic algorithms, committing happens like an avalanche, i.e., if one reliable process commits, all other reliable processes will commit one or two rounds later. This makes it possible to detect whether agreement has been reached and stop the algorithm earlier. This is especially helpful when agreement can be obtained through one or two rounds, e.g., in the case of a reliable commander, and will save significantly compared with the rounds needed for the worst case. It is also very useful in randomized algorithms to detect the termination of the algorithm. For example, Rabin [RAB 83] uses a procedure *CloseFinish*, running in parallel with the agreement procedure, to detect whether agreement has been reached and dynamically set the *LastRound* parameter to stop the algorithm.

The implementation is not complicated. In algorithm *IC* (Fig. 2.4), we can add a third round to each epoch, in which each process p will send a message "I have committed" to other processes, if $c(p) \geq HIGH$. When a process receives $t + 1$ copies of "I have committed", it implies that at least one reliable process has committed. So that process can stop in the next epoch, knowing that agreement will be reached by that time.

2.6.6 Missing Communication Paths. So far, we have been assuming that each process can directly communicate with all the other processes, i.e., the corresponding graph is fully connected. This is not necessary, and the algorithms can be extended to more general cases.

Lamport *et al.* [LAM 82] extended their algorithm to the so called p-regular network. In such a network, each node i has a regular set of neighbors consisting of p distinctive nodes, such that for a node k other than i, there exist paths, $p_{j,k}$,

where j is in the regular set of i, not passing through i, and two different such paths have no node in common other than j and k. For t faulty processes and $n \geq 3t + 1$ processes, Lamport *et al.* define a $3t$-regular graph, select $3t + 1$ nodes including the commander and run the ordinary algorithm. After the ordinary algorithm stops, every non-participating process can get $2t + 1$ copies of the decision from those participants, and a simple majority function will give the correct answer. But when $n = 3t + 1$, this is just the ordinary algorithm.

Dolev [DOL 82] has shown that agreement is achievable if and only if the connectivity of the graph is $2t + 1$, in the case of t faulty processes. In graph theory, Menger's theorem [HAR 72] states that if the connectivity of a graph is k, then there exist k disjoining paths between every pair of nodes. Thus, a process i can let its value be known to process j by sending its value along $2t + 1$ disjoining paths between i and j. Since at most t paths may have faulty processes, the number of copies that may be be corrupted is at most t, and a majority of those received value should reveal the real value of i. The implementation of this method requires that each process know in advance the network topology and transmit route information along with the values being sent.

CHAPTER 3

Applications of Byzantine Agreement

3.1 Clock Synchronization

Keeping the logical times of processes in a distributed system close enough in the presence of arbitrary faults is crucial in many situations. This task, clock synchronization, was one of the first initial motivations in proposing the Byzantine agreement. During the development of the SIFT fault-tolerant computer system, researchers at SRI first assumed that a simple majority voting scheme could be devised to treat such situations as clock synchronization, stabilization of input from sensors, and agreement on results. Gradually, they realized that simple majority voting is not enough, leading to the formulation of the interactive consistency problem [PEA 80].

Almost all the proposed clock synchronization algorithms run in rounds, resynchronizing periodically to prevent clocks from drifting too far away. However, these algorithms differ from each other in the dependency of the closeness of synchronization and size of adjustment on the number of processes and faulty processes, the authentication of message used, and the number of message bits exchanged.

Here, we present a simple and straightforward implementation. This algorithm shares the same spirit with the one by Lundelius and Lynch [LUN 84]. In fact, we have directly used some lemmas in their proof. The one presented here can be considered as a special case of the model in [LUN 84], but we think it fits with most of today's digital clocks. The logical clock is assumed to be ρ-bounded, i.e., over time period T, the difference between the logical clock and the real clock (a conceptual one for comparison) is at most ρt, for a very small constant ρ. Unlike the approach in [LUN 84], where the logical clock is the sum of the value of a

read-only physical clock and a corrective value, we do not distinguish between the logical clock and the physical clock. The clock here is just like a quartz oscillator controlled counter. The counter increases by one with each pulse of the oscillator. The oscillating frequency would guarantee the ρ-bounded property of the clock. A process can either read the value of its logical clock or adjust it by adding a corrective value. The adjustment would not change the ρ-bounded property.

In the rest of this section, we will represent real time in lower case letters and logical times in upper case letters. Also, we use upper case letters to denote mappings from real time to logical time and user lower case letters for the reverse mappings. Thus, the ρ-bounded property can be represented as

$$(1-\rho)(t_2-t_1) \leq 1/(1+\rho)(t_2-t_1) \leq C(t_2-t_1)$$
$$\leq (1+\rho)(t_2-t_1) \leq 1/(1-\rho)(t_2-t_1).$$

The reverse also holds

$$(1-\rho)(T_2-T_1) \leq 1/(1+\rho)(T_2-T_1) \leq c(T_2-T_1)$$
$$\leq (1+\rho)(T_2-T_1) \leq 1/(1-\rho)(T_2-T_1).$$

Suppose in a distributed system, there are n fully connected processes of which there are at most t faulty processes, where $n \geq 3t+1$. Each process maintains its own logic clock which is supposed to be ρ-bounded. Processes can communicate via messages over reliable connections. The message delay for every message is in the range of $[\delta-\varepsilon, \delta+\varepsilon]$, for some nonnegative constants δ and ε with $\delta > \varepsilon$. All the reliable processes are synchronized enough at the initial time. Suppose for each process p the initial time is T^0 on its own clock C_p^0 and the corresponding real time is t_p^0, then the initial synchronization can be represented as $|t_p^0 - t_q^0| \leq \beta$, for some fixed β and all nonfaulty p and q. Let

$$tmax^0 = max\{t_p^0 \mid p \; is \; nonfaulty\}$$

and

$$tmin^0 = min\{t_p^0 \mid p \text{ is } nonfaulty\}.$$

The object of the clock synchronization is to satisfy the following two requirements.

1. γ-Agreement: all the nonfaulty processes are synchronized within γ, i.e., for all $t \geq tmin$ and all nonfaulty p and q.

$$|C_p(t) - C_q(t)| \leq \gamma.$$

2. $(\alpha_1, \alpha_2, \alpha_3)$-validity: the logical time of a nonfaulty process increases in some relation to real time, i.e.,

$$\alpha_1(t - tmax^0) + T^0 - \alpha_3 \leq C_p(t) \leq \alpha_2(t - tmin^0) + T^0 + \alpha_3.$$

The algorithm given in Fig. 3.1 executes in rounds in a time-driven manner. In the algorithm $T^{i+1} = T^i + P$, and $U^i = T^i + (1 + \rho)(\beta + \delta + \varepsilon)$, where P is the period we are trying to determine. A process enters the ith round when it is triggered by its logical clock time reaching the value T^i. By selecting a proper value P, all the logical clocks can reach T^i within real time β of each other, i.e., $|t_p^i - t_q^i| \leq \beta$, for reliable processes p and q. When a process p reaches T^i, it is interrupted to broadcast a T^i message which indicates that p has reached T^i on its own logical clock. At the same time, it sets another timer to interrupt it at time U^i on its own logical time. During the time period from T^i to U^i, p collects all the T^i messages from as many processes as possible (including itself) and records their arrival time (for those faulty processes whose message is not received, U^i is used instead). The selection of U^i ensures that all T^i messages from nonfaulty processes can be received. When the timer interrupts at U^i, p applies a fault tolerant majority function , similar to the one in Section 2.5.4, to the set of arriving time of those received T^i messages to calculate a corrective value. This corrective value is used to adjust the current clock C_p^i, resulting in a new clock C_p^{i+1}. Here, we assume

```
procedure ClockSyn;
begin
  T := some initial value;
  U = T + (1 + ρ)(δ + β + ε);
  for i := 0 to ∞ do
  begin
    if Interrupted by Clock = T then
    begin
      broadcast(i);
      set-timer(U);
      while Clock ∈ [T, U] do
        if receive message from process q then ARR_q := Clock;
/* Clock = U */
      AV := mid(reduce^f(ARR));
      ADJ := T + δ − AV;
      Clock := Clock + ADJ;
      T := T + P;
      U := T + (1 + ρ)(δ + β + ε);
      set-timer(T)
    end
  end
end;
```

Fig. 3.1 A clock synchronization algorithm.

that $\delta - \varepsilon \geq \beta$ to simplify the analysis. The analysis can be easily generalized to cases of $\delta - \varepsilon < \beta$, in which each process can start to collect T^i messages at time $T^i + (\delta - \varepsilon - \beta)$.

In the following analysis, we will explore the conditions under which the following three statements are valid for all $i \geq 0$:

S1 $U^i + Adj_p^i < T^i + P$, for all nonfaulty processes p.

S2 $t_p^i + \delta - \varepsilon > u_q^i$, for all nonfaulty processes p an q.

S3 $|t_p^i - t_q^i| \leq \beta$, for all nonfaulty processes p and q.

The first statement ensures that timers are set in the future. This prevents timer for T^{i+1} from being set to a time on a clock which has already passed. The second statement tries to guarantee that the T^{i+1} messages from other processes for the $(i+1)$th round can arrive only after the ith round for any process has finished. This makes it impossible for a process to receive more that one T^i message from any nonfaulty process during the period from T^i to U^i. The third requirement

states that the separation of any two nonfaulty processes' clock is bounded by β in real time. This is an important and useful property in distributed systems. If two processes initiate some events at the same time with respect to their own logical clock, these two event can be considered happening almost at the same time, differing at most by β in real time.

We use $ARR_p^i(q)$ to represent the time, measured on process p's logical clock, when process p receives process q's T^i message. By S3 and the ρ-bounded property, we have the first lemma about the arrival time of T^i messages.

Lemma 3.1. *For nonfaulty processes p and q.*

(1) $ARR_p^i(q) \leq T^i + (1+\rho)(\delta + \varepsilon + \beta)$.

(2) $ARR_p^i(q) \geq T^i + (1-\rho)(\delta - \varepsilon - \beta)$.

The above property can be used to bound the range of the adjustment.

Lemma 3.2. *For any nonfaulty process p, $|ADJ_p^i| \leq (1+\rho)(\beta + \varepsilon) + \rho\delta$.*

PROOF. By the algorithm

$$ADJ_p^i = T^i + \delta - AV_p^i.$$

The fault tolerant averaging function ensures that for nonfaulty processes q and r

$$ARR_p^i(q) \leq AV_p^i \leq ARR_p^i(r)$$

So, we get

$$T^i + \delta - ARR_p^i(q) \leq ADJ_p^i \leq T^i + \delta - ARR_p^i(r)$$

for some nonfaulty processes q and r.

By Lemma 3.1, the above inequality implies

$$\begin{aligned} ADJ_p^i &\geq T^i + \delta - (T^i + (1+\rho)(\beta+\delta+\varepsilon)) \\ &= -(1+\rho)(\beta+\varepsilon) - \rho\delta \end{aligned}$$

and

$$\begin{aligned} ADJ_p^i &\leq T^i + \delta - (T^i - (1+\rho)(\beta-\delta+\varepsilon)) \\ &= (1-\rho)(\beta+\varepsilon) + \rho\delta. \end{aligned}$$

The lemma follows. ∎

Now, we can see under what condition S1 holds. The following lemma gives a lower bound on P, the period.

Lemma 3.3. *If*

$$P > 2(1+\rho)(\beta+\varepsilon) + (1+\rho)\delta + \rho\delta$$

then

$$U^i + ADJ_p^i < T^{i+1}.$$

for any nonfaulty process p.

PROOF. By Lemma 3.2, we have

$$\begin{aligned} U^i + ADJ_p^i &\leq U^i + (1+\rho)(\beta+\varepsilon) + \rho\delta \\ &< T^{i+1} \\ &= U^i + P - (1+\rho)(\beta+\delta+\varepsilon) \end{aligned}$$

So, we get the lower bound on P. ∎

This lower bound of P also ensures S2.

Lemma 3.4. *If*

$$P > 2(1+\rho)(\beta+\varepsilon) + (1+\rho)\delta + \rho\delta$$

then

$$t_q^{i+1} + \delta - \varepsilon > u_p^i.$$

PROOF. Since $t_q^{i+1} + \delta - \varepsilon > t_p^{i+1} - \beta + \delta - \varepsilon$, by S3, it suffices to show that

$$t_p^{i+1} - u_p^i > \beta - \delta + \varepsilon.$$

Since the clock is ρ-bounded, we have

$$t_p^{i+1} - u_p^i \geq (P - (1+\rho)(\beta + \delta + \varepsilon) - ADJ_p^i)/(1+\rho)$$

By Lemma 3.3 and $\delta - \varepsilon \geq \beta$

$$\begin{aligned} P &> 2(1+\rho)(\beta+\varepsilon) + (1+\rho)\delta + \rho\delta \\ &> 3(1+\rho)(\beta+\varepsilon) + \rho\delta. \end{aligned}$$

By Lemma 3.2

$$ADJ_p^i \leq (1+\rho)(\beta+\varepsilon) + \rho\delta.$$

So, we get

$$\begin{aligned} t_p^{i+1} - u_p^i &> (3(1+\rho)(\beta+\varepsilon) + \rho\delta \\ &\quad - (1+\rho)(\beta+\delta+\varepsilon) \\ &\quad - (1+\rho)(\beta+\varepsilon) - \rho\delta)/(1+\rho) \\ &= \beta - \delta + \varepsilon. \end{aligned}$$

∎

To prove the third statement and to determine the upper bound of the period P, we need several more lemmas. The following lemma bounds the error in a process's estimate of the difference in real time between its own clock and another nonfaulty process's clock reaching T^i.

Lemma 3.5. *For nonfaulty processes p and q,*

$$|(ARR_p^i(q) - (T^i + \delta)) - (t_q^i - t_p^i)| \leq \varepsilon + \rho(\beta + \delta + \varepsilon).$$

PROOF. Suppose $t_q^i - t_p^i = \Delta$, $|\Delta| \leq \beta$. If $\Delta \geq 0$, then

$$\begin{aligned} ARR_p^i(q) - (T^i + \delta) - \Delta &\leq (1+\rho)(\delta + \varepsilon + \Delta) + T^i - (T^i + \delta) - \Delta \\ &= \varepsilon + \rho(\Delta + \delta + \varepsilon) \\ &\leq \varepsilon + \rho(\beta + \delta + \varepsilon). \end{aligned}$$

If $\Delta < 0$, then

$$\begin{aligned} ARR_p^i(q) - (T^i + \delta) - \Delta &\geq (1-\rho)(\delta - \varepsilon + \Delta) + T^i - (T^i + \delta) - \Delta \\ &= \varepsilon - \rho(\Delta + \delta + \varepsilon) \\ &\geq -\varepsilon - \rho(\beta + \delta + \varepsilon). \end{aligned}$$

The lemma follows. ∎

The essence of the clock synchronization algorithm presented here is to try to reduce the distance between the clocks from β to $\beta/2$. Central to this reduction is the halving property of the fault tolerant averaging function used in the algorithm. Consider three multisets, U, V, and W, with $|U| = |V| = n$ and $|W| = n - f$. If there are injections c and c', $c : W \to V$, $c' : W \to V$, such that under these mappings, there is no $w \in W$ with $|w - c(w)| > x$ or $|w - c'(w)| > x$, for some constant x, then $|mid(reduce^f(U)) - mid(reduce^f(V))| \leq diam(W) + 2x$, where $mid(U)$ is the midpoint of the set U, $reduce^f(U)$ returns the set U with the f highest and f lowest elements removed, and $diam(W)$ is defined as $max(W) - min(W)$. Using this halving property, we can prove the following lemmas, which indicate how adjustments are related to the clocks.

Lemma 3.6. *For nonfaulty processes p and q,*

$$|(t_p^i - t_q^i) - (ADJ_p^i - ADJ_q^i)| \leq \beta/2 + 2\varepsilon + 2\rho(\delta + \beta + \varepsilon).$$

PROOF. We define multisets

$$U = \{t_p^i - (T^i + \delta) + ARR_p^i(s)\}.$$

$$V = \{t_q^i - (T^i + \delta) + ARR_q^i(s)\}.$$

$$W = \{t_r^i | r \text{ is nonfaulty}\}.$$

Here, $|U| = |V| = n$ and $|W| = n - f$. By S3, $diam(W) = \beta$.

Let $x = \varepsilon + \rho(\delta + \beta + \varepsilon)$. Define injection c from W to V as $c(t_r^i) = t_p^i - (T^i + \delta) + ARR_p^i(r)$ and injection c' from W to V as $c'(t_r^i) = t_q^i - (T^i + \delta) + ARR_q^i(r)$. By Lemma 3.5, we can not find any nonfaulty r such that

$$|(ARR_p^i(r) - (T^i + \delta) - (t_r^i - t_p^i)| > x$$

or

$$|(ARR_q^i(r) - (T^i + \delta) - (t_r^i - t_q^i)| > x.$$

Thus we can use the halving property discussed above, resulting in

$$|mid(reduce^f(U)) - mid(reduce^f(V))| \leq \beta/2 + 2\varepsilon + (\delta + \beta + \varepsilon).$$

Since

$$\begin{aligned} mid(reduce^f(U)) &= mid(reduce^f(t_p^i - (T^i + \delta) + ARR_p^i)) \\ &= t_p^i - ADJ_p^i. \end{aligned}$$

and

$$\begin{aligned} mid(reduce^f(V)) &= mid(reduce^f(t_q^i - (T^i + \delta) + ARR_q^i)) \\ &= t_q^i - ADJ_q^i. \end{aligned}$$

The lemma follows. ∎

Lemma 3.7. *For nonfaulty processes p and q,*

$$|(u_p^i - u_q^i) - (ADJ_p^i - ADJ_q^i)| \leq \beta/2 + 2\varepsilon + 2\rho(2 + \rho)(\delta + \beta + \varepsilon).$$

PROOF. Relating u^i to t^i, we can get

$$|(u_p^i - u_q^i) - (ADJ_p^i - ADJ_q^i)| \leq |(t_p^i - t_q^i) - (ADJ_p^i - ADJ_q^i)| + |(u_p^i - u_q^i) - (t_p^i - t_q^i|.$$

By Lemma 3.6, the first item is $\leq \beta + 2\varepsilon + 2\rho(\delta + \beta + \varepsilon)$. The ρ-bounded property implies that

$$(1-\rho)(1+\rho)(\delta+\beta+\varepsilon) \leq u_r^i - t_r^i \leq (1+\rho)(1+\rho)(\delta+\beta+\varepsilon).$$

So, the second item is $\leq 2\rho(1+\rho)(\delta+\beta+\varepsilon)$. Thus, we have

$$\begin{aligned}|(u_p^i - u_q^i) - (ADJ_p^i - ADJ_q^i)| &\leq \beta/2 + 2\varepsilon + 2\rho(\delta+\beta+\varepsilon) \\ &\quad + 2\rho(1+\rho)(\delta+\beta+\varepsilon) \\ &= \beta + 2\varepsilon + 2\rho(2+\rho)(\delta+\beta+\varepsilon).\end{aligned}$$

∎

The next lemma bounds the distance in real time between the new clock at U^i. This distance must be appreciably less than β in order to accommodate their relative drift during the interval between U^i and T^{i+1}. This lemma also implies that β should be bounded from below by the inequality

$$\beta \geq 4\varepsilon + 4\rho(3\beta + \delta + 3\varepsilon) + 8\rho^2(\delta + \beta + \varepsilon).$$

Lemma 3.8. *For nonfaulty processes p and q,*

$$\begin{aligned}&|c_p^{i+1}(U^i) - c_q^{i+1}(U^i)| \\ &\quad \leq \beta/2 + 2\varepsilon + 2\rho(2\beta + 2\delta + 3\varepsilon) + 4\rho^2(\delta + \beta + \varepsilon).\end{aligned}$$

PROOF. Since $C_p^{i+1}(u_p^i) = U^i + ADJ_p^i$, we have

$$\begin{aligned}u_p^i &= c_p^{i+1}(U^i + ADJ_p^i) \\ &= c_p^{i+1}(U^i) + (1+\rho')ADJ_p^i, \qquad |\rho'| \leq \rho.\end{aligned}$$

Similarly,

$$u_q^i = c_q^{i+1}(U^i) + (1+\rho'')ADJ_q^i, \qquad |\rho''| \le \rho.$$

So, we have

$$\begin{aligned}
&|c_p^{i+1}(U^i) - c_q^{i+1}(U^i)| \\
&\quad = |(u_p^i - u_q^i) - (ADJ_p^i - ADJ_q^i) - (\rho' ADJ_p^i - \rho'' ADJ_q^i)| \\
&\quad \le |(u_p^i - u_q^i) - (ADJ_p^i - ADJ_q^i)| + |\rho' ADJ_p^i| + |\rho'' ADJ_q^i|
\end{aligned}$$

By Lemma 3.7, the first item is $\le \beta/2 + 2\varepsilon + 2\rho(2+\rho)(\delta+\beta+\varepsilon)$, while the sum of the second and third items

$$\begin{aligned}
&|\rho' ADJ_p^i| + |\rho'' ADJ_q^i| \\
&\quad \le 2\rho((1+\rho)(\beta+\varepsilon) + \rho\delta)
\end{aligned}$$

by Lemma 3.2.

The result follows by combining these bounds. ∎

Now, we can determine the upper bound of period, P.

Lemma 3.9. *If*

$$P \le \beta/3\rho - \varepsilon/\rho - \rho(\delta+\beta+\varepsilon) - 2\beta - \delta - 2\varepsilon$$

then

$$|t_p^{i+1} - t_q^{i+1}| \le \beta$$

for nonfaulty processes p and q.

PROOF. By definition

$$\begin{aligned}
&|t_p^{i+1} - t_q^{i+1}| = |c_p^{i+1}(T^{i+1}) - c_q^{i+1}(T^{i+1})| \\
&\le |(c_p^{i+1}(T^{i+1}) - c_q^{i+1}(T^{i+1})) - (c_p^{i+1}(U^{i+1}) - c_q^{i+1}(U^{i+1})|. \\
&\quad + |c_p^{i+1}(U^{i+1}) - c_q^{i+1}(U^{i+1}|
\end{aligned}$$

Using the same reasoning as in the proof of Lemma 3.7, the first item is less than $2\rho(P - (1+\rho)(\delta+\beta+\varepsilon))$. The bound on the second item is given by Lemma 3.8. So, we have

$$\begin{aligned}\beta \geq \quad & 2\rho(P - (1+\rho)(\delta+\beta+\varepsilon)) + \beta/2 + 2\varepsilon \\ & + 2\rho(3\beta + 2\delta + 3\varepsilon) + 4\rho^2(\delta+\beta+\varepsilon).\end{aligned}$$

The lemma follows. ∎

Now, we have obtained the most important results concerning the correctness of the clock synchronization algorithm. Based on these results, the γ-agreement is satisfied with

$$\gamma = \beta + \varepsilon + \rho(7\beta + 3\delta + 7\varepsilon) + 8\rho^2(\delta+\beta+\varepsilon) + 4\rho^3(\delta+\beta+\varepsilon)$$

and the $(\alpha_1, \alpha_2, \alpha_3)$-validity is satisfied with

$$\alpha_1 = 1 - \rho - \varepsilon/\varphi$$

$$\alpha_2 = 1 + \rho + \varepsilon/\varphi$$

$$\alpha_3 = \varepsilon$$

where

$$\varphi = (P - (1+\rho)(\beta+\varepsilon) - \rho\delta)/(1+\rho)$$

A detailed proof can be found in [LUN 84].

3.2 Fail-Stop Processor

As mentioned in the previous chapter, the Byzantine generals problem was first proposed to cope with arbitrary malicious faults in a distributed system. Interestingly enough, the solution to the Byzantine generals problem can be used to implement a special processor that has an extremely simple failure behavior, namely

fail-stop. Such a processor stops executing when a failure is encountered, instead of producing an erroneous output. This is a very attractive failure model since arbitrary failures, such as generation of erroneous outputs or loss of state information, often makes the design and programming of fault tolerant systems extremely difficult. Besides, many approaches to implementing fault tolerant systems assume that processors in the system are either fail-stop or something equivalent.

Ideally, a fail-stop processor (fsp) should have the following properties [SCH 84]:

(1) *Halt on Failure Property*: A processor will halt instead of performing an erroneous state transformation visible to other processors.

(2) *Failure Status Property*: Any processor can detect whether any other processor has failed, and thus halted.

(3) *Stable Storage Property*: The storage of a processor is partitioned into stable storage and volatile storage. The contents of stable storage are not affected by any failure and can always be read by any processor. The contents of volatile storage are not accessible to other processors and are lost as a result of a failure.

Of course, the problem of designing fault tolerant systems can not be completely solved by fail-stop processors; however it is greatly simplified since the designer is free to consider arbitrary failure behavior. For example, to construct a fault tolerant system for an application requiring N processors when no failure exists, $N + f$ fail-stop processors are able to tolerate up to f failures. Whenever a fail-stop processor halts, its failure will be detected and the job will be taken over by other working fsp's using the information in stable storage. In [SCH 83], Schlichting and Schneider proposed a methodology to program an fsp. In their approach, a recovery protocol R is associated with an action statement A, a sequence of statements, to form a fault-tolerant action FTA as follows

FTA: action

A

recovery

R

end

Normally, the execution of an FTA starts by running action A. Recovery protocol R is in effect only if the execution of A is interrupted by a failure. R also serves as its own recovery protocol for subsequent failures occurring during the execution of R. The execution of an FTA terminates if either A or R is executed entirely without failure. A special case of an FTA is the restartable action in which an action statement is also the recovery protocol.

Using Hoare-like logic [HOA 69], the correctness of an FTA can be written as

$$\{P\}FTA\{Q\}$$

where P and Q are the assumed precondition and postcondition respectively. Suppose that $\{P'\}A\{Q'\}$ and $\{P''\}R\{Q''\}$ are established. Then $\{P\}FTA\{Q\}$ can be established, provided

(1) $P \Rightarrow P'$ and $Q' \Rightarrow Q$.

(2) $Q'' \Rightarrow Q$.

(3) All program variables named in P'' must be in stable storage.

(3) P'' is satisfied whenever the action is restarted. This requires that P'' be derivable from any internal assertion appearing in the proof outline of $\{P'\}A\{Q'\}$ and $\{P''\}R\{Q''\}$ and these assertions do not use variables in volatile storage on other processors.

One interesting thing is that while this Hoare-like logic can be used to validate an FTA action, a fault tolerant program can be developed by combining a sequence of FTA's with proofs leading the way.

Just as a completely fault tolerant computing system can not be built with a finite amount of hardware, it is also impossible to implement a truly fail-stop pro-

cessor using a finite amount of hardware, since error detection capabilities provided by a finite hardware would be easily disabled by a sufficient number of failures. However, a fail-stop processor can be approximated in the sense that a processor behaves like a fail-stop processor as long as there are no more than k failures within its components. Such a k-fail-stop processor can be implemented by $3k+2$ processors, each with its own storage, interconnected by a communication network. The implementation consists of:

(1) $k+1$ p-processes (p for program), $\{p_1, \ldots, p_{k+1}\}$, each running on its own processor.

(2) $2k+1$ s-processes (s for storage), $\{s_1, \ldots, s_{2k+1}\}$. It is required that the s-processes of one fsp run on different processors. However, a simple processor can run several s-processes for different fsp's.

All of the $k+1$ p-processes in one fsp run the same program. The contents of the stable storage of a fsp are duplicated at each site of the $2k+1$ s-processes. Each p-process interacts with $2k+1$ s-processes by issuing either a read or a write request for stable storage. Each request r contains such information as the issuing time at p-process's local clock, *r.time*; the type of request (read or write), *r.type*; variable in the stable storage to be read, or written, *r.var*; the value of the variable in case of a write request, *r.val*. Since all the p-processes run the same program, they should issue the same request at the same logical time. When an s-process receives a request, it will record the receiving time on its own local clock, *r.rectime*. A solution to Byzantine generals problem using authenticated messages is used to implement a reliable broadcasting of requests such that

IC1: If p_i is nonfaulty, then every nonfaulty s-process s_u in the same fsp receives the request within δ (as measured on s_u's clock).

IC2: If s-process s_u and s_v in the same fsp are nonfaulty, then both of them agree on every request from p_i.

Also, a clock synchronization method is used to ensure

IC3: For all the p-processes in any fsp, their clocks are synchronized.

Fig. 3.2 is a program run by each s-process in a fsp. Each s-process maintains a variable *Failed* in stable storage to indicate whether the fsp has failed or not. *Stable*[*Var*] denotes the variable *Var* in that s-process's copy of the stable storage. Let $p(FSP_i)$ denote all the p-processes belonging to fail-stop processor FSP_i and $orig(r)$ denote the p-process issuing the request r. Each s-process assumes that requests with the same issuing time (according to each p-process's local clock) should be the same request and they arrive within Δ. Failure is detected when a write request is not from exactly $k+1$ distinct processes or the contents of those $k+1$ requests are not the same.

A p-process interacts with s-processes in the following way:

(1) A p-process can only write to the stable storage in its own fsp by initiating a Byzantine protocol for a write request with all the s-processes in that fsp.

(2) A p-process can read from stable storage in any fsp by broadcasting the read request to all the s-processes in that fsp and take majority of the values received.

(3) A p-process determines whether any fsp has failed by reading the variable *Failed* from the stable storage in that fsp.

Since there are $2k+1$ s-processes in each fsp, the majority of those copies of the stable storage are correct as long as the failures of s-processes are no more than k. Also, the implementation of s-process enforces that only nonfaulty fsp with all its p-processes issue the same write request can write to its stable storage, and any fsp can read from stable storage in any other fsp, even from a halted fsp. This implies that the above implementation satisfies the stable storage property. Failure status property is implemented by letting each s-process maintain in stable storage a *Failed* variable which can be read by any fsp. Halt on failure is obtained as long as in any fsp, the number of faulty p-processes is at most k. Failure is detected if

```
procedure Storage_Maneger;
begin
  owner := i;
  failed :=false;
  repeat
    for s := 1 to N do
    begin
      T := CLOCK;
      D := {r | orig(r) ∈ p(FSP_s)};
      while D ≠ φ do
        begin
          minT := min{r.time | r ∈ D};
          minRecT := min{r.rectime | r ∈ D&r.time = minT};
          if minRecT < T − Δ then
          begin
            R := {r | r ∈ D&r.time = minT};
            D := D − R;
            if ∀r : r ∈ R : r.type = read then
              for for r ∈ R do
                send Stable[r.val] to orig(r)
            else if (∀r, r' : distinct r, r' ∈ R :
                m = m'&m.type = write&orig(m) ≠ orig(m'))&
                |M| = k + 1&s = owner&¬failed) then
              begin
                choose an r ∈ R;
                Stable[r.val] := r.val;
              end
            else if s := owner& ≠ failed then
              begin
                failed := true;
                for all p ∈ p(FSP_i) do send "halt" to p
              end
          end
        end
    end
  forever
end;
```

Fig. 3.2 Program for s-process in a k-fail-stop processor FSP_i.

the write requests issued by the k p-processes in a fsp are not the same or some p-processes fail to issue the write requests. A more restricted measure may require that all the $k + 1$ requests be the same, irrespective of whether they are read or write requests. The one adopted here is under the belief that a read failure would not corrupt the stable storage but eventually would cause a write failure.

3.3 State Machine Method

Another general methodology to design fault tolerant distributed systems is the state machine approach proposed by Lamport [LAM 84]. In this method, a program is viewed as a state machine driven by events (input), such as the arrival of a message or the reaching of a certain time, to generate actions (output). A reliable system is constructed by replicating these state machines and running them in parallel. The innovative distinction about this approach is that absolute time, instead of timeout, is used to achieve fault tolerant synchronization in distributed system.

For the state machine to be effective, the underlying distributed system should satisfy the following assumption:

A1: For any event e that causes process i to send a message to process j, the delay of such a message is at most δ, a constant. This time is either measured according to process i's clock or measured according to process j's clock.

A2: The clocks of any two nonfaulty processes are synchronized within ε.

A3: Any process can identify the source of any message it receives.

A1 and A2 imply that if process i sends a message at time T on i' clock, process j would receives that message by $T + \delta + \varepsilon$ on its own clock, provided that the communication link between them is nonfaulty. Combined with A3, this property further implies that if i sends j a message timestamped T on i's clock and does not send another one until $T + \delta + \varepsilon$, j would receive it by time $T + \delta + \varepsilon$ on its own clock and knows that it is the one sent by i at time T if the communication link between them is nonfaulty. One of the essential idea in the state machine method is to let each process broadcast a message at every time unit with the unit long enough for some processing to be done. Thus a process i can send a null message to process j at time T by sending nothing so that process j receives nothing by time $T + \delta + \varepsilon$ on j's clock. Also, if more than one message is received from process i in

the interval $[T, T+\delta+\varepsilon]$ (suppose that $\delta > \varepsilon$), process i is considered to be faulty and the null message is taken as a default. This property can be easily generalized to cases where no direct links exist between i and j, instead the message has to be sent through the path $i = k_0, k_1, \ldots, k_n = j$. A message timestamped by T is sent to k_0 by i at time T on i's clock, relayed by every intermediate processes, and received by j by the time $T + n\delta + \varepsilon$ on j's clock, if all the intermediate nodes and links are nonfaulty. A Byzantine protocol is used to implement a reliable broadcasting satisfying the following requirement:

IC: If process i initiates the broadcast of a message at time T on its clock, then

1) if i is nonfaulty, then every nonfaulty process j receives that message by $T + \Delta$ on j's clock.

2) if both j and j' are nonfaulty, then each of them receives the same message (may be null) by $T + \Delta$ on its own clock. where Δ is some unspecified constant depending on the communication network and the Byzantine protocol used.

Now, we can describe the general idea of the state machine algorithm (Fig. 3.3). At every instant of time T, each process i issues a command $C_{i,T}$ by broadcasting it to all the processes in the system, using a protocol satisfying IC. At time $T+\Delta$ (the time *Clock* in Fig. 3.3), the command $C_{1,T}, \ldots, C_{N,T}$, some of which may be null, are executed in order. Any "timeout" action that is supposed to occur is performed by executing a special *timeaction* command. Here, a command is a subroutine with arguments of a state, a time, and a process number in case of a $C_{i,T}$ command. The execution of these commands forms the *time T execution step.* As a result of the *time T execution step*, the state is changed from $state_{T-1}$ to $state_T$ and some outputs are generated. After it finishes, the state machine goes to the *time* $T+1$ *execution step*, and so on.

If we regard the output as a message from a process to itself, then the time need

```
procedure statemachine;
begin
  repeat
    for i := 1 to N do
    execute C_{i,clock-Δ}(state, clock − Δ, i);
    execute timeaction(state, clock − Δ);
    generate command C_{i,clock} and initiate its broadcast
      to all processes;
      clock := clock + 1
  until false
end;
```

Fig. 3.3 State machine algorithm.

to executed the *time T execution step* is at most δ. Thus every process can finish the *time T execution step* by time $T+\Delta+\delta$. This implies that the time unit should be appreciably larger that $\Delta + \delta$ in order to do some meaningful processing. The fault tolerant synchronization property is established by the following assertions, which are guaranteed by condition IC.

(1) If process i has not failed by time $T + \Delta + \delta$ on its own clock, it would by that time have executed the *time T execution step*, using the command it generated at time T as the command $C_{i,T}$.

(2) Any two processes that have not failed by time $T + \Delta + \delta$ would have executed the identical sequences of state machine execution steps though time T.

Now, consider the implementation of a distributed semaphore as an example of the state machine method. In the implementation, the state has four components—the value of the semaphore, a queue of waiting processes, a queue of granted processes with their granting time. A P command issued by process i adds i to the end of the waiting queue, and a V command issued by i deletes i from the granted queue (in any position). The *timeaction* command effectively executes a V command when a process has been in the granted queue for too long. (A process is assumed to stay in the granted queue for at most ω, and take an extra Δ' for itself to be notified that it had been granted.) A process at the head of the waiting queue

can enter the critical section if the value of the semaphore is greater than zero and is notified by the appropriate output.

```
procedure P(state, T, i);
begin
  state.value := state.value − 1;
  if state.value ≥ 0 then
  begin
    state.gqueue := insert((i, T), state.gqueue);
    output "i granted"
  end else state.wqueue := insert(i, state.wqueue)
end;
procedure V(state, T, i);
begin
  if i is in state.gqueue then
  begin
    state.gqueue := delete(i, state.gqueue);
    if state.wqueue is empty then state.value := state.value + 1;
    else begin
      output "head(state.wqueue) granted";
      state.wqueue := tail(state.queue);
      state.gqueue := insert((i, T), state.gqueue)
    end
  end;
end;
procedure timeaction(state, T);
begin
  while T ≥ head(state.gqueue).time + ω + Δ'
  begin
    state.gqueue := tail(state.gqueue);
    if state.wqueue is empty then state.value := state.value + 1;
    else begin
      output "head(state.wqueue) granted";
      state.gqueue := insert((head(state.wqueue), T), state.gqueue);
      state.wqueue := tail(state.queue)
    end
  end;
end;
```

Fig. 3.4 Command definitions for the distributed semaphore problem.

The commands (procedures) are defined in Fig. 3.4, where $insert(item, queue)$ returns the new queue by inserting *item* at the end of the queue, $delete(i, gqueue)$ returns the new queue by deleting the $(i, time)$ pair from the *gqueue*, $tail(queue)$ returns the new queue obtained by deleting the head of *queue*, and $head(queue)$

returns the head of the *queue*. Each process executes the state machine algorithm in Fig. 3.3. Process i enter the critical section from the time it generates the "i granted" output until it either executes an V command or fails. Note the output of the state machine is only used to notify the process of the result of executing the state machine. It is not sent from one processes to another. Fig. 3.5 describes how the algorithm can be implemented by a more practical "interrupt-driven" program. In such a scheme, each process performs an action only in response to events of receipt of a message, a clock interrupt, a process's desire to issue a P or V command.

Condition	Actions

1. ($clock = T + \Delta$) and (there is a message in the buffer) timestamped T)	for $j := 1$ to N do
	if "$T : j : *$" message is in buffer then
	begin
	remove the message from the buffer;
	if the message is "$T : j : P$" then execute $P(state, T, j)$;
	if the message is "$T : j : V$" then execute $V(state, T, j)$
	end;

2. ($clock = j.time + \omega + \Delta'$) for some process j in *state.gqueue*	delete the $(j, time)$ pair form *state.gqueue*;
	if *state.wqueue* is empty
	then *state.value* := *state.value* + 1;
	else begin
	output "*head*(*state.wqueue*) granted";
	state.gqueue := *insert*((*head*(*state.wqueue*), T), *state.gqueue*);
	state.wqueue := *tail*(*state.queue*);
	output "*head*(*state.wqueue*) granted"
	end;

3. (wants to enter the critical section) and ($clock = T$)	initiate broadcast of "$T : i : P$" message to all processes;
4. (wants to exit from the critical section) and ($clock = T$)	initiate broadcast of "$T : i : V$" message to all processes;

Fig. 3.5 Process i's interrupt-driven program for distributed semaphore.

It is easy to verify that the above implementation satisfies the mutual exclusion,

progress and bounded waiting time requirements of critical regions. Compared to the implementations in [THE 83], the state machine method does not need such special hardware operations as decrement (increment) and test, or read and clear. This implementation is somewhat similar to solutions using time stamp and event-ordering [LAM 78] [RIC 81]. Since in the state machine, absolute time is used and the absence of message is counted as a null message, the number of message required per critical section entry is only half of the minimum number required in [RIC 81].

3.4 Distributed Database

Several papers have addressed the applications of Byzantine agreement to distributed database system [LAM 82] [LAM 84] [GAR 84]. Generally speaking, the scope of applying Byzantine agreement techniques here is relatively small. Almost all the applications try to use reliable broadcasting, implemented by the Byzantine agreement, to assure that transactions are properly executed by reliable sites in the system.

In the following discussion, we usually assume that each transaction originates from some single user. In case the requests for a transaction are from several users, the reliable broadcasting can be executed in parallel so that whenever a process receives the request from the required number of users, the target request can be executed. Another assumption is that the processing nodes are, generally, not the input/output nodes. Although this is usually true, especially when the database is fully replicated, we want to emphasize here that it is the main reason for using reliable broadcasting. Here, we assume that each node can play the dual roles as an input node and a processing node.

The correctness of such an ideal fault tolerant distributed database system can be defined as follows:

(1) Users should get the same results from the system that they would obtain from an ideal system where no failures occur.

(2) If a transaction is submitted at a reliable input node, that transaction will be in the resulting schedule.

(3) The time to commit a transaction is bounded.

The first requirement is natural since it just enforces that a fault tolerant system should fulfill the same task as the ideal system. The second requirement may not be preferable from the point of view of the users, who would like the system to guarantee the correct execution of all the transactions. We choose that requirement since we feel it is unreasonable to assume that the input nodes are faulty free. The third requirement is essential, too, since it would be absurd to let the user wait, say, one year, to get a transaction finished.

Given these assumptions and requirements, we can discuss the construction of such a system. It is obvious that if each piece of data has only one copy, the database can be easily destroyed in case the single copy of some data item happens to be stored on a faulty node. So, it is necessary to replicate the database here. If there are t faulty processes in the system, at least $t + 1$ copies are needed to ensure that at least one reliable node handles the data correctly. Unfortunately, there is no general method to tell whether some node is reliable or not. So, we actually need to store at least $2t + 1$ copies of the database so that the correct database can be drawn from the majority of the copies. In case the database is not fully replicated, we just let every node participate in the reliable broadcasting process, when a transaction is issued, and only let those nodes where the required data reside response. In the following discussion, we identify two major applications of Byzantine agreement to distributed data processing: the distribution of input request and transaction commitment.

It is obvious that the input node is critical. It would be unreasonable to move

the distribution function to the user by asking the user to submit the request at different locations. So, we would like the input node to take the request for a transaction and distribute it to all the processing nodes. In case the input nodes are perfect, all the processing nodes can agree on the same request without using Byzantine agreement. In order to guarantee that all the processing nodes have the same "final" state, each processing node should have its execution schedule equivalent to a serial schedule. This can be done by letting each input i node number the transactions it generates as $T_{i,1}, T_{i,2}, \ldots$ The processing nodes will then process the transaction in the order of $T_{1,1}, T_{2,1}, \ldots, T_{n,1}, T_{1,2}, T_{2,2}, \ldots, T_{n,2}, \ldots$ In this way, all the processing nodes will receive the same sequence of transactions. This method satisfies the three correctness requirements. But one question with this scheme is that transaction processing will proceed at the rate of the slowest input node. The state machine method can be used here to let all the requests be issued periodically. If a processing node does not receive a request from some input node i at some period, then i is considered to have issued a null request. As pointed out in Section 3.3, this solution requires all the clocks to be synchronized.

Next, we relax the assumption for the input node. We assume that the input node may fail to send the transaction to some or all processing nodes, may wait an arbitrary amount of time between transmissions, and may send transactions in any order. However, if a processing node receives a transaction from some input node, this transaction is the one actually issued by the user. In this model, an input node can hold a transaction arbitrarily long before it broadcasts it, violating the bounded time requirement. To avoid this, we can attach to each transaction a timestamp, storing its arrival time, which the input node can not modify by our assumption. This timestamp can be used by processing nodes to discard transactions which are delayed for too long. However, in this model, there is no way to guarantee the first correctness requirement since an input node may fail to broadcasting it, or

hold it for too long so that it will be discarded by the processing nodes. Since an input node may send a transaction to only some of the processing nodes, Byzantine agreement is required to ensure that all the processing nodes receive the same sequence of transactions. In the following scenario, processing nodes agree to perform a Byzantine protocol every T time units, t_d is the guaranteed network delivery time, t_r is the time taken by an input node to process the incoming transaction, t_B is the time taken to reach Byzantine agreement and ε is the maximum clock drift.

(1) Each processing node collects transaction from input nodes.

(2) At time iT, each node selects transactions with timestamp $ts \in [(i-1)T-(\varepsilon+t_d+t_r), iT-(\varepsilon+t_d+t_r)]$. Transactions with timestamp $ts < (i-1)T-(\varepsilon+t_d+t_r)$ are discarded.

(3) At time iT, each processing node initiates a Byzantine Agreement with transactions selected in step (2).

(4) At time $iT + \varepsilon + t_B$, the Byzantine agreement is complete. Each reliable processing node has the same set of transactions. These transactions are then executed in the same order by the reliable processing nodes.

In this scenario, similar to the state machine approach, each transaction waits for at most $2\varepsilon + t_d + t_r + t_B$ time units before it is being processed.

Now, we relax the assumption further such that the input nodes can be arbitrarily faulty, even maliciously. Unfortunately, we can not come up with a more innovative way to apply Byzantine agreement. What we have is still the strategy above. While we can still guarantee that all processing nodes execute the same sequence of transactions, the bounded time requirement can no longer be satisfied, since a faulty input node can even change the timestamp of a transaction to make an old transaction look new.

Another application of Byzantine agreement to distributed database systems is the transaction commit problem. We have two cases here. In the first case, there

is a transaction coordinator which decides whether to commit or abort a transaction and initiates a Byzantine protocol to broadcast its decision to its subordinates. In another case, one process issues a *transaction complete* request, whereupon every other process issues either a *commit* or *abort* request. If all processes issue a *commit* request, then the transaction must be committed, otherwise it is aborted. Again, in this scheme, processing has to proceed at the rate of the slowest process, even worse, a faulty process could never response. One solution presented in [LAM 84] is to use the state machine method. In the state machine solution, each state consists of a mapping from transaction identifier to time, status pair. For a transaction I, $state(I).time$ is the time at which a *transaction complete* request is issued, $state(I).status$ is an n-tuple whose ith component is equal to *commit* or *undecided*–depending on whether process i has issued a *commit* request. The definitions of these commands are given in Fig. 3.6. Initially, $state(I).time$ equals to ∞, for any transaction I. A *transactioncomplete* request issued at time T must be responded to before $T + \Omega$, otherwise the *timeaction* is taken to abort that transaction.

With this algorithm, any single failure will result in the abortion of all transactions. The solution can be modified to commit a transaction despite some failures. This is an elegant solution. There is no "window of vulnerability" and processes and communication lines may fail at any time without delaying the system. Again, this approach can only satisfy the second correctness requirement.

Although not applications of Byzantine agreement, there are two other problems concerning arbitrary failures of faulty nodes in the system. One problem is with the output nodes. The output nodes are crucial here since users have to get the results from the output nodes. If an output node happens to be a faulty node, then the user can not be assured to get the correct result. Since there is no way to know which nodes are faulty in advance, at least $2t + 1$ copies of data should be

```
procedure transaction_complete(I, state, T, i);
begin
  state(I).time := T;
  state(I).status := ("undecided", ..., "undecided");
  state(I).status.i := "commit"
end;
procedure commit(I, state, T, i);
begin
  if state(I).time ≠ ∞
        then state(I).status.i := "commit";
  if state(I).status = ("commit", ..., "commit") then
  begin
    output "transaction I committed";
    state(I).time := ∞
  end
end;
procedure abort(I, state, T, i);
begin
  if state(I).time ≠ ∞ then
  begin
    output "transaction I aborted";
    state(I).time := ∞
  end
end;
procedure timeaction(state, I)
begin
  for all transaction I do
    if state(I).time ≤ T − Ω then
    begin
      output "transaction I aborted";
      state(I).time := ∞
    end
end;
```

Fig. 3.6 Command definitions for transaction commit problem.

collected to reveal the correct result when there are t faulty nodes. It is obvious that no solution may exist if the output node is also a processing node. One solution is to move the duty of output node to users such that users directly check the $2t + 1$ copies and choose the majority. This approach is not satisfactory also, because this merely moves the burden from the system to the users.

Another problem is error recovery and reintegration of faulty nodes. In case of arbitrary failure, this problem is extremely difficult if not totally insolvable. One reason is that in a long term system, we can can not assume there will be at most

t faulty nodes forever. This condition can be relaxed by assuming that at any instant, the number of faulty nodes is at most t and reliable nodes are always in a majority. Another reason is that the local stable storage may be fully corrupted after a maliciously failure so the whole database should be recovered. Suppose at some instant T, there are p reliable nodes, t faulty nodes, and r recovered nodes being reintegrated into the system. To cover that state information, those recovering nodes should obtain data from nodes which ware alive before time T. If we assume that all the recovering nodes are reliable, then the recovering nodes can recover data from the majority of those collected if $p > t$. In case r' of the recovering nodes are faulty, p should be greater than $t + r'$. This scheme works only if the whole database is recovered in that recovering procedure and no reliable process fails in between. This is not practical because it imposes a large overhead on the system. Also, not all the data items may be required in the future. A better method is to try to recovery a data item when it is required. In this scheme, a recovering node collects such a data item from nodes with a longer life time where that data item resides. This requires that a majority of those originally reliable nodes be kept alive before the whole database is recovered.

CHAPTER 4

Self-stabilization and Byzantine Agreement

4.1 Self-stabilization

In his classical paper on self-stabilization [DIJ 74], Dijkstra defined a system to be self-stabilizing if "it is guaranteed to arrive at some legitimate state in a finite number of steps", regardless of its initial state and if it remains so forever. Here, the property that "the system is in a legitimate state" should be kept invariant as a consequence of the synchronization between processes. This problem can be solved simply and systematically if there is a common store where the "the current system state" is recorded and can be accessed by different processes in a mutually exclusive way.

However, the problem gets harder when there is no such common store and the state information is stored in variables distributed over the various processes. If the state information in each process can fully reveal the state of the whole system or each process can get information from every other process in the system, the situation is similar to having a common storage. Further, suppose that each process's knowledge about the system is not complete and the communication facilities are limited such that a process can exchange information only with its "neighbors" which constitute only a small subset of all the processes in the system. In this case, the solution is no longer trivial.

In his paper, Dijkstra gives a small yet elegant and convincing example to illustrate the self-stabilizing technique. Consider a system built from $N+1$ finite state machines numbered 0 through N. The machines are arranged in a ring. Machine i, $0 \leq i \leq N$, has machine $(i-1) \bmod (N+1)$ as its left-hand neighbor and

machine $(i + 1) \bmod (N + 1)$ as its right-hand neighbor. In the center of the ring stands a demon that gives the command "adjust yourself" to one of the machines in "fairly random order". Here, the "fairly random order" means that in each infinite sequences of successive commands issued by the demon, each machine receives the commands infinitely often. Upon "adjustment", a machine goes into a new state, which must be a function of its own state and the states of its two neighbours.

For each machine, a "privilege" is defined as a function of its own state and the state of its neighbours. The legitimate states are defined as those states such that: (1) exactly one privilege will be present; (2) each possible move from a legitimate state will bring the system again into a legitimate state; and (3) for any pair of legitimate states there exists a sequence of moves transforming the state of the system from one into another.

In Dijkstra's solution, each machine has K states so each state can be represented as a number between 0 and $K-1$. A machine i, $1 \leq i \leq n$ is privileged when its state differs from that of $i-1$, i.e., $x[i] \neq x[i-1]$. There is an exception for machine 0. (The problem is generally not solvable if all the machines are identical.) Machine 0 is privileged if its state is the same as machine N, i.e., $x[0] = x[N]$. An adjustment on a privileged machine would cause that machine to lose its privilege and pass the privilege to its right neighbour. A non-privileged machine is said to "fire" if it becomes privileged. No effect would result from an adjustment on a non-privileged machine. So, the adjustment of machine i other than 0 means

if $x[i] \neq x[i-1]$ **then** $x[i] := x[i-1]$ **fi**.

For the exceptional machine 0, this would be

if $x[0] = x[N]$ **then** $x[0] := (x[0] + 1) \bmod K$ **fi**.

It is easy to see that if system is already in a legitimate state any move will transform the system into another legitimate state since in this case any adjustment either has an effect to let the privileged machine pass its privilege to its right-hand

neighbor or results in no changes at all. To see that any legitimate state can be reached from one legitimate state, we just need to show that the exceptional machine will continue forever to get the chance to be privileged. In this way, the privilege is guaranteed to rotate around the ring and the loss of privilege of process 0 would let all processes enter another state. Suppose the exceptional machine is not privileged, i.e., $x[0] \neq x[N]$, then in a finite number of adjustment it will become so. For let j be the smallest number such that $x[j] \neq x[0]$. Since j is the smallest such number, then $x[j-1] = x[0]$, i.e., $x[j] \neq x[j-1]$, which means that j is privileged. Since the demon is "fairly random", it will point to j in a finite number of commands and increasing j if $j < N$ or making $x[N] = x[0]$ if $j = N$, i.e. firing the exceptional machine. This is true even when the system is not in a legitimate state.

Let us now suppose that system is in an arbitrary state, we show that after a finite number of steps the system will enter a legitimate state. Suppose when the exceptional machine is fired for the first time, it is marked blue and all other nodes are marked white. From then onwards every state created by the exceptional machine or copied from a normal blue machine will also be blue. Since there are at most $N-1$ other white states which machine N can get from the copying actions along the chain of normal processes, the number of times that the the exceptional machine fires while machine N is still in a white state is at most N. Without loss of generality, suppose initially $x[0] = K-1$. If $K > N$, then the first N firings of the exceptional machine would have created blue states from 0 though $N-1$. These blue values form a non-increasing sequence starting from the exceptional machine 0. At the next firing of the exceptional machine we would have $x[0] = N-1$ and $x[N] = N-1$. Since machine N must be blue as well, thus all the blue states must be $N-1$, which means that the system is in a legitimate state. So, the legitimate state is reached from an arbitrary state after at most N firings of the exceptional machine.

The above reasoning uses a centralized powerful demon process. Dijkstra also showed that this methods works even if each machine can adjust itself with a finite speed and a finite frequency.

Although a good example for self-stabilization, some requirements in this problem are not necessarily essential to all self-stabilizing system. Here we want to emphasize that by self-stabilizing we mean any procedure that would lead the system into a legitimate state. In particular, a system need not to be in a legitimate state forever since in case not all machines in the system are reliable, it is highly probable that the system would enter a non-legitimate state as a consequence of failures in the system. However, it is required that the system should have the capability of stabilizing itself to enter a legitimate state again whenever it is in a non-legitimate state. Also, it is not necessary for a machine to get information only from its neighbors or a small subset of the processes in the system. Rather a machine can get information from a majority, even all, of the machines in the system whenever it is convenient or necessary. The latter case is justified as a meaningful self-stabilization since if there are arbitrary failures in the system, a machine will not be able to get the state of the whole system even if it collects state information from all other machines in the system. This is just what happens in Byzantine agreement.

4.2 The Mostly Byzantine Agreement

Results in Chapter 2 show that protocols to reach Byzantine agreement are often very expensive, both in the time required and the number of messages exchanged. On the other hand, it is quite acceptable for some applications not to have to reach agreement every time; rather, agreement is an overall property. For example, in controlling the trajectory of an object, the object may be under the control of several controllers, each with its own sensors which would monitor some

aspects of the environment. From time to time, each controller has to gather information about the environment to make some adjustment, if necessary. It is ideal for all the controllers to get the same information so that the object would traverse the exact trajectory. But there might be some disagreements from time to time, causing the actual trajectory to deviate from the ideal one. However, as long as there is agreement most of the time, the effect of a disagreement need not cause irreversible changes in the state of the environment; rather, it may cause a slight deviation from the ideal trajectory, which can be compensated for by the next adjustment. So, we can eventually get an actual trajectory which is roughly the same as the ideal one within tolerable limits. This suggests that a relaxed Byzantine agreement may be more practical.

Consider a set of processes running indefinitely. Periodically, at times $T, 2T, \cdots$, they try to reach Byzantine agreement based on their current values at that time. They may fail to reach agreement at some time NT, but they should reach agreement *almost* every time. This "almost" property can be represented as

$$\lim_{n\to\infty} \frac{\text{number of disagreements from 0 to } nT}{n} = 0$$

Note, in the above statement of the "almost" property, the number of disagreement can also be infinite. In this paper, we shall consider a more restrictive problem, namely, the Mostly Byzantine agreement. Its requirement can be stated as follows:

(1) *Validity*: if all the reliable processes have the same value at time T, then all the reliable processes should agree on that value at time T.

(2) *Finite Disagreement*: the number of times that reliable processes agree on different values is at most C, a constant depending on n and t only, where n and t are the total number of processes and the number of faulty processes

respectively.

This requirement naturally suggests that self stabilizing techniques can be used to solve the problem. In our approach, self stabilization is achieved by starting with a fully connected network and gradually disconnecting those processes suspected of being faulty from the system. The behavior of faulty processes can be arbitrary and even malicious. The messages used are still oral messages, as in the original Byzantine agreement, which satisfies the following assumptions [LAM 82]:

A1: Every message that is sent is delivered correctly.

A2: The sender of a message can be identified.

A3: The absence of a message can be detected.

4.3 Framework of the Algorithm

Before working on the algorithm, we give the following theorem which states that a redundancy of at least 3 is required to solve the Mostly Byzantine agreement.

Theorem 4.1. *Mostly Byzantine agreement can not be assured for n processes, of which at most t are faulty, if $n < 3t + 1$.*

PROOF. The proof is similar to the impossibility proof on assuring agreement for $3t$ generals and t traitors in [LAM 82]. The only thing that needs to be pointed out is that in addition to the impossibility to assure agreement, it is also impossible to detect which processes are faulty. We omit the proof here. ∎

The framework of such self stabilizing algorithms is given in Fig. 4.1. It works synchronously. At every time $T, 2T, \cdots$, they try to reach agreement by two rounds of message exchanges. In the first round, each process broadcasts its own value and receives values from other processes. There are two special values, "halt" and "absent". If a process thinks that some process is faulty, i.e., in its traitor set,

```
procedure Self_Stablizing_BA(p, n, t);
begin
  Traitor_Set_p := φ;
  repeat /* at time, ···, -T, 0, T, 2T, ···, */
  round1
       broadcast(v_p);
       receive(v_*);
       V_p := (v_1, ..., v_n);
/* v_i = "halt", if i ∈ Traitor_Set_p */
/* v_i = "absent", if p can not receive i's message of that round */
  round2
       broadcast(V_p);
       receive(V_*);
       for i := 1 to n do
       begin
         d_i := Select_Majority(V_1(i), ..., V_{i-1}(i), V_{i+1}(i), ..., V_n(i));
         S_p(i) := {j | V_p(i) ≠ V_j(i)};
         Judge_Reliable(p, i);
       end
       ⟨decision_p := majority(d_1, ..., d_n)⟩
  forever
end;
procedure Judge_Reliable(p, i)
begin
  if |S_p(i)| ≥ t + 1 then
     Traitor_Set_p := Traitor_Set_p + {i}
end;
```

Fig. 4.1 A framework for self stabilizing Byzantine Agreement algorithms.

it uses "halt" as that process's value. "absent" is taken as the default value for those processes whose values can not be received by some specified time. In the second round, each process broadcasts the values it has received in the first round. Based on the values received in the second round, each process p construct the set $S_p(i) = \{j \mid V_p(i) \neq V_j(i)\}$. $V_p(i)$ refers to the ith element of vector V_p. This set contains the processes which are considered to have been "cheated" by process i. *Judge_Reliable* determines if a process is faulty by looking at how many processes are cheated by that process. If a process is judged by process p to be faulty because it cheated too much, it is placed in the set $Traitor\_Set_p$. Each process chooses d_i, any other process i's value at that time, by applying the $Select\_Majority$ function to the set $\{V_1(i), \ldots, V_n(i)\}$, which are i's value relayed by processes in the system.

The agreement value is obtained by applying some kind of majority function to the set $\{d_1, \ldots, d_n\}$.

In the algorithm, we leave function *Select_Majority* unspecified. When choosing *Select_Majority*, we can either let "halt" and "absent" be replaced by some default value and allow each process to use the same threshold for majority, or let "halt" and "absent" be omitted and permit each process to use a different threshold. A process is determined to be faulty if it has cheated too many processes and/or it fails to send messages at some time. One interesting thing about this algorithm is that, depending on the redundancy in the system, we can get various different converging speeds, by properly designing the function *Select_Majority*.

4.4 Two Algorithms

The definition of *Select_Majority* in Fig. 4.2 is for $n \geq 4t$. It "converges" very fast in the sense that if a faulty process is so "ill" that it would cause disagreement, it would then be judged by all reliable processes to be faulty and disconnected from every reliable process.

```
function Select_Majority(v_1, v_2, ..., v_{n-1});
begin
  for i := 1 to n do
    if v_i = "halt" or v_i = "absent" then v_i := DEFAULT;
    if ∃v : v ∈ {v_1, ..., v_{n-1}, DEFAULT} & count(v) ≥ ⌈n/2⌉
      then Select_Majority := v
      else Select_Majority := DEFAULT
end;
```

Fig. 4.2 *Select_Majority* definition for $n \geq 4t$.

Theorem 4.2. *The algorithm in Fig. 4.1 and Fig. 4.2 assures Mostly Byzantine agreement, if* $n \geq 4t$.

PROOF. First, we claim that by *Judge_Reliable*, no reliable process would ever be disconnected from the system. To see this, observe that a reliable

process i's value would be wrongly relayed by at most t processes. So, any reliable process p would get at least $n-t$ identical copies of that process's value. This implies that $|S_p(i)| \leq t$, which means no reliable process would ever determine that another reliable process is faulty.

The validity property is easy to prove. Since any reliable process i's value will be relayed by at least $n-t$ reliable processes, and each reliable process will get at least $n-t$ identical copies of that value, well beyond $\lceil n/2 \rceil$ for $n \geq 4t$, it thus gets the same d_i. Because all the reliable processes have the same value, each process would get the same majority based on the set $(d_1, \ldots, d_n)$.

Now, we prove that if no agreement is reached at some time, then at least one faulty process will be disconnected by all the reliable processes. By the algorithm in Fig. 4.1, the only possible reason that causes disagreement is that two reliable processes can not agree on some faulty process i's value, i.e., they have different d_i. This happens only if there is no value which appears at least $\lceil n/2 \rceil$ times in the set $\{V_p(i) \mid p \textit{ is reliable}\}$. This implies for every reliable process p,

$$\begin{aligned} |S_p(i)| &\geq n - t - \lceil n/2 \rceil + 1 \\ &\geq t + 1, \qquad \text{for } n \geq 4t \end{aligned}$$

So, i will be judged by every reliable process to be faulty and disconnected from the system.

If a faulty process is disconnected from every reliable process, it would not cause disagreement any more, since every reliable process would use the same default value for that faulty process. Since any disagreement at any time would cause at least one faulty process to be removed from the system and there is a finite number of faulty processes, the number of disagreements is certainly finite. ∎

It is delightful to see what we have right now. At each time, only two rounds of message exchanges are needed to guarantee that either agreement is reached or at least one faulty process is removed. The removal of a faulty process ensures that a faulty process can not invert the agreement any more and therefore makes agreement easier. Also, there is no limit on the possible values of message items. A multi-value agreement can use exactly the same algorithm as a binary value agreement, except they need a different *DEFAULT* value. If a binary value is used, we can let each process just send a vector of binary values in the second round, by using a default value (0 or 1) for those considered to be faulty, instead of using "halt" and "absent". A process sends n bits in the first round, and n^2 bits in the second round. So at most $n(n^2 + n)$ message bits are transmitted in total.

One limitation of this algorithm is that it works for $n \geq 4t$, which is quite large compared to those required by algorithms for the original Byzantine agreement problem, namely $n \geq 3t+1$. However, in our algorithm, the convergence speed is fast enough that there are at most t disagreements in case of t faulty processes. In other words, a faulty process can invert the agreement at most once before it is removed from the network. This suggests that there might be some gradual detection of faulty processes when $3t + 1 \leq n < 4t$, such that if there is a disagreement, at least one faulty process will get disconnected from at least one reliable process. In a system of n processes and t faulty processes, the number of connections between a reliable process and a faulty process is at most $t(n - t)$. By assuring at least one break of such connections per disagreement, the number of disagreements must be finite. Fig. 4.3 presents such an algorithm that assures the Mostly Byzantine agreement for $n \geq 3t + 1$.

Theorem 4.3. *The algorithm in Fig. 4.1 and Fig. 4.3 assures Mostly Byzantine agreement, if* $n \geq 3t + 1$.

```
function Select_Majority(v_1, v_2, ..., v_{n-1});
begin
  H := {v_i | v_i = "halt"};
  V := {v_1, ..., v_{n-1}} - H;
  f := |H|;
  if f ≥ 2t + 1 then Select_Majority := DEFAULT;
  else if ∃v : v ∈ V &count(v) ≥ ⌈(n - f)/2⌉ then
       Select_Majority := v
    else Select_Majority := DEFAULT
end;
```

Fig. 4.3 *Select_Majority* definition for $n \geq 3t + 1$.

PROOF. Using the same arguments as in the proof of Theorem 4.2, we can show, under the *Judge_Reliable* procedure, that no reliable process would ever be disconnected from another reliable process and validity is achieved by the definition of *Select_Majority*.

To prove the finite disagreement property, we just need to prove that if some reliable processes do not agree on some faulty process i, then at least one more connection between a reliable process and i would be broken. Suppose at some time T, there are m reliable processes which have broken their connection with i, $0 \leq m$. Cases for $m \geq t + 1$ are trivial, since in these cases, at least $t + 1$ of the reliable processes would relay "halt" as the value of process i. If any process p has a value of i other than "halt", i.e., it has a connection with i, then $|S_p(i)| \geq count(\text{"}halt\text{"}) \geq t + 1$, so all the reliable processes would break their connection with i.

Now, consider the case of $0 \leq m \leq t$. Let $r = n - t$, the number of reliable processes. Any reliable processes p and q, which still have connection with i will receive at least $r - m$ non-"halt" values of i relayed by other reliable processes, but no more than $r - m + t$. If all those $r - m$ values relayed by reliable processes have the same value, then they should agree on that value, since $r - m \geq \lceil (r - m + t)/2 \rceil$, for $r \geq 2t + 1$ and $m \leq t$. Otherwise, there must be an $m_1 > 0$, such that m_1 of those $r - m$

reliable processes have the same value and the rest $r - m - m_1$ have some other values. We claim that

$$r - m - m_1 \geq t - m + 1$$

or

$$m_1 \geq t - m + 1$$

Since $count(\text{"}halt\text{"}) \geq m$ for any process p of those $r-m$ reliable processes, the above implies either m_1 or $r-m-m_1$ of those $r-m$ processes will have $|S(i)| \geq t-m+1+count(\text{"}halt\text{"}) \geq t-count(\text{"}halt\text{"})+1+count(\text{"}halt\text{"}) \geq t+1$, thus disconnecting i.

To see why the above claim holds, suppose

$$r - m - m_1 < t - m + 1$$

and

$$m_1 < t - m + 1.$$

This implies

$$\begin{aligned} m_1 + r - m - m_1 &= r - m \\ &< 2t - 2m + 1. \end{aligned}$$

Contradicting to $r \geq 2t + 1$. ∎

If binary values are used, the above algorithm uses at most $n(n^2 + n)$ message bits in total.

The converging speed can be analysed in the following way. Let $r^{(i)}$ denote the number of reliable processes having connection with process p after i disagreements have been caused by process p, $d^{(i)}$ the number of reliable processes disconnecting from process p at that time, $h^{(i)}$ the threshold, and $new\_d^{(i)}$ the number of reliable processes that have newly disconnected p after the ith disagreement caused by p.

For $n \geq 3t + 1$, we have the following relations

$$d^{(0)} = 0$$

$$r^{(0)} = n - t$$

$$\lceil (n-t)/2 \rceil \leq h^{(0)} \leq \lceil n/2 \rceil$$

and

$$new\_d^{(i)} \geq r^{(i-1)} - h^{(i-1)} + 1$$

$$d^{(i)} = d^{(i-1)} + new\_d^{(i)}$$

$$r^{(i)} = r^{(i-1)} - new\_d^{(i)}$$

$$\lceil (n - d^{(i)} - t)/2 \rceil \leq h^{(i)} \leq \lceil (n - d^{(i)})/2 \rceil$$

The number of disagreements that can be caused by any faulty process is the smallest $s + 1$, such that

$$\sum_{i=1}^{s} new\_d^{(i)} \geq t + 1$$

for t faulty processes and s is less than $t/2$, for $t \geq 8$. When $n \geq 4t$, this algorithm has the same convergence speed as the first algorithm.

It might be helpful at this point to construct some scenarios to explain how this algorithm works. First, we give an example to show that the Mostly Byzantine agreement can not reached by this algorithm if $n < 3t + 1$. Consider 30 processes, 10 of which are faulty. Suppose faulty process 1 sends 1 to group A of 10 reliable processes and 0 to group B of the other 10 reliable processes. Then all faulty processes relay 1 to group A and relay 0 to group B. In this case, Group A will agree on 1 and group B will agree on 0, a disagreement. But no reliable process could ever detect that process 1 is faulty.

Now, assume there are 31 processes in total and the number of faulty processes is still 10. By our argument above, at least 6 reliable processes would disconnect from process 1 at the first disagreement caused by process 1. Similarly, we can get

at the 2nd disagreement, 3 reliable processes will disconnect process 1;

at the 3rd disagreement, 2 reliable processes will disconnect process 1;

at the 4th disagreement, all reliable processes are disconnected from process 1.

After the third disagreement, there are 11 processes disconnected from process 1, therefore, it will be disconnected by all the reliable processes. After the fourth disagreement, all reliable processes will agree on some default value for that faulty process. Thus, any process can invert an agreement at most 4 times in this case.

4.5 Extensions

4.5.1 Reducing the Number of Messages. The above two algorithms require n message exchanges per round for n processes. When $n \geq 4t$, we may have two choices. One is to let every process participate in the whole procedure and have a fast convergence speed. The other one is to just let $3t+1$ processes participate in the decision procedure and add a third round each time in which $2t+1$ of those processes broadcast their decisions to the bystanding processes. The latter one may have a slower convergence speed. So, we have a tradeoff between the number of messages exchanged per round, the number of rounds each time and the convergence speed. If $n >> t$, the saving of messages in the second choice is significant. Another consideration is that in case the distances between the processes are not balanced, it may be more practical to just let those closely connected processes to make the decision after each process has broadcast its value. In this way, a group of those closely connected processes forms the kernel of the system and when any process in this kernel is identified as faulty, it may be replaced by a process not in the kernel.

In the above algorithms, each process requires at least $3t$ messages, one directly from a particular process while $3t - 1$ are relayed by other processes, to make a decision on that process. We do not know whether this amount is minimal. In

the case of only $3t$ processes or if every process has the $3t-1$ messages relayed by the same set of processes (consider the one received directly from the transmitting process as relayed by itself), then it is possible that no process is able to detect the failure of the transmitting process while there is a disagreement. However, if each process has a different set of $3t-1$ relaying processes, then at least one process can detect the failure of a process that causes a disagreement. To see this, consider a system in which process p_{n-1} is faulty, each process p_i collects p_{n-1}'s value from processes $p_i, p_{i+1}, \ldots, p_{i+3t-2}$ (all the additions here are mod $n-1$). Suppose that there is a disagreement, though no reliable process has detected that p_{n-1} is faulty. In this case, there must two reliable processes such that they decide on different values and are separated by faulty processes only. Without lost of generality, suppose that p_i decides on 0 (receiving at least $\lceil 3t/2 \rceil$ 0's) and p_j decides on 1, $j = i + t_1 + 1$, $t_1 \leq t$, and processes $p_{i+1}, p_{i+2}, \ldots, p_{j-1}$ are faulty. Further, suppose that n is large enough, which means that there are a sufficient number of processes on the path from p_j to p_i. Since p_i agrees on 0 and does not detect that p_{n-1} is faulty, then it receives at most t 1's, i.e., at least $2t-1$ of the messages it receives are 0, among them at least $2t-2-t_1-t_2$ will also be received by p_j, where t_2 is the number of faulty processes between p_j and p_{i+3t-2}. In order for p_j to decide on 1 and not to detect that $p_{n-1}$is faulty, $2t-2-t_1-t_2$ must be less than $t+1$. This implies that $t_1+t_2 \geq t-2$. So, there is at most one faulty process on the path from p_{i+3t+1} to p_i. In the process set $\{p_{i+3t-1}, \ldots, p_{i+4t}\}$, there must be two reliable processes p_k and p_l, which are separated by at most one faulty process, such that they relay different values. Otherwise some reliable process in $\{p_j, \ldots, p_{i+3t-2}\}$ would detect that process p_{n-1} is faulty. Then it comes out that either p_k or p_l, or both, would detect that p_{n-1} is faulty, as long as $i + 6t + 1 \leq i \bmod (n-1)$ for $n \geq 6t+3$, a contradiction.

4.5.2 Error Recovery. The algorithms presented above assume that the reliable processes will be reliable forever. This assumption is not always true in the sense that if the system is assumed to run for an infinite time, any process would have a sufficiently high probability of becoming faulty. This makes rapid detection and repair of faulty processes mandatory. Practically, it should be allowable that in some period one group of processes is faulty and in another period another group becomes faulty while all the previously faulty processes have been repaired or replaced by reliable processes, as long as the required redundancy is maintained at any time. Our self stabilizing approach would ease the detection of faulty processes. In the algorithm, whenever a reliable process identifies that some process is faulty, it breaks its connection with that faulty process. By properly inspecting the topology of the connections in the system, a faulty process can be easily detected. For example, in a process control system, where manual intervention is possible, a process can break its connection with another process simply by switching to ground. A human operator inspects the connection from time to time and removes some process when its active connection is below certain threshold ($n - t$ in our case). A repaired process can be reintegrated into the system by letting all the other process switch their connection back.

In the case where manual intervention is not feasible, we can let each process be backed up by several processes. When a process detects that another process is faulty, it switches its connection with that process to the spare process. The algorithm in Fig.4.4 assumes that each process is backed by s spare processes. When a reliable process exhausts all the spare processes for process p, it declares that p is a faulty process and takes "halt" as the default value for p afterwards. If some kind of self-repair is provided in each process, the repaired process could be recycled and used as a spare process for the currently active process. Again, we have to assume that each time the algorithm is activated, a minimal redundancy of

```
procedure Self_Stablizing_BA_With_Spares(p, n, t, s);
begin
  Traitor_Set_p := φ;
  repeat /* at time, ···, -T, 0, T, 2T, ···, */
  round1
       broadcast(m_p);
       receive(m_*);
       M_p := (m_1, ..., m_n);
/* m_i.value = "halt", if i ∈ Traitor_Set_p */
/* m_i.num = ∞, if i ∈ Traitor_Set_p */
/* v_i = "absent", if p can not receive i's message of that round */
  round2
       broadcast(M_p);
       receive(M_*);
       for i := 1 to n do
       begin
         d_i := Select_Majority(M_1(i), ..., M_{i-1}(i),
                                M_{i+1}(i), ..., M_n(i));
         S_p(i) := {j | (M_p(i).value ≠ M_j(i).value&M_p(i).num
                         = M_j(i).num or M_j(i).num > M_p(i).num};
         Judge_Reliable(p, i);
         ⟨decision_p := majority(d_1, ..., d_n)⟩
       end
  forever
end;

procedure Judge_Reliable(p, i)
begin
  if |S_p(i)| ≥ t + 1 then
    if M_p(i).num < s then switch to i_{num+1}
    else Traitor_Set_p := Traitor_Set_p + {i}
end;

function Select_Majority(m_1, m_2, ..., m_{n-1});
begin
  H := {m_i | m_i.value = "halt"};
  V := {m_1, ..., m_{n-1}} - H;
  f := |H|;
  if f ≥ 2t + 1 then Select_Majority := DEFAULT;
  else if ∃v, i, : m.value = v&m.num = i&m ∈ V
       &count(m) ≥ ⌈(n - f)/2⌉ then Select_Majority := m.value
  else Select_Majority := DEFAULT
end;
```

Fig. 4.4 A stabilizing Byzantine Agreement algorithms with s spares for each process.

3 is required.

CHAPTER 5

Summary

The main focus of this thesis is on techniques for tolerating *arbitrary* failures in distributed computing systems. Central to this theme are solutions to Byzantine agreement and their various applications. There is no doubt that a truly fault tolerant system can not be built by using a finite amount of hardware. However, among various failure models used to design fault tolerant systems, the one considered in the Byzantine agreement problem is the least restrictive and most powerful.

It has been a traditional approach to implement a reliable system by duplicating the processors to compute the same result and then to perform a majority voting on their output values to obtain a single value. An important assumption in this approach is that all nonfaulty processors will produce the same output. This is true as long as all reliable processors use the same input. However, if the input datum is from a single physical component and this component happens to be faulty, it is quite possible that each processor may get a different input value. Further, different processors can get different values even from reliable components if the input values change with time. For example, if two processors are reading a clock which is advancing then one may get the old time and the other the new time.

In order for majority voting to yield a reliable system, the following two conditions should be satisfied:

(1) All reliable processors should use the same input value.

(2) If the input component is reliable, then all reliable processors should use the value of the input component.

These are precisely the requirements in the Byzantine generals problem with the input device acting as the commander.

Another assumption used in the majority voting method is that every voting processor has the same set of values to work with. For a nonfaulty processor this poses no problems. However, if a faulty processor's value will be distributed to all other processors, there is no way to guarantee that every nonfaulty processor will receive the same value. Byzantine generals strike again in situations like this.

One might question the justification of Byzantine agreement by circumventing the problem with a "hardware" solution. For example, one may try to ensure that all processors obtain the same input value by having them all read it from the same wire. However, a faulty wire may place a marginal signal along the wire such that different processors may interpret the value differently. In this case, different processors must still communicate with each other to agree on that value by solving the Byzantine generals problem. Another more tempting attempt is to use a broadcast network, such as Ethernet, for the input component to distribute the value. Such a network assumes that 1) there exists a single route between any given processor and all other processors and 2) messages never need to be forwarded (except by repeaters at gateways). In this way, the behavior of a faulty process is limited. When a processor tries to send one value to all the other processors, it simply places it on the outgoing channel and and every other processor can read the value from that channel. In this way, the problem of unanimity seems to disappear. However, we still have reasons to justify the importance of Byzantine agreement. First, such a network may have failures itself. For example, the channel may break and leave the network partitioned, the link between a processor and the channel may break or even corrupt the message it gets from the channel when delivering the message to the connected processor. To enhance the reliability, one might try to duplicate the channels and the links so that any pair of processors are connected by at least one nonfaulty path of channels and links. In this case, we encounter another kind problem, although simpler than the Byzantine agreement, that is the

transmitter may place different values on different channels and it is possible for the links to corrupt the messages. Solutions to these problem are discussed in [BAB 85]. Second, Byzantine agreement does not require any special "hardware". It simply requires a point to point communication network and failures in the underlying network are modeled as a failure of the connected processor. Solutions to Byzantine agreement can be considered as a "pure" software solution.

Of course, the model in Byzantine agreement can not fully characterize the behavior of faulty processes in a system. In the study of Byzantine agreement, a process is assumed to be faulty if it behaves inconsistently to different processes, and is considered to be reliable otherwise. So, an inherently faulty process is taken as reliable as long as if it behaves consistently. The Byzantine agreement can do nothing to faulty processes like this. However, a protocol for Byzantine agreement enables all processes in the system to have the same view of the faulty process. This would ease the design of redundancy at higher levels. For example, if the input is an important one, there should be several separate input devices providing redundant values. Each input device initiates a Byzantine protocol so that every processing devices will receive the same set of input values. If the majority of the input devices are nonfaulty, then the reliable processing devices will use the correct values and produce the same output.

If the input device is nonfaulty but gives different values because it is read while the value is changing, it is still desirable that all nonfaulty processors should obtain a reasonable input value such that the values obtained by nonfaulty processors lie within a tolerable range if the input unit produces a reasonable range of values. This is exactly the case for the approximate Byzantine agreement wherein each process's initial value is taken as the value read from the changing input device.

The authenticated Byzantine agreement uses a more restrictive model. It requires that processors be able to sign their messages in such a way that a nonfaulty

processor's signature can not be forged by any faulty processor. This is a property that can never be guaranteed, since a signature is merely a data item and a faulty processor is free to generate any data item. This makes solutions using digital signatures more like randomized algorithms. However, we can make the system as reliable as we wish by making the probability of forging a nonfaulty processor's signature as small as we wish. Suppose that the faulty processor malfunctions randomly, then the probability that a random malfunction in a processor generates a correct signature is essentially equal to the probability of its doing so through a random choice procedure, i.e., the reciprocal of the number of possible signatures. If the faulty processor is being guided by a maliciously intelligent entity—for example, a perfectly good processor being operated by a human who is trying to disrupt the system— the problem of constructing the signature becomes a cryptography problem.

The application of Byzantine agreement is largely due to the implementation of reliable broadcasting. Such a kind of broadcasting requires that whenever a message is sent, either all or none of the receiving processes will receive that message. This is just the requirement for the Byzantine generals problem. Previous implementations of reliable broadcasting can only tolerate failures of omission, i.e., a process can only fail by not sending a message. A reliable broadcasting implemented using Byzantine agreement can tolerate arbitrary failures. In Chapter 3, we have reviewed some of the typical applications of Byzantine agreement. Clock synchronization was one of the first applications and has been frequently cited as an example of Byzantine agreement. The implementation of fail-stop processor adds another fine account on Byzantine agreement. The "clean" failure behavior of fail-stop processors can be expected to facilitate design and programming of fault tolerating systems. The state machine method, another design methodology, is also an application of Byzantine agreement. Due to the expensiveness of Byzantine agreement, this method may

not be used widely in some system, but it can implement a very reliable kernel at moderate cost. The applications of Byzantine agreement in distributed database system are somewhat limited. Due to the criticality of input/output nodes, there is always a tradeoff between reliability and user friendliness.

The Mostly Byzantine agreement is presented in the belief that protocols to reach Byzantine agreement are too expensive to be used in some real time systems where fast response is preferable to a totally failure free but inefficient system. When constructing a reliable system, the self stabilizing approach may not be suitable if each agreement is crucial or if there are only a few attempts to reach agreement in the whole process. However, if reaching agreement is a consecutive procedure and each disagreement is not too irreversible to cause the failure of the whole system, say, it only produces a defective product, this approach achieves a high degree of fault tolerance with relatively low cost. Also, the finite number of disagreements during the whole process of Byzantine agreement suggests that the method in this paper is applicable even to crucial applications if some high level design redundancy is deployed to tolerate a certain number of disagreements.

In a sense, the Mostly Byzantine agreement can be considered as a kind of randomized Byzantine agreement since there is no guarantee that agreement could be reached at some specific time. However, unlike randomized Byzantine agreement algorithms where a disagreement probability of ε exists each time when trying to reach agreement, no matter how many disagreements have occurred before, the self stabilizing approach assures, at least in theory, that the chance of having disagreement at each time becomes smaller, whenever a disagreement occurs, and eventually becomes zero. The reason is that in randomized algorithms, as well as in deterministic algorithms, no measures are taken to detect the presence of faulty processes, while in the self stabilizing method, the detection of faulty processes is carried out at the same time as trying to mask their illness.

For the handful of algorithms presented in previous chapters, we feel the one by Lynch *et al.* is certainly the best available deterministic one. This algorithm requires a relatively large number of rounds. If early-stopping techniques are used, a better performance can be expected. Although the one by Lamport *et al.* has reached the $t + 1$ lower bound of rounds for $t + 1$ faulty processes, it requires an exponential number of messages to be exchanged, and hence usually serves as an existence proof. For randomized algorithms, the one proposed by Chor and Coan seems best since it does not use any additional resources and can terminate in a constant expected number of rounds. But it fails to cope with asynchronous systems. Rabin/Perry's algorithms can work with asynchronous systems, but require an additional trusted dealer to distribute the random coin tosses in advance. If reaching agreement is in consecutive attempts, the Mostly Byzantine agreement algorithm is certainly the best one for achieving a high degree of fault tolerance at a relatively low cost.

Bibliography

[AND 82] Anderson, T., and Lee, P. A. Fault tolerance terminology proposals. In *12th Annual International Symp. on Fault-Tolerance Comput. Syst.*, June, 1982, pp. 29-33.

[BAB 85] Babaoğlu, Ö., and Drummond, R. Streets of Byzantinum: Network architecture for reliable broadcast. *IEEE Trans. on Software Engineering*, vol.SE-11, no. 6, (June, 1985), pp. 546-554.

[BEN 83] Ben-Or, M. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proc. 2nd Annu. ACM Symp. Principles of Distributed Comput.*, Aug. 1983. pp. 27-30.

[BRU 85] Brucha, G., and Toueg, S. Asynchronous consensus and broadcast protocol. *JACM*, vol. 32, no. 4, (Oct. 1985), pp. 824-840.

[CHO 85] Chor, B., and Coan, B. A. A simple and efficient randomized Byzantine agreement algorithm. *IEEE Trans. on Software Engineering*, vol. SE-11, no. 6, (June, 1985), pp. 531-539.

[DIJ 74] Dijkstra, E. W. Self-stabilizing systems in spite of distributed control. *CACM*, vol. 17, no. 11, (Nov. 1974), pp. 643-644. Also as EWD 391 In *Selected Writings on Computing: A Personal Perspective* by E. W. Dijkstra, Springer-Verlag, New York, 1982, pp. 41-46.

[DOL 82] Dolev, D. The Byzantine Generals strike again. *J. of Algorithms*, vol. 1, (Jan. 1982), pp. 14-30.

[DOL 82a] Dolev, D., Fischer, M. J., Fowler, R., Lynch, N. A., and Strong, H. A. An efficient algorithm for Byzantine agreement without authentication. *Information and Control*, vol. 52, no. 3, (Jan. 1982), pp. 257-274.

[DOL 82b] Dolev, D., and Strong, H. A. Polynomial algorithms for multiple processor agreement. In *Proc. 14th ACM Symp. on Theory of Computing*,

1982, pp. 401-407.

[DOL 86] Dolev, D., Lynch, N. A., Pinter, S. S., Stark, E. W., and Weihl, W. E. Reaching approximate agreement in the presence of faults. *JACM*, vol. 33, no. 3, (July 1986), pp. 77-97.

[FIS 82] Fischer, M. J., and Lynch, N. A. A lower bound for the time to assure interactive consistency. *Inform. Processing Letters*, vol. 14, no. 4, (June 1982), pp. 183-186.

[FIS 85] Fischer, M. J., Lynch, N. A., and Paterson, M. S. Impossibility of distributed consensus with one faulty process. *JACM*, vol. 32, no. 2, (April, 1985), pp. 374-382.

[GAR 84] Garcia-Molina H., Pitteli, F., and Davidson, S. Applications of Byzantine agreement in database systems. Technical Report, TR-315 (revised), Dept. Comput. Sci, Princeton, Univ., June, 1984.

[HAR 72] Harary, F. *Graph Theory*, Addison-Wesley, Reading, MA. 1972.

[HOA 69] Hoare, C. A. R. An axiomatic basis for computer programming. *CACM*, vol. 12, no. 10, (Oct., 1969), pp. 576-580.

[KAR 76] Karp, R. The probability analysis of some combinational algorithms. In *Algorithm and Complexity: New Directions and Recent Results* (J. Traub, ed.), Academic Press, New York, 1976, pp. 1-19.

[LAM 82] Lamport, L., Shostak R., and Pease M. The Byzantine Generals problem. *ACM Trans. on Programming Languages and Systems*, vol. 4, no. 3, (July, 1982), pp. 382-401.

[LAM 78] Lamport, L. Time, clocks, and the ordering of events in a distributed system. *CACM*, vol. 21, no. 7, (July, 1978), pp. 558-565.

[LAM 83] Lamport, L. The weak Byzantine Generals problem. *JACM*, vol. 30, no. 3, (July, 1983), pp. 668-676.

[LAM 84] Lamport, L. Using time instead of timeout for fault-tolerant dis-

tributed systems. *ACM Trans. on Programming Languages and Systems*, vol. 6, no. 2, (April, 1984), pp. 254-279.

[LUN 84] Lundelius, J., and Lynch, N. A. A new fault-tolerant algorithm for clock synchronization. In *Proc. 3rd ACM Symp. on Principles of Distributed Comput.*, 1984, pp. 75-88.

[LYN 82] Lynch, N. A., Fischer, M. J., and Fowler, R. J. A simple and efficient Byzantine Generals algorithm. In *IEEE Symp. on Reliability in Distributed Software and Database Systems*, 1982, pp. 46-52.

[PEA 80] Pease, M., Shostak, R., and Lamport, L. Reaching agreement in the presence of faults. *JACM*, vol. 27, no. 2, (April, 1980), pp. 228-234.

[PER 85] Perry, K. J. Randomized Byzantine agreement. *IEEE Trans. on Software Engineering*, vol. SE-11, no. 6, (June, 1985), pp. 539-546.

[PET 85] Peterson, J., and Silberschatz, A. *Operating System Concepts* (2nd ed.), Addison-Wesley, Reading, 1985.

[RAB 76] Rabin, M. Probability algorithms. In *Algorithms and Complexity: New Directions and Recent Results* (J. Traub, ed.), Academic Press,, New York, 1976, pp. 21-39.

[RAB 83] Rabin, M. Randomized Byzantine generals. In *Proc. 24th Symp. Foundations of Comput. Sci.*, Nov, 1983, pp. 403-409.

[RIC 81] Ricart, G., and Agrawala, A. K. An optimal algorithm for mutual exclusion in computer networks. *CACM*, vol. 24, no. 1, (Jan. 1981), pp. 9-17.

[SCH 83] Schlichting, R. D., and Schneider, F. B. Fail-stop processors: An approach to designing fault tolerant computing system. *ACM Trans. on Comput. Syst.*, vol. 1, no. 3, (Aug. 1983), pp. 222-237.

[SCH 84] Schneider, F. B. Byzantine Generals in action: Implementing fail-stop processors. *ACM Trans. on Comput. Syst.*, vol. 2, (May, 1984),

pp. 145-154.

[SHA 79] Shamir, A. How to share a secret. *CACM*, vol. 22, no. 11, (Nov. 1979), pp. 612-613.

[THE 83] Theaker, C. J., and Brookes, G. R. *A Practical Course on Operating Systems*, Springer-Verlag, New York, 1983.

[TUR 84] Turpin, R., and Coan, B. A. Extending binary Byzantine agreement to multivalued Byzantine agreement. *Inform. Processing Letters*, vol. 18, no. 2, (Feb. 1984), pp. 73-76.