A Continuum of Failure Models for Distributed Computing*

Juan A. Garay[†] Kenneth J. Perry[‡]

Abstract

A range of models of distributed computing is presented in which processors may fail either by crashing or by exhibiting arbitrary (Byzantine) behavior. In these models, the total number of faulty processors is bounded from above by a constant t subject to the proviso that no more than $b \le t$ of these processors are Byzantine. At the two extremes of the range (i.e., $b = 0$ or $b = t$) we get models that are equivalent to the traditional models of either pure crash failures or pure Byzantine failures. For $0 < b < t$, the models that we introduce accommodate "real-world" experience that shows that the overwhelming majority of failures are crashes but occasionally some number of less-restrictive failures occur.

We examine the Reliable Broadcast, Consensus and Clock Synchronization problems within this new family of models and prove lower bounds on the relationship required between the number of processors, t, and b. We also present protocols to solve these problems, which match the lower bounds. In presenting the protocols, we emphasize new algorithmic techniques that are fruitful to use in the new models but which have limited value in either of the pure models.

Key words: Hybrid failure model, Byzantine agreement, reliable broadcast, clock synchronization, fault tolerance.

---

*Parts of this paper appeared under the same title in *Proc. 6th International Workshop on Distributed Algorithms*, LNCS Vol. 647, Springer-Verlag 1992 [13], and as "Synchronizing Clocks in a System Resilient to a Wide Class of Failures" in *Proc. IEEE 1994 Workshop on Fault-Tolerant Parallel and Distributed Systems*, Computer Society Press, 1995 [14].

[†]Bell Laboratories – Lucent Technologies, 600 Mountain Ave., Murray Hill, NJ 07974. E-mail: `garay@research.bell-labs.com`. URL: `www.bell-labs.com/user/garay/`. Work done while the author was at the IBM T.J. Watson Research Center, Yorktown Heights, NY.

[‡]Goldman Sachs, 85 Broad Street, New York, NY 10004. E-mail: `ken.perry@gs.com`. Work done while the author was at the IBM T.J. Watson Research Center, Yorktown Heights, NY.

1 Introduction

Suppose that we are to design a distributed computing system tolerant of up to t processor failures. Key problems in such systems are achieving consensus and performing reliable broadcast. If the failures are mild, i.e., solely processor crashes, then known results [21] indicate that the total number of processors required to solve these problems merely has to exceed t. But if failures are severe, i.e., arbitrary "Byzantine" behavior, then the lower bounds [22] indicate that more than $3t$ processors are required.

Experience demonstrates that processor hardware failures are predominantly, but not exclusively, crashes. Moreover, software subsystems on the same processor may function or crash independently on one another, thereby producing an overall processor behavior more malevolent than a simple crash. E.g., some services may continue to function – albeit with incorrect clocks – in spite of a crashed time-server. Thus, our dilemma is either to assume the best (that *only* crashes will occur), use low redundancy, and possibly suffer occasional catastrophic results, or to assume the worst (i.e., all failures are Byzantine) and pay a high cost in processors in order to tolerate rare events.

In this paper, we propose a range of failure models that allows a mix of crash and Byzantine failures. In these models, the total number of faulty processors is bounded from above by a constant t subject to the proviso that no more than $b \leq t$ of these processors are Byzantine. At the extremes of the range (i.e., $b = 0$ or $b = t$) the model turns into either the pure crash fault or pure Byzantine models. Within this range, we examine three problems that lie at the core of distributed computing: performing reliable broadcast, achieving consensus, and synchronizing clocks.

Previous and related work. The Agreement problems (i.e., Byzantine Agreement and Consensus) [22, 26] introduced a model in which the behavior of a faulty processor could be "Byzantine" (i.e., not restricted in any way), even to the extent of giving the impression that faulty processors were colluding to foil agreement. In this pure Byzantine model, it was shown that the Agreement problems admitted solutions if and only if n, the total number of processors, exceeded three times t, the upper bound on number of faulty processors. A less restricted model in which processors fail only by halting (i.e., crash failures) has also been used to study these problems and several protocols requiring only $n > t$ have been found [21].

It has been recognized that processors fail in ways that are less restrictive than crashing and more restrictive than acting arbitrarily. A "send omission" model in which processors can fail either by halting or failing to send some messages was proposed by Hadzilacos [16] as a model of intermediate power. This model was subsequently made even less restrictive by permitting faulty processors the additional power of arbitrarily failing to receive messages [27]. In both these models, Agreement protocols were presented that only required $n > t$. However, the failure types admitted in these weaker models of failure are still considered "benign" in the sense that any message sent by a processor is correct relative to the protocol and the processors state. In contrast, the messages sent by a faulty processor in the Byzantine model is in no way constrained.

In [31, 25, 4], mechanisms are provided that restrict the behavior of faulty processors in the Byzantine model. Still, these mechanisms require $n > 3t$ and lower bounds have been established [4] that show that such mechanisms multiply the running time of algorithms designed for the weaker models.

Independently—but previously,[1] hybrid failure models and algorithms for the Agreement problems had also been considered by Thambidurai and Park [33] and Meyer and Pradhan [24]. In [33]

[1]At the time we did this work, circa 1991/92, we were unaware of the existence of any related work.

the faults are partitioned into non-malicious and malicious (Byzantine), and in turn the malicious faults are partitioned into *symmetric* and *asymmetric* (asymmetric faults refer to the case when a message is not received identically by all the non-faulty processors). They adapt the OM ("Oral Messages") algorithm of [22] to this setting. (This algorithm was later shown to be flawed—and corrected—by Lincoln and Rushby [23].) In [24] the failure mix that is considered—"dormant" and Byzantine—is similar to the one in this paper, and the same necessary bounds for number of processors and network connectivity are derived. The consensus algorithms they present are "re-parameterizations" of the algorithms of [9] and [22]. Additional work on mixed failure models include [17, 18, 28, 15, 19, 2, 3, 3].

Recently, Altmann, Fitzi and Maurer [1] have extended our results for the Agreement problems to the generalized adversarial setting. (A general adversary is characterized by an adversary structure which is a set of subsets of the player set.)

In [12], Fitzi, Hirt and Maurer consider a hybrid failure model relevant to secure multi-party computation [5, 7], where processors can be actively or passively corrupted (i.e., Byzantine or gossiping, resp.)

The problem of clock synchronization has been widely studied under a model that makes the following assumptions (e.g., [10, 31]). It is assumed that processors communicate only by exchanging messages and that the maximum message delay is bounded. Hardware clocks are assumed to be imperfect and drift at a bounded rate. Thus, hardware clocks advance within a linear envelope of real time. If hardware clocks were used unaltered, then the difference between the fastest and slowest clocks would progressively increase. In order to keep the software clocks close to real time and to each other, a *clock-synchronization protocol* is run periodically. It has been shown by Dolev, Halpern and Strong [10] that in the Byzantine model, also $n > 3t$ is necessary for achieving clock synchronization. Clock synchronization under a hybrid failure model has also been studied by Rushby [28]

Our results. One contribution of this paper is, as in [33, 24], the introduction of a new family of models of failure in distributed computing. The new models permit some processors to behave in an unrestricted Byzantine manner but do not require the high processor redundancy of the pure Byzantine model. In contrast to the "classical" models, the new models admit (non-authenticated) solutions to the Agreement and Clock Synchronization problems for $n \leq 3t$ processors in the presence of even one Byzantine failure.

A second contribution is the introduction of several algorithmic techniques for coping with a mixture of crash and Byzantine failures. These simple techniques are particularly interesting since they are of limited value at best in either the pure crash or pure Byzantine failure model.

A further contribution is to establish the relationship necessary between the number of processors, total faults, and Byzantine faults if the above problems are to be solvable in the new model, namely, $n > t + 2 \cdot b$. Protocols that achieve these bounds are presented. We also give a lower bound on the time and network connectivity required to solve the Agreement problems—$t+1$ rounds and $t+b+1$, respectively. "Early-stopping" variants of the Agreement protocols are also presented. (Early stopping is a technique introduced by [11] as a means of circumventing the lower bound on running time.)

Our Agreement protocols are particularly interesting because they show how a solution to a weakened version of the Consensus problem (*Frangible* Consensus) can be used to create a solution to the stronger version of the Consensus and Byzantine Generals Problems.

Difficulties of the models. A trivial way of addressing the problem of mixed failures is to assume that *all* faulty processors may behave in a Byzantine manner, but some faulty processors "choose" to only exhibit benign faults. However, doing so rules out the possibility of low redundancy since the lower bound on Byzantine faults would apply. In contrast, we demonstrate a lower bound on mixed failures that admits the possibility of using fewer processors.

Another interesting aspect of the lower bounds that we establish is that it admits the possibility of solutions to these problems even if the *majority* of processors are faulty. Note that in the pure Byzantine model, non-faulty processors outnumber faulty processors by a factor of more than two. By demonstrating protocols meeting the lower bounds, we show this not to be an obstacle.

Organization of the paper. The rest of the paper is organized as follows. In Section 2 we introduce the new failure models and present the relevant definitions and assumptions for the Agreement and Clock Synchronization problems. In Section 3 we establish the necessary relationship among n, t, and b for the problems to have a solution. In Section 4 we present the new algorithmic techniques. The last two Sections, 5 and 6, present the protocols for Agreement and Clock Synchronization in the new failure models, respectively. We conclude with some final remarks.

2 Definitions

A more realistic family of failure models. We are given n processors numbered 0 through $n-1$ that may communicate only by exchanging messages. For the Agreement problems we assume the synchronous round-based model.

The Flexible Failure model is a family FF of models indexed by the triple (n, t, b) where $n > t \geq b$. In model $\mathrm{FF}(n, t, b)$, there are a total of n processors, any of which may fail either by crashing or by acting arbitrarily (Byzantine), but in any event the total number of failed processors is no more than t, and of these t processors, no more than b may be Byzantine. Model $\mathrm{FF}(n, t, 0)$ corresponds to the traditional crash failure model while $\mathrm{FF}(n, t, t)$ corresponds to the traditional Byzantine failure model. These relationships are illustrated in Figure 1.

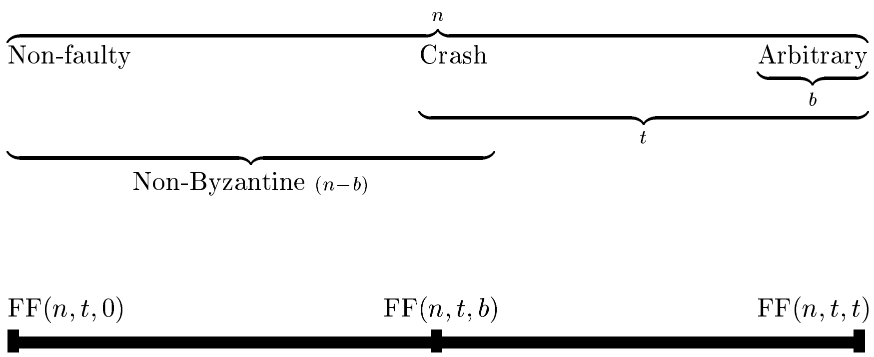

Figure 1: Flexible failure model

The Agreement problems. In the Consensus Problem, each processor begins with an initial binary value and the non-faulty processors are required to compute a final binary value subject to an *Agreement* condition:

- *Agreement*: The final values of non-faulty processors are equal.

and a *Validity* condition. The usual Validity Condition:

- *Strong Validity*: If the initial value of at least $n - t$ non-faulty processors is equal to v, then the final value is v.

gives rise to the "strong" Consensus Problem. That is, no matter what failures occur, if all non-faulty processors have the same initial value, then that value is the final value. The Weak Consensus Problem [20] uses the weaker condition:

- *Weak Validity*: If the initial value of at least n non-faulty processors (i.e., no failures occur) is equal to v, then the final value is v.

Weak Validity restricts the final value only in the case of failure-free executions. We define the *Frangible Consensus Problem* to use a Validity condition lying between the other two:

- *Frangible Validity*: If the initial value of at least $n - b$ non-faulty processors is equal to v, then the final value is v.

The motivation behind Frangible Validity is that if the non-Byzantine processors, including the ones that subsequently fail by crashing, begin with the same initial value, then that value must be the final value as well.

Model for clock synchronization. The problem of clock synchronization has been widely studied under a model that makes the following assumptions. It is assumed that processors communicate only by exchanging messages and that the maximum message delay is bounded by t_{del} units of real time. Hardware clocks are assumed to be imperfect and drift at a rate bounded by ρ. That is, after 1 unit of real time, a hardware clock can advance by as much as $(1 + \rho)$ units or as little as $\frac{1}{(1+\rho)}$ units. Thus, (non-faulty) hardware clocks advance within a linear envelope of real time. If hardware clocks were used unaltered, then the difference between the fastest and slowest clocks would progressively increase. The solution is to implement software clocks and keep them close together: Each processor p maintains a variable C_p that is periodically incremented and used in place of the hardware clock, whose value is denoted by R_p. For simplicity, C_p is assumed to be initially equal to 0 for all processors.

It is assumed that we are given a quantity, $D_{\max}$, that denotes the maximum difference to be achieved between the software clocks of two nonfaulty processors; this is the *agreement* condition for clock synchronization. Keeping the clocks of nonfaulty processors close together is not sufficient in itself. There should also be a relationship between the software clocks and real time. This is expressed by an *accuracy* condition, which roughly expresses that the software clocks be in a linear envelope of real time. It has been shown that this envelope cannot be smaller than the envelope for the hardware clocks [30].

In order to keep the software clocks close to real time and to each other, a *clock-synchronization protocol* is run periodically. We use P to denote how often the synchronization protocol is executed. Obviously, there is a relationship between $D_{\max}$, the time needed to resynchronize, and P: To achieve closer synchronization, smaller values of P and faster algorithms are required. It has been shown [10, 30] that, when optimal accuracy is needed, then the maximum difference between software clocks

$$D_{\max} \geq (1 + \rho)^{-1} \cdot t_{\mathrm{del}}/2$$

($t_{\mathrm{del}}/2$ applies when the clocks read the same value [10]; the drift factor is due to [30]).

It has been shown by Dolev, Halpern and Strong [10] that in the Byzantine model, where processors may behave arbitrarily and also collude, and the hardware clocks are not restricted by the drift assumption, also $n > 3t$ is necessary for achieving clock synchronization.

3 Lower Bounds

In this section we examine lower bounds in the new failure models. We establish a necessary relationship among n, t and b if the Agreement and Clock Synchronization problems are to be solvable. A (trivial) lower bound on the time to reach agreement is also stated.

Theorem 1 *Consensus is achievable in the model* $\mathrm{FF}(n, t, b)$ *only if* $n > t + 2 \cdot b$.

Proof To the contrary, suppose Consensus were achievable with $n \leq t + 2 \cdot b$. Then consider the following failure scenario: Of the t faulty processors, some number c are initially crashed (and therefore do not participate in the protocol) and the remaining $b = t - c$ faulty processors fail in a Byzantine manner. Any Consensus protocol for the participating processors must be a protocol that achieves consensus among $n' = n - c$ processors while being resilient to b Byzantine failures. By the results of Lamport, Shostak, and Pease [22] a Consensus protocol that tolerates this number of Byzantine failures is possible if and only if $n' > 3 \cdot b$. But this condition implies that $n > t + 2 \cdot b$, contradicting the assumption on b. ∎

The lower bound on time required to achieve Consensus in the new model follows from the lower bound in the crash model [16] and is stated without proof.

Theorem 2 *A protocol to achieve Consensus in the model* $\mathrm{FF}(n, t, b)$ *requires at least* $(t+1)$ *rounds of communication in some runs.*

Regarding network connectivity, the following lower bound is established.

Theorem 3 *Consensus is achievable in the model* $\mathrm{FF}(n, t, b)$ *only if the connectivity of the network exceeds* $(t + b)$.

Proof To the contrary, suppose Consensus were achievable with a network having connectivity $k \leq (t + b)$. Then consider the following failure scenario: Of the t faulty processors, some number c are initially crashed (and therefore do not participate in the protocol) and the remaining $b = t - c$ faulty processors fail in a Byzantine manner. Any Consensus protocol for the participating processors on the surviving network must be be resilient to b Byzantine failures. By the results of Dolev [8] a Consensus protocol that tolerates this number of Byzantine failures is possible if and only if the surviving network has connectivity exceeding $(2 \cdot b)$-connected. But this condition implies that the connectivity of the original network was more than $c + 2 \cdot b = t + b$. contradicting the assumption on b. ∎

Theorem 4 *Clock Synchronization is achievable in the model* $\mathrm{FF}(n, t, b)$ *only if* $n > t + 2 \cdot b$.

Proof To the contrary, suppose clock synchronization were achievable with $n \leq t + 2 \cdot b$. Then consider the following failure scenario: Of the t faulty processors, some number c are initially crashed (and therefore do not participate in the protocol) and the remaining $b = t - c$ faulty processors fail in a Byzantine manner. Any clock-synchronization protocol for the participating processors must be a protocol that achieves clock synchronization among $n' = n - c$ processors while being resilient to b Byzantine failures. By the results of Dolev, Halpern, and Strong [10] a clock synchronization protocol that tolerates this number of Byzantine failures is possible if and only if $n' > 3 \cdot b$. But this condition implies that $n > t + 2 \cdot b$, contradicting the assumption on b. ∎

4 New Algorithmic Techniques

Many "obvious" techniques fail to work as expected in the Flexible Failure models. For example, one might be tempted to take a Consensus protocol for the Byzantine model and convert it into a protocol for the Flexible Failure model by selectively replacing the quantity t by the quantity b. This can fail because it may totally ignore the crash failures. Even worse, it may result in the crashed processors exhibiting near-Byzantine failure. This is because many Byzantine protocols implicitly use a "default" value in place of the value not sent by a silent processor. For example, suppose that 0 were the default value and processor p crashed after having sent 1 to a single non-faulty processor. Then all other non-faulty processors interpret p's silence as being equivalent to having sent 0, thereby causing p to appear Byzantine.

Even when the "obvious" techniques work, they often succeed only for a sub-optimal number of processors. For example, in Section 6 we first formulate a time-efficient protocol for clock synchronization this way but only for $n > 2t + b$ processors.

4.1 Veto

The "veto" is the first new technique that we introduce. A "veto" occurs when a minority of processors expresses a preference for some value other than the one indicated by the majority of processors. For example, in order for a non-faulty processor p to accept value v from processor q, we can require not only that there be a sufficient number T (usually equal to $n - t$) of witnesses to q having sent v but *also* no more than b witnesses to any other value. Thus, a quorum of at least $b + 1$ processors can "veto" v. This is in contrast to several existing protocols, e.g., [25, 31, 29], that only require a minimum number of positive witnesses.

In order to prevent Byzantine processors from vetoing the actions of non-faulty processors, it is necessary that a veto quorum include at least one non-Byzantine processor. But, if we tried to implement the veto in the pure Byzantine model with the most common value of T, i.e., $n - t$, we would discover that the existence of T positive witnesses necessarily precludes the existence of a veto quorum. Thus, the veto is redundant in the pure models. Only in the Flexible Failure models, e.g., $\mathrm{FF}(n, t, b)$ where $t > b > 0$, is it possible to simultaneously have both $n - t$ positive witnesses and a veto quorum of size at least $b + 1$.

Use of the veto technique is illustrated in the `MakeUnique` protocol of Figure 2 which implements a form of Crusader Agreement [8]. Each processor simultaneously executes `MakeUnique`, supplying its current value of the binary-valued variable V. `MakeUnique` utilizes the veto technique to ensure that all processors that accept a value accept the same value, but some processors may not accept any value (indicated by "accepting" the value 2). In the sequel, the value of a variable local to processor i will be denoted by a subscript, e.g., V_i denotes the value of processor i's V variable.

Lemma 1 *If p, q are non-faulty processors such that, in* `MakeUnique`*, p assigns $v \neq 2$ to V_p and q assigns $w \neq 2$ to V_q, then $v = w$.*

Proof Since of the at least $n - t$ non-faulty processors, there is some value that is sent in `MakeUnique` by the majority of them and $\lceil (n-t)/2 \rceil > b$. Thus, if the majority sent v, no non-faulty processor could set V to $\bar{v}$ at the end of `MakeUnique`. ∎

```
MakeUnique(V):
        send(V) to all processors;
        C[0] := number of 0's received;
        C[1] := number of 1's received;
        if     C[0] ≥ n − t ∧ C[1] ≤ b then V := 0
        elseif C[1] ≥ n − t ∧ C[0] ≤ b then V := 1
        else   V := 2
        fi;
```

Figure 2: `MakeUnique` Protocol for processor $0 \leq i < n$.

4.2 Double-echo

Many exisiting protocols [25, 31] utilize a "broadcast-accept" sub-protocol for "broadcasting" a value. These sub-protocols involve having a "broadcaster" first send its value to all processors who in turn act as witnesses to this fact by "echoing" the value to all processors. Only if a processor receives a certain number of echoes does it "accept" a value from the broadcaster. This number serves to limit the abilities of a faulty broadcaster.

We introduce a variation of "broadcast-accept" protocols in which each processor receiving a value from the "broadcaster" *immediately* sends two consecutive identical echoes. Although the second echo does not appear at first to convey any information, it does in fact serve an important purpose: receipt of the second echo from a non-Byzantine processor q proves that q did not crash before sending the first echo to all processors. We demonstrate the value of this information through the `Unicast` protocol of Figure 3 and its proof.

`Unicast` is similar in purpose to `MakeUnique` in that each processor executes `MakeUnique`, supplying its current value of variable V. At the end of `Unicast` each processor "accepts" some value from every other processor, and the array D is set to indicate how many of each value is accepted, e.g., $D[v]$ is the number of v's accepted. It is stronger than `MakeUnique` in that it also guarantees that if non-faulty processor p accepts value $v \neq 2$ from processor j, any other non-faulty processor q either also accepts v from j or accepts 2 from j. Thus, a Byzantine processor is prevented from having 0 accepted by one non-faulty processor while 1 is accepted by some other non-faulty processor. This property will be vital in allowing us to implement "early stopping" in a subsequent section.

The power of the double-echo technique is made clear by the following lemma.

Lemma 2 *If p, q are non-faulty processors such that, in* `Unicast`*, p* `accept`*'s $\langle v, j\rangle, v \neq 2$ and q* `accept`*'s $\langle w, j\rangle, w \neq 2$, then $v = w$. Moreover, a value sent by a non-faulty processor is accepted by all non-Bzyantine processors.*

Proof For the sake of contradiction, suppose p accepted $\langle v, j\rangle, v \neq 2$ and q accepted $\langle w, j\rangle, v \neq 2$ such that $w \neq v$. Then there were at least $n - t$ processors that sent $\langle v, j, 1\rangle$ and $\langle v, j, 2\rangle$ to p. But q must have received $n - t - b > b$ of these $\langle v, j, 1\rangle$'s since there are no more than b Byzantine failures. Thus, q could not accept w unless $v = w$.

That the value v sent by a non-faulty processor j is accepted can be seen by simple inspection of the code. There are guaranteed to be at least $n - t$ double echoes of $\langle v, j\rangle$ since all non-faulty

```
Unicast(V, D):
    send(V) to all processors ;

    for j := 0 to n − 1 do
        if receive w from processor j then
            send(⟨w, j, 1⟩) to all processors ;
            send(⟨w, j, 2⟩) to all processors
        fi;

        for l := 0 to 2 do
            E[l, j] := number of ⟨l, j, 1⟩'s received;
            F[l, j] := number of processors from which both ⟨l, j, 1⟩ and ⟨l, j, 2⟩ was received;
        od;

        if     F[0, j] ≥ n − t ∧ E[1, j] ≤ b ∧ E[2, j] ≤ b then accept(⟨0, j⟩)
        elseif F[1, j] ≥ n − t ∧ E[0, j] ≤ b ∧ E[2, j] ≤ b then accept(⟨1, j⟩)
        elseif F[2, j] ≥ n − t ∧ E[0, j] ≤ b ∧ E[1, j] ≤ b then accept(⟨2, j⟩)
        fi;
    od;

    D[0] := number of 0's accepted;
    D[1] := number of 1's accepted;
    D[2] := number of 2's accepted;
```

Figure 3: `Unicast` Protocol for processor $0 \leq i < n$.

```
V := "initial value";
for K := 1 to t+1 do
    /* Universal Exchange 1 */
        MakeUnique(V) ;

    /* Universal Exchange 2 */
        Unicast(V, D);

        if     D[0] > b then V := 0
        elseif D[1] > b then V := 1
        fi;

    /* King's Broadcast */
        if i = K then send(V) to all processors  fi;
        W := value received from processor K;
        if     (V = 2 ∨ D[V] ≤ b ∨ D[2] > b) then V := min(1, W) ;
        elseif (V ≠ 2 ∧ D[V] ≥ n − b ∧ D[2] ≤ b) then DecidedAction;
        fi
od;
"final value" := V
```

Figure 4: (Early-stopping) Frangible Consensus Protocol for processor $0 \leq i < n$.

processors receive v from j and the only echoes with contrary values can come from the at most b Byzantine processors. ∎

In the next sections, we illustrate the use of the new algorithmic technqiues via Agreement and Clock Synchronization protocols.

5 Protocols for the Agreement Problems

5.1 A protocol for Frangible Consensus

A protocol that solves the Frangible Consensus problem for $\mathrm{FF}(n,t,b)$ where $n > t + 2 \cdot b$ is shown in Figure 4. It follows the Phase King paradigm of [6] in which the computation proceeds in phases, each of which has a processor designated as "phase king." Each phase K consists of 3 rounds of communication. During the first two rounds of a phase, all processors communicate with one another; in the final round only processor K (the king) sends messages.

In the final statement of the figure, there is the notation `DecidedAction`. This notation is merely a place-hold for either of two statements: `skip` or `halt`.

We now argue the correctness of the protocol. In the sequel, assume that the `DecidedAction` is `skip`. As a preliminary step, we show that if all non-Byzantine processors begin a phase with the same value of V, then all non-faulty processors end the phase with V unchanged.

Lemma 3 (Persistence) *If there is a v such that for each non-Byzantine processor p, $V_p = v$ at the beginning of a phase, then at the end of the phase $V_q = v$ for each non-faulty processor q.*

Proof During Universal Exchange 1, each of the at least $n-t$ non-faulty processors send v and only the Byzantine processors send a value other than v. By inspection of the code of `MakeUnique`, at the end of Universal Exchange 1, $V=v$ for each non-faulty processor. Similarly by inspection of the code of `Unicast`, at the end of Universal Exchange 2, $D[\bar{v}] \leq b$, $D[2] \leq b$ and $D[v] \geq n-t > b$ for each non-faulty processor. Thus $V_q = v$ for each non-faulty processor q at the end of Universal Exchange 2 and all non-faulty processors ignore the Phase King's broadcast. ∎

We now show that the *Agreement* property is established.

Lemma 4 (Agreement) *At the end of phase $t+1$ for any non-faulty processors p and q, $V_p = V_q$.*

Proof Let $g \leq t+1$ be the non-faulty processor with the lowest number. At the end of phase g, there are two possibilities:

1. All non-faulty processors replace V with the value received from processor g. By assumption g is non-faulty so all non-faulty processors assign the same value to V.

2. For some non-faulty processor p, at the end of Universal Exchange 2 of phase g, $V_p \neq 2, D[V_p]_p > b, D[2]_p \leq b$ and thus p ignores the King's Broadcast.

   First observe that, by Lemma 1, at the end of Universal Exchange 1 if processors p and q are non-faulty and $V_p = v \neq 2$ and $V_q = v' \neq 2$ then $v = v'$. So in Universal Exchange 2, any two non-faulty processors that send values less than 2 must send the same value. Because $D[V_p]_p > b$, this value must be equal to V_p.

   In Universal Exchange 2, the majority of non-faulty processors either sends V_p or 2. But by assumption $D[2]_p \leq b$ so the majority of non-faulty processors did not send 2 in Universal Exchange 2. Therefore every non-faulty processor q has $D[V_p]_q \geq \lceil (n-t)/2 \rceil > b$ and $V_q = V_p$ at the end of Universal Exchange 2. In particular, processor g, the King of phase g has $V_g = V_p$ at the end of Universal Exchange 2.

   Therefore, any non-faulty processor that does not ignore the King's Broadcast of phase g ends the phase with the same value of V as any non-faulty processor that does ignore the phase's King.

By Lemma 3, no non-faulty processor changes its value of V subsequent to phase g. ∎

Note that the above proof does not use any of the special properties of the `Unicast` protocol and would, in fact, remain true even if `Unicast` was replaced by "`send`(V) to all processors" and where every message received was accepted. Doing so would save 2 rounds per phase. We have chosen to express the algorithm in the manner we have so that "early stopping" may be easily added, as discussed in a subsequent section.

Theorem 5 (Correctness) *The protocol of Figure 4 solves the Frangible Consensus problem for $n > t + 2 \cdot b$.*

Proof The Agreement property follows from Lemma 4 and Frangible Validity follows from Lemma 3. ∎

5.2 Protocols for the Strong Agreement problems

We now demonstrate how a protocol for Frangible Consensus can be used to derive protocols for the Agreement problems with the *Strong Validity* condition. This shows that the "strong" variants of the Agreement problems are solvable in our new models.

In the Byzantine Generals problem [22], a distinguished processor (called the General) needs to broadcast a value subject to *Agreement* and *Strong Validity* conditions similar to those described for the Consensus problem in Section 2. That is, all non-faulty processors must agree on the value broadcast by the General and, if the General is non-faulty, the agreed-upon value is the one chosen by the General.

Given a protocol that solves the Frangible Consensus Problem in $\mathrm{FF}(n, t, b)$, a protocol that solves the Byzantine Generals problem in $\mathrm{FF}(n, t, b)$ is obtained as follows:

1. In round 1, the General sends the value it wishes to broadcast to all processors including itself.

2. The processors subsequently execute the Frangible Consensus Protocol using the value received from the General in round 1 (or 0 if no value was received) as the initial value.

If the General is non-faulty, all non-Byzantine processors execute the Frangible Consensus protocol with the same initial value and, by the correctness of the Frangible Consensus protocol, that is the value computed as the final value. Thus, the above protocol satisfies both the Agreement and Strong Validity conditions.

Given a protocol that solves the Byzantine Generals problem in $\mathrm{FF}(n, t, b)$,, a protocol that solves the Consensus problem with the *Strong Validity* condition in $\mathrm{FF}(n, t, b)$ is obtained as follows:

1. Each processor acts as the General in a separate instance of the Byzantine Generals Protocol. Thus, each processor obtains an identical vector of n values.

2. The final value is the majority value in the vector.

Thus, the existence of a protocol for Frangible Consensus in the new model implies that the Agreement problems with the *Strong Validity* condition are solvable in the new model that is introduced in this paper.

5.3 Protocols with reduced running time

When `DecidedAction` is a `skip` statement, the protocol of Figure 4 always takes exactly $(t+1)$ phases to complete even if no processors fail during its execution. Protocols that terminate in fewer phases when the number of processors that fail during its execution is less than t are called "early-stopping." By having an early-stopping protocol that runs in time proportional to the number of failures that actually occur during its execution, we pay only for the degree of fault tolerance actually needed. Figure 4 can be turned into an early-stopping protocol for the Frangible Consensus problem by defining the `DecidedAction` action to be `halt`.

We now turn to the correctness of the early-stopping protocol for Frangible Consensus. Because some non-faulty processors halt before others, we adopt the convention that if processor p receives no value from processor q in any round, p assumes that q sent a value equal to V_p at the start of the phase.

The correctness of the early-stopping algorithm is now argued. The proof is similar to that of the non-early-stopping protocol, so we just highlight the key points.

Theorem 6 *The protocol of Figure 4 solves the Frangible Consensus problem for $n > t + 2 \cdot b$ and always terminates by phase $t+1$. Moreover, let f be the number of processors that actually fail. If $f \leq b$ then the protocol terminates in at most $f + 2$ phases.*

Proof The following facts are easy to establish:

1. If $n-b$ non-faulty processors begin a phase with the same value of V, they all halt at the end of the phase with V unchanged.

   The proof is similar to that of Lemma 3.

2. If the King of phase K is non-faulty all non-faulty processors will end phase K with the same value of V.

   The proof is similar to that of Lemma 4. By fact 1, all non-faulty processors will halt by phase $K+1$ provided that no more than b processors fail.

3. If the King of phase K is faulty and the first non-faulty processor p halts in phase K, then all non-faulty processors end phase K with the same value of V.

   By inspection of the code's stopping condition, p halts only if at the end of phase K, $V_p \neq 2, D[V_p]_p \geq n-b$ and $D[2]_p \leq b$. Since at most t processors are faulty, this implies that for all non-faulty q, at the end of phase K, $D[V_p]_q \geq n-b-t > b$. Moreover, since the values are sent by `Unicast`, by Lemma 2, $D[2]_q \leq b$. By inspection of the code, this implies that all non-faulty q ignore the King's Broadcast of phase K and have $V_q = V_p$ at the end of phase K.

   By fact 1, all non-faulty processors will halt by phase $K+1$ provided that no more than b processors fail.

Taken together, the above facts imply that if no more than $f \leq b$ failures occur, all non-faulty processors reach agreement and halt by phase $f+2$. ∎

6 Clock Synchronization Protocols

In most protocols for synchronizing clocks in the traditional failure models, a processor sends a "tick" message to all other processors once its hardware clock has advanced a certain amount (denoted P). Only after a processor has received "tick" messages from a sufficient number of processors does it increment its software clock. This serves to keep the software clocks of the processors closely synchronized. If the software clocks of nonfaulty processors are required to be more closely synchronized (indicated by a smaller $D_{\max}$), then P is decreased; that is, software resynchronization is performed more frequently.

Our protocols for synchronizing clocks in model $\mathrm{FF}(n,t,b)$ are based on the protocol of Srikanth and Toueg [30], which follows the above paradigm. The general structure of the protocols is shown in Figure 5; we obtain the two protocols in the sequel by instantiating different `Broadcast` primitives.

6.1 A simple but inefficient protocol

We obtain the first protocol for synchronizing clocks in model $\mathrm{FF}(n,t,b)$ by replacing `Broadcast` in Figure 5 by the `EchoOnce` primitive of Figure 6. However, the protocol requires $n > 2 \cdot t + b$ processors rather than the optimal $n > t + 2 \cdot b$ processors. We subsequently will modify this protocol to achieve processor optimality

The intuition behind this protocol is as follows. When a processor's hardware clock indicates that P units of time have elapsed (we use ΔR_p to keep track of this), a processor sends a `ready` message to all other processors in order to inform them of this fact. Once a processor has received more than b such messages, it sends an `echo` message to all other processors. Note that since there are no more than b processors that may fail arbitrarily, an `echo` message indicates that the hardware clock of at least one non-Byzantine processor (whose clock therefore operates within the

```
/* Assume clocks are initially all equal */
ΔR_p = 0; /* keeps track of R_p */
for K := 1 to ∞ codo
    if  ΔR_p = P then  /* ready to update C_p */
          Broadcast(round K);
    //
    if  accepted round K messages then  /* according to the primitive */
          C_p := C_p + P + α;
          ΔR_p = 0
cood;
```

Figure 5: A generic clock synchronization protocol; code for processor p

true hardware drift rate) has advanced P units. At this point, a correct processor with a slow software clock (one that has not yet advanced P units) should recognize that it is slow and advance its software clock. Unfortunately, there is no guarantee that *all* correct processors receive more than b `ready` messages at the same instant as the first processor, because b of these messages could have been generated by faulty processors that may not send messages to any other processor. Therefore, an additional step is necessary to ensure that all processors receive a sufficient number of messages within a bounded time of the first processor's receipt of this number of messages. This is what is achieved by the last two steps of `EchoOnce`.

We also use the same technique as Srikanth and Toueg [30] to ensure that software clocks are monotonically increasing, by increasing the clocks by an additional factor α. This quantity is calculated to take into account how much the hardware clock could have advanced in the interval between when a processor sends its `ready` message and when it actually advances its software clock. We refer the reader to [30] for details.

We now proceed to prove the bound on the number of processors required by this protocol. We focus here on the number of processors that are required, but remark that a complete proof should contain *progress*, that is, the fact that at least one correct processor advances its clock, as well as the inductive proof that the synchronization bounds are maintained from one iteration of the protocol to the next. The treatment of these properties is identical to that of [30], and is omitted for brevity. We let p denote the first processor to advance its clock in a given iteration of the protocol.

Theorem 7 *Using protocol* `EchoOnce` *of Figure 6 in the generic protocol of Figure 5 yields a clock synchronization protocol for model* $\mathrm{FF}(n,t,b)$*, provided that* $n > 2 \cdot t + b$.

Proof In order to advance its clock C_p, processor p must receive `echo` messages from a set of at least $n-t > t+b$ processors (see Figure 6). This set contains more than b nonfaulty processors because there are, by definition, no more than t processors that can fail. Each of these nonfaulty processors sends an `echo` to *all* of the at least $n-t$ nonfaulty processors. Thus, within t_{del} time units of p advancing its clock, at least $n-t$ nonfaulty processors will have received more than b `echo` messages and will, by the Once $> b$ test, have sent `echo` messages of their own. Within an additional t_{del} time units, all nonfaulty processors will have Once $\geq n-t$ and will therefore advance their clocks (last step of `EchoOnce`). ∎

```
EchoOnce(round k);
    send(ready,k) to all processors;

    /* Let Once denote the number of processors from which
    at least one (echo,k) message has been received. */
    if   received (ready,k) messages from > b processors then
            send(echo,k) to all processors;
    []   Once > b and p has not echoed then
            send(echo,k) to all processors;
    fi

    if Once ≥ n − t then
            accept(round k);
```

Figure 6: Broadcast primitive for simple but inefficient clock synchronization protocol; code for processor p

Note that, by the above proof, all nonfaulty processors will advance their software clocks within two message delays of the first processor that advances its clock. This effectively constrains how small $D_{\max}$ may be.

6.2 A processor-optimal protocol for clock synchronization

`EchoOnce` is a "broadcast-accept" protocol where the echo is withheld until a processor hears from more than b "broadcasters" (processors sending a `ready` message), and "accept" corresponds to advancing the clock. We now apply the dobule-echo technique of Section 4.2. As before, receipt of the second echo from a non-Byzantine processor q proves that q did not crash before sending the first echo to all processors. This information is critical in allowing to derive a clock-synchronization protocol optimal in the number of processors; that is, only $n > t + 2 \cdot b$ processors are needed. The broadcast-accept subprotocol, `EchoTwice`, is shown in Figure 7.

Theorem 8 *Using protocol* `EchoTwice` *of Figure 7 in the generic protocol of Figure 5 yields a clock synchronization protocol for model* $\mathrm{FF}(n, t, b)$*, provided that* $n > t + 2 \cdot b$.

Proof In order to advance its clock C_p, processor p must receive double-`echo` messages from a set of at least $n - t > 2 \cdot b$ processors (see Figure 7). This set, by definition, contains at most b processors that fail in an arbitrary manner and therefore more than b *non-Byzantine* processors, that is, processors that are either nonfaulty or fail only by crashing. But each non-Byzantine processors that has sent two echoes to p must have succeeded in sending at least one echo to every one of the at least $n - t$ nonfaulty processors. Thus, within t_{del} time units of p advancing its clock, at least $n - t$ nonfaulty processors will have received more than b `echo` messages at least once and will, by the Once $> b$ step, have sent double-`echo` messages of their own. Within an additional $2 \cdot t_{\mathrm{del}}$ time units, all nonfaulty processors will have Twice $\geq n - t$ and will therefore advance their clocks by the last step of `EchoTwice`. ∎

Note that, by the above proof, all nonfaulty processors will advance their software clocks within three message delays of the first processor that advances its clock. Thus, we achieve processor

```
EchoTwice(round k);
      send(ready,k) to all processors;

      /* Let Once denote the number of processors from which
      at least one (echo,k) message has been received. */
      /* Let Twice denote the number of processors from which
      two (echo,k) messages has been received. */
      if   received (ready,k) messages from > b processors then
              send(echo,k) to all processors;
              send(echo,k) to all processors;
      □    Once > b and p has not double-echoed then
              send(echo,k) to all processors;
              send(echo,k) to all processors;
      fi

      if Twice ≥ n − t then
              accept(round k);
```

Figure 7: Broadcast primitive for processor-optimal clock synchronization protocol; code for processor p

optimality at the cost of requiring a larger $D_{\max}$.

7 Discussion

We have presented a new model of failure for distributed computing and demonstrated protocols that solve the Agreement and Clock Synchronization Problems in that model. The attractiveness of the model is that it allows some processors to fail in an unrestricted manner while not incurring the high redundancy required in a model of pure Byzantine failures. Since experience shows that failures worse than processor crashes do occur, the new model is closer to reality than either the pure crash or pure Byzantine models.

The new failure models introduced in this paper necessitate types of reasoning different from that commonly employed in either the pure crash or pure Byzantine models of failure. For example, the technique of forcing a processor to repeat an action in the presence of witnesses contributes little in either of the pure models but proves useful in the new model, as illustrated by the `Unicast` protocol of Figure 3 and its proof.

This paper focussed on introducing a new model and showing that important problems could be solved within it. Thus, our protocols emphasized simplicity over efficiency. Our protocol without early-stopping requires $3(t+1)$ rounds (when `Unicast` is replaced by a simple `send`(V) to all processors) and single-bit messages; we also presented a simple extension that was able to stop early under certain conditions. The issues of efficiency and early-stopping have recently being discussed in greater generality in [1].

References

[1] B. Altmann, M. Fitzi, and U. Maurer. Byzantine agreement secure against general adversaries in a dual failure model, personal communication. In *Thirteenth International Symposium on Distributed Computing—DISC '99 (P. Jayanti ed.)*, pages 123–137. Springer-Verlag, 1999.

[2] M.H. Azadmanesh and A.W. Krings. Analysis of a realistic fault model. *Journal of Systems Science*, 22(3):29–51, 1997.

[3] M.H. Azadmanesh and A.W. Krings. Egocentric voting algorithms. *IEEE Transactions on Reliability*, 46(4):494–502, 1997.

[4] R. Bazzi and G. Neiger. Optimally simulating crash failures. In *Proceedings of the Fifth International Workshop on Distributed Algorithms*. Springer-Verlag, 1991.

[5] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In STOC'88 [32], pages 1–10.

[6] P. Berman and J.A. Garay. Asymptotically optimal distributed consensus. In *Proceedings of the Sixteenth Annual International Conference on Automata, Languages, and Programming, LNCS Vol. 372*, pages 80–94. Springer-Verlag, 1989.

[7] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In STOC'88 [32], pages 11–19.

[8] D. Dolev. The byzantine generals strike again. *Journal of Algorithms*, 3(1):14–30, 1982.

[9] D. Dolev, M. Fischer, R. Fowler, N. Lynch, and H. Strong. An efficient algorithm for byzantine agreement without authentication. *Information and Control*, (52):257–274, 1982.

[10] D. Dolev, J. Halpern, and H.R. Strong. On the possibility and impossibility of achieving clock synchronization. In *Proc. Sixteenth Annual ACM Symp. on the Theory of Computing*, pages 504–511. ACM, 1984.

[11] D. Dolev, R. Reischuk, and H.R. Strong. Early stopping in byzantine agreement. *Journal of the ACM*, 37(4):720–741, 1990.

[12] M. Fitzi, M. Hirt, and U. Maurer. Trading correctness for privacy in unconditional multi-party computation. In *Advances in Cryptology – CRYPTO '98*, volume 1462, pages 122–136. Springer-Verlag Lecture Notes in Computer Science, 1999.

[13] J.A. Garay and K.J. Perry. A continuum of failure models for distributed computing. In *Proceedings of the Sixth International Workshop on Distributed Algoriths (WDAG-6)*, volume 647, pages 153–165. Springer-Verlag Lecture Notes in Computer Science, 1992.

[14] J.A. Garay and K.J. Perry. Synchronizing clocks in a system resilient to a wide class of failres. In *Proceedings 1994 IEEE Workshop on Fault-Tolerant Parallel and Distributed Systems (eds. D. Pradhan and D. Avresky)*, pages 260–267. Computer Society Press, 1995.

[15] L. Gong, P. Lincoln, and J. Rusby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Dependable Computing for Critical Applications–5 (eds.R. Iyer, M. Morganti, K. Fuchs and V. Gligor)*, pages 139–157. Dependable Computing and Fault-Tolerant Systems series, IEEE Computer Society Press, 1995.

[16] V. Hadzilacos. Issues of fault tolerance in concurrent computations. *Ph.D. Dissertation, Harvard University*, 1984.

[17] R.M. Kieckhafer and M.H. Azadmanesh. Low cost approximate agreement in partially connected networks. *Journal of Computing and Information*, 3(1):53–85, 1993.

[18] R.M. Kieckhafer and M.H. Azadmanesh. Reaching approximate agreement with mixed mode faults. *IEEE Transactions on Parallel and Distributed Systems*, 5(1):53–63, 1994.

[19] R.M. Kieckhafer and M.H. Azadmanesh. A unified approach to synchronous and asynchornous approximate agreement in the presence of hybrid faulrs. *IEEE Transactions on Reliability*, 44(4):622–631, 1995.

[20] L. Lamport. The weak byzantine generals problem. *Journal of the ACM*, 30(3):668–676, 1983.

[21] L. Lamport and M. Fischer. Byzantine generals and transaction commit protocols. Technical Report Opus 62, SRI, 1982.

[22] L. Lamport, R.E. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, 1982.

[23] P. Lincoln and J. Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Proceedings of the Twenty-Third Annual International Symposium on Fault-Tolerant Computing*, pages 402–411. IEEE, 1993.

[24] F. Meyer and D. Pradhan. Consensus with dual failure models. *IEEE Transactions on Parallel and Distributed Systems*, 2(2):214–222, 1991.

[25] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[26] M. Pease, R.E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[27] K.J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, 1986.

[28] J. Rushby. A formally verified algorithm for clock synchronization under a hybrid fault model. In *Proceedings of the Thirteenth Symposium on Principles of Distributed Systems*, pages 304–313. ACM, 1994.

[29] K.J. Perry S. Toueg and T.K. Srikanth. Fast distributed agreement. *SIAM Journal of Computing*, 16(3):445–457, 1987.

[30] T.K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[31] T.K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[32] *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, Chicago, Illinois, 2–4 May 1988.

[33] P. Thambidurai and Y. Park. Interactive consistency with multiple failure models. In *Proceedings 7th Symp. on Reliable Distributed Systems*, pages 93–100. IEEE Computer Society Press, 1988.