A Compiler that Increases the Fault Tolerance of Asynchronous Protocols

BRIAN A. COAN

Abstract—**We give a compiler that increases the fault tolerance of certain asynchronous protocols. Specifically, it transforms a "source protocol" that is resilient to crash faults into an "object protocol" that is resilient to Byzantine faults. Our compiler can simplify the design of protocols for the Byzantine fault model because it enables us to break the design process into two steps. The first step is to design a protocol for the crash fault model. The second step, which is completely mechanical, is to compile the protocol into one for the Byzantine fault model. We use our compiler to produce a new asynchronous approximate agreement protocol that operates in the Byzantine fault model. Specifically, we design a new asynchronous approximate agreement protocol for the crash fault model and observe that this protocol can be compiled into a protocol for the Byzantine fault model. In the Byzantine fault model, the new protocol improves in several respects on the performance of the asynchronous approximate agreement protocol of Dolev, Lynch, Pinter, Stark, and Weihl.**

Index Terms—**Approximate agreement protocols, asynchronous distributed systems, Byzantine faults, crash faults, fault tolerance, protocol transformations.**

I. Introduction

We GIVE a compiler that transforms an arbitrary standard-form asynchronous protocol that tolerates crash faults into an asynchronous protocol that tolerates Byzantine faults and that solves the same problem as the original protocol. Our compiler incorporates communication primitives and a message validation scheme developed by Bracha [1]. Bracha argues informally that his tools restrict the disruptive behavior of a processor that fails with a Byzantine fault. He argues that the restricted behavior is similar to that of a processor subject only to crash failures. He then uses these primitives to construct an interesting new randomized protocol for the Byzantine fault model.

Our goal is similar to Bracha's. It is to simplify the design and proof of asynchronous protocols that are resilient to Byzantine faults. Specifically, our approach is as follows. We incorporate Bracha's communication primitives and message validation scheme in a compiler, which we prove correct. Then, we design and prove protocols correct in the crash fault model. It follows from the correctness of the compiler that the protocols that we design for the crash fault model can be compiled into protocols that operate correctly in the Byzantine fault model.

A limitation of our compiler is that it only works for deterministic protocols. It is an open question to construct and prove correct a compiler for randomized protocols. Because our compiler works only for deterministic protocols, it is not useful in the particular application that Bracha considers.

There seem to be two principal benefits of our approach. First, it is simpler to design and prove protocols in the crash model than it is to do the same in the Byzantine model. Using our method, only the compiler needs to be proved correct for the Byzantine model. Second, our approach is modular. For example, we give two versions of our compiler with slightly different performance tradeoffs. (The two versions of the compiler use slightly different communication primitives.) After we prove a protocol correct in the crash fault model, we can use either version of the compiler to transform it into a protocol that is correct in the Byzantine fault model.

It should be clear that our compiler must change some of the properties of a protocol (like the kind of faults tolerated) and leave other properties unchanged (like the problem solved by the protocol). We use *correctness predicates* to formalize one of the properties that we would like our compiler to preserve. A correctness predicate is any predicate defined on the inputs to and answers of correct processors. We show that our compiler preserves the satisfaction of correctness predicates and the property that all correct processors eventually decide. Thus, our compiler preserves the solution to any problem that can be formalized by a correctness predicate and a requirement that all correct processors eventually decide. Agreement and approximate agreement are such problems.

Asynchronous systems are "harder" than synchronous ones because they can experience a superset of the executions of synchronous systems. If a protocol solves some problem in an asynchronous system, then it follows that the protocol solves the same problem in a synchronous system. Of course, this holds for any protocol that is the output of our compiler. The foregoing argument might lead one to think that our compiler is capable of transforming a synchronous protocol that solves some problem in the crash model into a synchronous protocol that solves the same problem in the Byzantine model. This is wrong. The difficulty is that our compiler relies on the fact that its source protocol operates correctly in asynchronous executions with crash faults. The property of asynchronous executions that our compiler relies on is that some messages between correct processors may be delivered very late.

Manuscript received March 5, 1988; revised July 25, 1988. This work was supported by the Advanced Research Projects Agency of the Department of Defense under Contract N00014-83-K-0125, the National Science Foundation under Grant DCR-83-02391, the Office of Army Research under Contract DAAG 29-84-K-0058, and the Office of Naval Research under Contract N00014-85-K-0168. An earlier version of this paper appears as chapter 4 of the author's Ph.D. thesis [3].

The author is with Bell Communications Research, Morristown, NJ 07960.

IEEE Log Number 8824079.

A limitation of our technique is that we are unable to force a faulty processor to accurately report its input value. We have accommodated this limitation by requiring that correctness predicates not depend on the input to a faulty processor. A related limitation accounts for our inability to generalize our compiler to randomized protocols. The specific problem is that there seems to be no way of detecting a processor that is behaving correctly in all respects except that it "cheats" when it makes random choices.

In general, the running time of a protocol may depend on the input. Informally, for some fixed input we say that a protocol has running time r if all correct processors decide by time r in any execution in which the message-delivery time is bounded above by one. Our compiler imposes certain overhead in the running time. There are two versions of the compiler. The first version increases the time by a factor of two and requires that the number of processors be more than four times the number of faults tolerated. The second version increases the time by a factor of three and requires that the number of processors be more than three times the number of faults tolerated. The first version of the compiler uses new communication primitives; the second version incorporates the communication primitives developed by Bracha. Both versions of the compiler substantially increase the number of message bits sent.

Our compiler requires that its "source protocols" be in a particular standard form. We believe that many asynchronous protocols can be put into this standard form. A notable exception is that the protocols that are output by our compiler appear incapable of being put into this standard form. It would be desirable to extend our compiler so that it could compile arbitrary source protocols. We believe that this can be done, but the unrestricted compiler and its proof seem very complicated. We leave the construction and proof of the unrestricted compiler as an open problem. We also leave open the problem of ensuring that all correct processors eventually stop sending messages (i.e., terminate).

Unfortunately, there are a limited number of known protocols that are potential source protocols for our compiler. The well-known impossibility result of Fischer, Lynch, and Paterson [8] shows that many problems have no deterministic solution in an asynchronous system. The only problems currently defined in the literature that can be solved with deterministic protocols in asynchronous systems are the approximate agreement problem [5], the inexact agreement problem [9], and the task assignment problem [2]. Despite the limited number of potential source protocols, we believe that our compiler is interesting because of the method that it embodies (i.e., modularizing the verification of fault-tolerant distributed protocols). In the synchronous case, where more problems have solutions, an analogous compilation technique has been developed by Neiger and Toueg [10].

Using the two versions of our compiler, we produce a pair of new asynchronous approximate agreement protocols that are resilient to Byzantine faults. Our protocols improve in several respects on the asynchronous approximate agreement protocol of Dolev, Lynch, Pinter, Stark, and Weihl [5]. Our method is to design a new asynchronous approximate agreement protocol for the crash fault model and observe that this protocol can be compiled into a protocol for the Byzantine fault model (using either version of our compiler). The protocol that we design in the crash fault model uses many ideas developed by Dolev *et al.* for use in their protocol. (The approximate agreement problem is defined in Section II.)

In the Byzantine fault model, our new approximate agreement protocols tolerate a larger proportion of faulty processors than the protocol of Dolev *et al.* Their protocol requires that the number of processors be more than five times the number of faults tolerated. One version of our protocol requires that the number of processors be more than four times the number of faults tolerated; the other requires that the number of processors be more than three times the number of faults tolerated. The second version of our protocol has an optimal amount of redundancy. This follows because Fischer, Lynch, and Merritt [7] have shown that no protocol solves the approximate agreement problem unless the number of processors is more than three times the number of faults tolerated.

Dolev *et al.* [5] propose using the convergence rate as a measure of the quality of an approximate agreement protocol. Intuitively, the *convergence rate* of an approximate agreement protocol is the factor by which the range of possible answers is reduced each unit of time. Despite the overhead introduced by the compiler, one of our approximate agreement protocols has an improved convergence rate for some small system sizes. Our improvement in the convergence rate does not contradict the proved optimality of the convergence rate of the protocol of Dolev *et al.* Their claim of optimality is for protocols of a particular form—a form which is very similar to the standard form defined in this paper. The output of our compiler is not of that form. Asynchronous approximate agreement protocols with optimal convergence rates, but large messages, have been designed for the crash and failure-by-omission models by Fekete [6].

The running time of either of our approximate agreement protocols depends only on the inputs to the correct processors and the size of the system. In the protocol of Dolev *et al.*, the faulty processors can choose the amount of time that will elapse before the correct processors decide.

We now give an outline of the remainder of the paper. In Section II, we define the approximate agreement problem. In Section III, we give our model for asynchronous protocols that operate in either the crash fault model or the Byzantine fault model. In Section IV, we present our compiler and we prove that it works correctly. In Section V, we give a new asynchronous approximate agreement protocol for the crash fault model and we prove it correct. We then note that this protocol can be compiled into a protocol for the Byzantine fault model.

II. The Approximate Agreement Problem

The approximate agreement problem is stated for various fault models including crash and Byzantine. The requirements given here apply to both of these fault models. In a protocol for the approximate agreement problem, each processor begins with some real number as its input. Each correct processor may, at some point during the execution of the protocol, irrevocably decide on a real number as its answer. A parameter ϵ specifies the precision required in the solution. There are three conditions that the correct processors must satisfy in all executions.

• *Agreement condition:* If v and v' are the decisions of two correct processors, then $|v - v'| \leq \epsilon$.

• *Validity condition:* If v is the decision of some correct processor, then there are correct processors with inputs i and i' such that $i \leq v \leq i'$.

• *Decision condition:* All correct processors eventually decide.

III. The Model

We model processors as state machines that communicate by sending messages. In Section III-A, we begin by defining some parameters that specify the size of the systems that we consider. In Section III-B, we give our model for asynchronous protocols subject to crash faults. In Section III-C, we define a subset of the possible executions in the crash fault model; we call them sequenced executions. In Section III-D, we give our model for asynchronous protocols subject to Byzantine faults. In Section III-E, we define correctness predicates as a way to formalize part of the definition of a problem in the asynchronous model. Finally, in Section III-F, we define our time measure.

A. Definition of Parameters

For the remainder of this paper, let n be the number of processors, let t be an upper bound on the number of processors that fail, and let $N = \{1, \cdots, n\}$. We define the *redundancy* to be $(n - 1)/t$. For the remainder of this paper, we assume that the redundancy is at least 3 and we let ϵ be an arbitrary fixed value of the parameter to the approximate agreement problem.

Let $\mathcal{I}^+$ be the set of positive integers; let $\mathcal{N}$ be the set of natural numbers, including 0; and let $\mathcal{R}$ be the set of real numbers.

B. The Crash Fault Model

A processor is modeled as an infinite state machine with a message buffer. The message buffer—modeled as a multiset of messages—holds those messages that have been sent to the processor but not yet received. Messages in the message buffer are tagged with the identity of the sending processor. In each step, a processor receives a set containing at most one message from its buffer and (based on its transition function) sends a finite set of messages. The transition function of a processor uses the current state and current set of messages received to compute a new state and a set of messages to be sent. There is a fixed set V of possible inputs to the processors. Without loss of generality, we assume that $* \notin V$ and $\perp \notin V$. For each element $v \in V$, each processor has one *initial state* that corresponds to having input v. The processors are indexed by the set N.

A *configuration* C is a vector of n states, one for each processor, and a vector of n multisets of messages, one for each message buffer. An *initial configuration* has all processors in initial states and all buffers equal to the empty multiset.

An *event* is denoted either (*step:* p) or (*receive:* p, q, m). The event (*step:* p) models processor p taking a step without receiving a message. The event e = (*step:* p) is *applicable* to any configuration. The configuration resulting from applying event e to configuration C, denoted $e(C)$, is obtained from C by changing the state of processor p according to the transition function and adding messages from processor p (tagged with sender p) to the appropriate buffers according to the transition function. The event (*receive:* p, q, m) models processor p receiving the message m from processor q. The event e' = (*receive:* p, q, m) is *applicable* to configuration C if the message m (tagged with sender q) is an element of the buffer of processor p in configuration C. The configuration resulting from applying event e' to configuration C, denoted $e'(C)$, is obtained from C by removing the message m from the buffer of processor p, changing the state of processor p according to the transition function, and adding messages from processor p (tagged with sender p) to the appropriate buffers according to the transition function.

A *schedule* is a finite or infinite sequence of events. A finite schedule $\sigma = e_1 e_2 \cdots e_k$ is *applicable* to configuration C if e_1 is applicable to C, e_2 is applicable to $e_1(C)$, etc. The resulting configuration is denoted $\sigma(C)$. An infinite schedule is applicable to configuration C if every finite prefix of the schedule is applicable to C.

For executions, we adopt a succinct representation which contains enough information to determine the behavior of every processor at all times. Formally, an *execution* of a protocol is a triple (F, I, σ) where $F \subset N$, where $I = \langle i_1, \cdots, i_n \rangle$ is a one-dimensional array of V, where σ is a schedule applicable to the initial configuration in which an arbitrary processor p begins in the initial state that corresponds to having input i_p, where all processors in $N - F$ take an infinite number of steps in σ, and where every message that is sent to a processor that takes an infinite number of steps in σ is eventually delivered. A processor p is *faulty* in the execution (F, I, σ) if $p \in F$; otherwise, p is *correct*. (In the case that some processor sends multiple identical messages to a processor that takes an infinite number of steps, our "eventual delivery" requirement should be taken to have the following meaning. If the number of copies sent is finite, then the number of copies delivered is equal to the number of copies sent. If the number of copies sent is infinite, then the number of copies delivered is infinite.)

Each processor has a *decision function* that maps from processor states to $V \cup \{*\}$. Recall that $* \notin V$. In the first step in which the decision function of a processor applied to the current state is $v \in V$, we say that the processor *decides* v. The intuition is that after a processor has decided, the value of its decision function is irrelevant. An execution is a *deciding execution* if all correct processors eventually decide. A protocol *decides* if all of its executions are deciding executions.

The model we have just given is one of the 32 crash fault models that were categorized and analyzed by Dolev, Dwork, and Stockmeyer [4]. Their 32 variants arise from five independent binary choices for five parameters that characterize the degree of asynchrony. The description of our model using these parameters follows. Processor step time is unbounded. Message delivery time is unbounded. Message delivery is unordered (i.e., any two messages sent to processor p can be delivered out of the order in which they were sent). A processor can send any finite set of messages in one step (i.e., processors have an "atomic broadcast" capability). A processor can receive and send as part of one step. The impossibility result of Fischer, Lynch, and Paterson holds for our model [4], [8].

There will be no atomic broadcast capability in our Byzantine fault model.

C. Sequenced Executions in the Crash Fault Model

In our correctness proofs, we construct certain executions for the crash fault model. All of these executions belong to the class of sequenced executions, which we now define. If $E = (F, I, \sigma)$ is an execution, C is the initial configuration in execution E, σ' is a prefix of σ, and $p \in N$, then we define deliver(E, σ', p) to be the set of schedules that 1) are applicable to the configuration $\sigma'(C)$, 2) deliver to processor p all of the messages that are in the buffer of p in the configuration $\sigma'(C)$, and 3) do nothing else. An execution $E = (F, I, \sigma)$ is *sequenced* if σ can be expressed as the concatenation of subschedules, $\sigma = \sigma_0\sigma_1\sigma_2 \cdots$, where $\sigma_r = \alpha(r, 1)\alpha(r, 2) \cdots \alpha(r, n)$, and where $\alpha(r, p)$ satisfies one of the following three conditions for all $r \in \mathfrak{N}$ and for all $p \in N$:

- *Condition 1:* $p \in F$ and, for all $r' \geq r$, $\alpha(r', p)$ is the empty sequence.
- *Condition 2:* $r = 0$ and $\alpha(r, p) =$ (*step:* p). Recall that (*step:* p) is the event in which processor p takes a step without receiving messages.
- *Condition 3:* $r \geq 1$ and $\alpha(r, p)$ is in deliver(E, $\sigma_0\sigma_1 \cdots \sigma_{r-1}$, p).

Intuitively a sequenced execution is organized into a series of "phases." In each phase, the processors are each given a "turn" (in ascending numerical order by processor number). In its first turn, a processor either takes a step without receiving messages or fails, never to recover. In each subsequent turn, a processor that is still operating either receives all of the messages that were in its buffer at the start of the current phase or fails, never to recover. In any turn in which a processor takes steps it, of course, sends messages according to its protocol.

We observe that the partitioning of any sequenced execution into phases is unique. Thus, we make the following definition. In a sequenced execution, we say that an event happens in *phase r* if it is in $\alpha(r, p)$ for some p.

D. The Byzantine Fault Model

The Byzantine fault model has much in common with the crash fault model. In this subsection, we define only those parts of the Byzantine fault model that differ from the crash fault model. The two differences are in the definition of events and in the definition of executions.

An *event* is denoted either (*step:* p), (*receive:* p, q, m), or (*error:* p, q, m). The events (*step:* p) and (*receive:* p, q, m) are defined as they are in the crash fault model. The event (*error:* p, q, m) models processor p erroneously sending the message m to processor q. The event $e =$ (*error:* p, q, m) is *applicable* to any configuration. The configuration resulting from applying event e to configuration C, denoted $e(C)$, is obtained from C by adding the message m (reliably tagged with sender p) to the buffer of processor q.

An *execution* of a protocol is a triple (F, I, σ) where $F \subset N$, where $I = \langle i_1, \cdots, i_n\rangle$ is a one-dimensional array of V, where σ is a schedule applicable to the initial configuration in which an arbitrary processor p begins in the initial state that corresponds to having input i_p, where all processors in $N - F$ take an infinite number of steps in σ, where every message that is sent to a processor in $N - F$ is eventually delivered, where processors in $N - F$ take no error steps, and where processors in F take only error steps. A processor p is *faulty* in the execution (F, I, σ) if $p \in F$; otherwise, p is *correct*.

In each message buffer, each message is tagged with its sender. The tags on all messages—even messages from faulty processors—are accurate. So a faulty processor does not have the ability to "impersonate" some other processor.

The requirement that a faulty processor take only error steps does not restrict the kinds of faults that can be exhibited in an execution. This is true because any message sending pattern can be achieved with error steps and because we are never interested in examining the state of a faulty processor. In particular, a faulty processor's error steps may mimic correct behavior for an arbitrary (even infinite) period of time.

We could give a definition of "sequenced executions" in the Byzantine fault model analogous to the definition of sequenced executions in the crash fault model. We omit that definition because we have no need to discuss such executions formally.

E. Correctness Predicates

In this subsection, we define correctness predicates as a way to formalize part of a problem definition (i.e., they formalize the relationship between inputs and outputs). A problem definition is formalized by requiring that all executions are deciding executions which satisfy some specific correctness predicate. Our formalism cannot be used to define problems for which nondeciding executions are acceptable. The definitions in this subsection apply to both the crash fault model and the Byzantine fault model.

Predicate $\mathcal{P}$ is a *correctness predicate* if its domain is $(V \cup \{\downarrow\})^{2n}$. If $E = (F, \langle i_1, \cdots, i_n\rangle, \sigma)$ is any execution, then inp(E) is defined to be $\langle i'_1, \cdots, i'_n\rangle$ where $i'_p = i_p$ if processor p is correct in E and $i_p = \downarrow$ otherwise. If E is any deciding execution, then ans(E) is defined to be $\langle a_1, \cdots, a_n\rangle$ where a_p is the decision of processor p in execution E if p is correct in E and $a_p = \downarrow$ otherwise. Protocol $\mathfrak{X}$ satisfies correctness predicate $\mathcal{P}$ if for any deciding execution E the value of $\mathcal{P}$(inp(E), ans(E)) is true. Correctness predicates furnish a convenient way of formalizing the correctness requirements for a consensus protocol. For example, protocol $\mathfrak{X}$ solves the approximate agreement problem (with parameter ϵ) if it decides and it satisfies correctness predicate $\mathcal{P}$ that is defined below. Let

$$\mathcal{A}(I, A) = \bigwedge_{j,k \in N} ((a_j = \downarrow) \vee (a_k = \downarrow) \vee (|a_j - a_k| \leq \epsilon)),$$

and

$$\mathcal{V}(I, A) = \bigwedge_{j \in N} \left((a_j = \downarrow) \vee \bigvee_{k,l \in N} ((i_k \neq \downarrow) \wedge (i_l \neq \downarrow) \wedge (i_k \leq a_j \leq i_l)) \right)$$

where $I = \langle i_1, \cdots, i_n \rangle$ and $A = \langle a_1, \cdots, a_n \rangle$. Now let $\mathcal{P}(I, A) = \mathcal{A}(I, A) \wedge \mathcal{V}(I, A)$. The correctness predicate $\mathcal{A}$ formalizes the agreement condition and the correctness predicate $\mathcal{V}$ formalizes the validity condition.

F. Time

In this subsection, we define a notion of time in asynchronous executions. We will use this notion when we discuss the performance of our compiler. The definitions in this subsection apply to both the crash fault model and the Byzantine fault model.

We define S to be a *timing* if S is an infinite nondecreasing unbounded sequence of real numbers. Let $E = (F, I, \sigma)$ be an execution. If event e is the ith element of σ, then the *time* at which event e occurs in timing S of E is r where r is the ith element of S. Timing S of execution E is 1-*bounded* if 1) each processor that takes a nonerror step in E takes its first step at time 0 and 2) any message that is sent at time x is delivered at or before time $x + 1$. (The time at which a particular message is sent is defined to be the time of the event that causes the message to be inserted in a buffer.) A *timed execution* is a pair (E, S) such that S is a 1-bounded timing of E.

Let $\mathcal{X}$ be a protocol and let $I = \langle i_1, \cdots, i_n \rangle$ be a vector of inputs to $\mathcal{X}$. Protocol $\mathcal{X}$ has *running time r for input I* if all correct processors decide by time r in all of the timed executions of protocol $\mathcal{X}$ where processor p has input i_p for all $p \in N$.

IV. The Compiler

We give two versions of our compiler. One works in any system where the redundancy is at least four. For any input, it increases the running time by a factor of two. The other works in any system where the redundancy is at least three. For any input, it increases the running time by a factor of three. We prove the correctness of the first version of the compiler. The correctness proof for the second version is similar and is only sketched.

It would be tedious to write protocols directly in terms of our formal model. In the remainder of this paper, we write protocols in a higher level language. The mapping from protocols written in the higher level language to protocols written directly in terms of the formal model is a straightforward exercise that we omit. To accommodate our new higher level language for expressing protocols, we will redefine the terms "transition function" and "decision function" in the body of this section.

A. Standard Protocols

Our compiler works only for protocols in standard form. A protocol is in standard form if it corresponds to an instance of Protocol 1 customized by specifying A, V, $\mathcal{S}$, and $\mathfrak{D}$. A is the set of possible values of the variable STATE. $V \subset A$ is the set of possible inputs. $\mathcal{S}: N \times \mathcal{I}^+ \times (A \cup \{\lambda\})^n \rightarrow A \cup \{\perp\}$ is the *transition function.* The transition function maps a triple consisting of a processor index, a positive integer (representing an "asynchronous round" number), and a vector of messages (λ represents the absence of a message) into either a processor state (i.e., element of A) or undefined (i.e., $\perp$). $\mathfrak{D}: N \times \mathcal{I}^+ \times A \rightarrow V \cup \{*\}$ is the *decision function.* The decision function maps a triple consisting of a processor index, a positive integer, and a processor state into a possible decision. In the range of the decision function, an element of V represents a decision and $*$ represents the absence of a decision. Throughout the rest of this paper, an instance of Protocol 1 customized with A, V, $\mathcal{S}$, and $\mathfrak{D}$ is denoted $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$. For the remainder of this section, we choose an arbitrary fixed A, V, $\mathcal{S}$, and $\mathfrak{D}$.

Protocol 1: The Standard Protocol (Crash Faults):

```
   Initialization for processor $p$:
      STATE ← the initial value of processor $p$
      $\text{MSG}_{l,q} \leftarrow \lambda$ for all $(l, q) \in \mathcal{I}^+ \times N$
1. for $r \leftarrow 1$ to $\infty$ do
2.    broadcast $(r, \text{STATE})$
3.    until $\text{MSG}_{r,p} \neq \lambda$ and $|\{q \in N: \text{MSG}_{r,q} \neq \lambda\}| \geq n - t$ do
4.       receive any message $(l, m)$ from any processor $q$
5.       $\text{MSG}_{l,q} \leftarrow m$
6.    STATE ← $\mathcal{S}(p, r, \langle \text{MSG}_{r,1}, \cdots, \text{MSG}_{r,n} \rangle)$
7.    DECISION ← $\mathfrak{D}(p, r, \text{STATE})$
8.    if DECISION $\in V$ then decide DECISION.
```

We impose the requirement that $\mathcal{S}(p, r, \langle m_1, \cdots, m_n \rangle) \neq \perp$ if and only if there is a set $G \subset N$ such that $p \in G$, $|G| \geq n - t$, and $m_q \neq \lambda$ for all $q \in G$. That is, $\mathcal{S}(p, r, \langle m_1, \cdots, m_n \rangle)$ is defined on exactly those message patterns (i.e., patterns of which elements of $\langle m_1, \cdots, m_n \rangle$ are defined) that would cause a correct processor p to exit the inner loop (steps 3-5) and proceed to step 6. We remark that it is essential to the operation of Protocol 1 that the transition function is defined on those message patterns on which it may be evaluated, but it is only as a technical convenience that we require that it is undefined on all other message patterns.

A standard-form protocol operates in a series of *asynchronous rounds.* The rth execution of the body of the main loop is asynchronous round r. A processor *ends asynchronous round r* when it completes the last instruction in its rth execution of the body of the main loop. At the start of each asynchronous round, a correct processor broadcasts a message containing its state. It then waits to receive messages from a sufficiently large group of processors (including itself). It computes its new state by applying its transition function to a triple consisting of its index, its current asynchronous round number, and the vector of messages received. Finally, it (possibly) decides on an answer by applying its decision function to its new state. It may seem unusual that a correct processor sends a copy of its state to itself (and waits to receive it) in each asynchronous round. We adopt this convention as a technical convenience which simplifies our compiler.

We say that M is a *message array* if it is a two-dimensional array of $A \cup \{\lambda\}$ indexed by $\mathcal{I}^+$ and N (asynchronous rounds and processor indexes). Message array L is an *extension* of message array M if for all r and p either $L_{r,p} = M_{r,p}$ or $M_{r,p} = \lambda$. Let $\mathfrak{M}$ be the set of all message arrays. In protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ the variable MSG at any processor is always an element of $\mathfrak{M}$. If the value at processor p of $\text{MSG}_{r,q}$ is ever m, then p received the message (r, m) from processor q. The value at processor p of $\text{MSG}_{r,q}$ is λ if p has received no message (r, m) from processor q for any m. It is easy to see from the code that if processor q sends a message (r, m) to processor p, then it never sends a message (r, m') to processor p. Thus, at any time, the message array MSG at processor p contains all of the messages received by processor p up to the present time. Note that messages that arrive "too late" are stored in MSG at processor p but do not affect the state or decision of processor p.

B. Filtered Message Arrays

In this subsection, we define a filter that we will use in both versions of our compiler. This filter operates on message arrays. It obliterates (replaces by λ) messages that seem "implausible" and it passes all other messages unchanged. It is an adaptation of a "validation" scheme due to Bracha [1]. We now define the filter.

For all $G \subset N$, define pick$(G, \langle v_1, \cdots, v_n \rangle)$ to be $\langle v_1', \cdots, v_n' \rangle$ where $v_i' = v_i$ if $i \in G$ and $v_i' = \lambda$ otherwise. The function pick returns a vector in which those elements of $\langle v_1, \cdots, v_n \rangle$ with indexes in $N - G$ are replaced by λ. The function filter maps from $\mathfrak{M}$ to $\mathfrak{M}$. Define filter(L) to be M where

$$M_{1,p} = \begin{cases} L_{1,p} & \text{if } L_{1,p} \in V; \\ \lambda & \text{otherwise,} \end{cases}$$

and

$$M_{r,p} = \begin{cases} L_{r,p} & \text{if } \exists_{G \subset N} L_{r,p} = \mathcal{S}(p, r-1, \\ & \quad \text{pick}(G, \langle M_{r-1,1}, \cdots, M_{r-1,n} \rangle)); \\ \lambda & \text{otherwise,} \end{cases}$$

for all $r \in \{2, 3, \cdots\}$ and for all $p \in N$. For all p, $M_{1,p}$ is equal to $L_{1,p}$ if and only if $L_{1,p}$ is an element of V, the set of possible inputs to the protocol. For all p and for all $r \geq 2$, $M_{r,p}$ is equal to $L_{r,p}$ if and only if there is some set $G \subset N$ such that $L_{r,p}$ is the message that would be sent by a correct processor p that received the messages in pick$(G, \langle M_{r-1,1}, \cdots, M_{r-1,n} \rangle)$.

In the next three lemmas, we prove basic properties of filtered message arrays.

Lemma 1: Suppose that L is a message array and $M =$ filter(L). For all $r \in \mathcal{I}^+$ and $p \in N$, if $M_{r,p} = \lambda$, then $M_{r+1,p} = \lambda$.

Proof: Immediate from the definition of filter(L). □

Lemma 2: If L is a message array, then L is an extension of filter(L).

Proof: Immediate from the definition of filter(L). □

Lemma 3: If message array L is an extension of message array M, then filter(L) is an extension of filter(M).

Proof: Let $L' =$ filter(L) and let $M' =$ filter(M). We prove by induction on r that, for all r and for all p, either $L'_{r,p} = M'_{r,p}$ or $M'_{r,p} = \lambda$.

Basis: ($r = 1$): Consider an arbitrary $p \in N$. If $M'_{1,p} = \lambda$, then the claim is trivially true, so assume that $M'_{1,p} \neq \lambda$. By Lemma 2, $M'_{1,p} = M_{1,p}$. By the definition of filter(M), $M_{1,p} \in V$. Because L is an extension of M, $L_{1,p} = M_{1,p}$. By the definition of filter(L), $L'_{1,p} = L_{1,p}$. Thus, $L'_{1,p} = M'_{1,p}$.

Induction: Consider an arbitrary $p \in N$. If $M'_{r,p} = \lambda$, then the claim is trivially true, so assume that $M'_{r,p} \neq \lambda$. By Lemma 2, $M'_{r,p} = M_{r,p}$. Because L is an extension of M, $L_{r,p} = M_{r,p}$. By the definition of filter(M), there is some set $G \subset N$ such that $M'_{r-1,q} \neq \lambda$ for all $q \in G$ and

$$M_{r,p} = \mathcal{S}(p, r-1, \text{pick}(G, \langle M'_{r-1,1}, \cdots, M'_{r-1,n} \rangle)).$$

By the induction hypothesis,

$$\text{pick}(G, \langle M'_{r-1,1}, \cdots, M'_{r-1,n} \rangle) = \text{pick}(G, \langle L'_{r-1,1}, \cdots, L'_{r-1,n} \rangle).$$

Thus, by the definition of filter(L), $L'_{r,p} = L_{r,p}$. So we have that $L'_{r,p} = M'_{r,p}$. □

C. The Object Protocol

Our compiler operates by translating an instance of Protocol 1 customized by A, V, $\mathcal{S}$, and $\mathcal{D}$ into an instance of Protocol 2 customized by A, V, $\mathcal{S}$, and $\mathcal{D}$. Throughout the rest of this paper, an instance of Protocol 2 customized with A, V, $\mathcal{S}$, and $\mathcal{D}$ is denoted $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$.

The skeleton of Protocol 2 is similar to that of Protocol 1. Four important features of Protocol 2 should be noted. First, Protocol 2 requires that the redundancy be at least four. Second, there is extra communication in Protocol 2. Each processor uses this extra communication to construct a message array called RAW. Third, in Protocol 2 each correct processor applies its transition function to its copy of the message array MSG, which it obtains by filtering its copy of the message array RAW. Fourth, the exit test in step 3 ensures that the transition function is defined (i.e., not $\perp$) whenever it is evaluated.

Protocol 2: The Object Protocol (Byzantine Faults, $n \geq 4t + 1$):

```
    Initialization for processor $p$:
        STATE ← the initial value of processor $p$
        $\text{VOTE}_{l,q,i,u}$ ← $\lambda$ for all $(l, q, i, u) \in \mathcal{I}^+ \times N \times \mathfrak{M} \times N$
        $\text{RAW}_{l,q}$ ← $\lambda$ for all $(l, q) \in \mathcal{I}^+ \times N$
        $\text{MSG}_{l,q}$ ← $\lambda$ for all $(l, q) \in \mathcal{I}^+ \times N$
 1. for $r$ ← 1 to ∞ do
 2.     broadcast $(r, p, 0, \text{STATE})$
 3.     until $\text{MSG}_{r,p} \neq \lambda$ and $|\{q \in N: \text{MSG}_{r,q} \neq \lambda\}| \geq n - t$ do
 4.         receive any message $(l, q, i, m)$ from any processor $u$
 5.         if $\text{VOTE}_{l,q,i,u} = \lambda$ then
 6.             $\text{VOTE}_{l,q,i,u}$ ← $m$
 7.             NUM ← $|\{s \in N: \text{VOTE}_{l,q,i,s} = m\}|$
 8.             if $i = 0$ and $q = u$ then broadcast $(l, q, 1, m)$
 9.             if NUM $= n - 2t$ then broadcast $(l, q, i + 1, m)$
10.             if NUM $= n - t$ then $\text{RAW}_{l,q}$ ← $m$
11.             MSG ← filter(RAW)
```

```
12.   STATE ← $\mathcal{S}(p, r, \langle \text{MSG}_{r,1}, \cdots, \text{MSG}_{r,n}\rangle)$
13.   DECISION ← $\mathcal{D}(p, r, \text{STATE})$
14.   if DECISION $\in V$ then decide DECISION.
```

As in a standard-form protocol, Protocol 2 operates in a series of *asynchronous rounds.* The rth execution of the body of the main loop is asynchronous round r. At the start of each asynchronous round, a correct processor p broadcasts a message containing the current value of the variable STATE at processor p. It then waits until its filtered message array (i.e., the value of MSG at processor p) contains messages from a sufficiently large group of processors (including itself). It computes a new value for its copy of the variable STATE by applying its transition function to a triple consisting of its index, its current asynchronous round number, and the vector of messages in its filtered message array. Finally, it (possibly) decides on an answer by applying its decision function to its new value of STATE.

A correct processor in Protocol 2 locally maintains the invariant that MSG = filter(RAW), except in steps 10 and 11. It updates its copy of the array RAW in step 10 and reestablishes the invariant in step 11. A correct processor places a message in its copy of the array RAW when it has accumulated enough votes for that message. It stores the votes that it receives in its copy of the array VOTES. If $\text{VOTES}_{l,q,i,u} = m$ at correct processor p, then processor u sent processor p a *level i* vote that $\text{RAW}_{l,q}$ should be m. If a correct processor receives a level 0 vote from processor q that $\text{RAW}_{l,q}$ should be m, then it sends a level 1 vote that $\text{RAW}_{l,q}$ should be m. If a correct processor receives $n - 2t$ level i votes that $\text{RAW}_{l,q}$ should be m, then it sends a level $i + 1$ vote that $\text{RAW}_{l,q}$ should be m. If it receives $n - t$ level i votes that $\text{RAW}_{l,q}$ should be m, then it sets its copy of $\text{RAW}_{l,q}$ to m.

D. Preliminary Lemmas

In this subsection, we give five lemmas that will be of use in our main correctness argument in the next subsection. The first four lemmas establish some basic properties of the communication primitives used in Protocol 2 and the last lemma establishes an important liveness property for Protocol 2.

Lemma 4: Let $r \in \mathcal{I}^+$ and $p \in N$. If any correct processor ever assigns any value m to its copy of $\text{RAW}_{r,p}$ in an execution of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$, then every correct processor eventually assigns the value m to its copy of $\text{RAW}_{r,p}$ and no correct processor ever assigns any value $m' \neq m$ to its copy of $\text{RAW}_{r,p}$.

Proof: Any correct processor that assigns a value to its copy of $\text{RAW}_{r,p}$ does it immediately after receiving an (r, p, i, m') message for some $i \in \mathfrak{N}$ and $m' \in A$. We call such a message a *level i* message. Let j be the smallest number such that a level j message causes some correct processor to assign a value of its copy of $\text{RAW}_{r,p}$, let q be such a processor, and let d be the value assigned. Processor q gets $n - t$ messages (r, p, j, d). At least $n - 2t$ are from correct processors. There are at most $2t$ processors that could send an (r, p, j, d') message for any $d' \neq d$. Thus, no correct processor assigns any $d' \neq d$ to its copy of $\text{RAW}_{r,p}$ based on a level j message. It is easy to show by induction on j' that for all $j' \geq j + 1$ and for all $d' \neq d$ there are no (r, p, j', d') messages sent by any correct processor. All correct processors eventually receive $n - 2t$ messages (r, p, j, d) and broadcast an $(r, p, j + 1, d)$ message. Thus, each correct processor eventually receives $n - t$ messages $(r, p, j + 1, d)$ and assigns d to its copy of $\text{RAW}_{r,p}$. □

Based on Lemma 4, for all $r \in \mathcal{I}^+$ and $p \in N$, we define the *eventual value* of $\text{RAW}_{r,p}$ in an execution of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$, denoted $[\text{RAW}_{r,p}]$, to be the common value assigned to $\text{RAW}_{r,p}$ by the correct processors. If the correct processors never assign a value to $\text{RAW}_{r,p}$, then we define $[\text{RAW}_{r,p}]$ to be λ. (Note that $[\text{RAW}_{r,p}]$ is an abstract global variable which we define based on the many local copies of $\text{RAW}_{r,p}$ maintained by the correct processors.) We define the *eventual value* of RAW, denoted [RAW], in the obvious way. That is, [RAW] is the two-dimensional array whose elements are the eventual values of the corresponding elements of RAW. Based on Lemmas 3 and 4, we define the *eventual value* of $\text{MSG}_{r,p}$ and MSG analogously.

Lemma 5: If a correct processor p ever broadcasts the message $(r, p, 0, m)$ for any r and m, then $[\text{RAW}_{r,p}] = m$.

Proof: All $n - t$ correct processors eventually receive the message $(r, p, 0, m)$ from processor p. They all broadcast the message $(r, p, 1, m)$. All $n - t$ correct processors eventually receive at least $n - t$ copies of the message $(r, p, 1, m)$ and at most t copies of any message $(r, p, 1, m')$ for any $m' \neq m$. Each correct processor assigns m to its local copy of $\text{RAW}_{r,p}$, and so $[\text{RAW}_{r,p}] = m$. □

Lemma 6: Let $r \in \mathcal{I}^+$. If a correct processor p never broadcasts a message $(r, p, 0, m)$ for any m, then $[\text{RAW}_{r,p}] = \lambda$.

Proof: It is easy to show by induction on i that no correct processor ever sends a message (r, p, i, m) for any m. Therefore, no correct processor ever assigns any value to its copy of $\text{RAW}_{r,p}$ and so $[\text{RAW}_{r,p}] = \lambda$. □

Lemma 7: If M is the value of the variable RAW at some processor p at some time in an execution and L is the value of the same variable at processor p at some later time in the same execution, then L is an extension of M.

Proof: Immediate from Lemma 4. □

In the next lemma, we prove an important liveness property of Protocol 2.

Lemma 8: $[\text{MSG}_{r,p}] \neq \lambda$ for all $r \in \mathcal{I}^+$ and for all $p \in N - F$.

Proof: The proof is by induction on r.

Basis: ($r = 1$): Let p be an arbitrary correct processor. Let $v \in V$ be the input to processor p. In its first step, processor p sends the message $(1, p, 0, v)$. By Lemma 5, $[\text{RAW}_{1,p}] = v$. By the definition of filter, $[\text{MSG}_{1,p}] = v$.

Induction: Let p be an arbitrary correct processor. By the induction hypothesis, $[\text{MSG}_{r-1,p}] \neq \lambda$. By Lemma 2, $[\text{RAW}_{r-1,p}] \neq \lambda$. By Lemma 6, there is some time when processor p sends the message $(r - 1, p, 0, m')$ for some m'. Thus, there is some time at which processor p executes the broadcast (step 2) in asynchronous round $r - 1$.

By the induction hypothesis, $[\text{MSG}_{r-1,q}] \neq \lambda$ for all $q \in N - F$. By Lemmas 3 and 7, there is some time after which the variable MSG at processor p always satisfies the condition that $\text{MSG}_{r-1,q} \neq \lambda$ for all $q \in N - F$. Therefore, processor p eventually sends some message $(r, p, 0, m)$. Let M be the value of the variable MSG at processor p when processor p sends the message $(r, p, 0, m)$. It follows from the code that

$$m = \mathcal{S}(p, r-1, \langle M_{r-1,1}, \cdots, M_{r-1,n}\rangle).$$

By Lemmas 3 and 4, [MSG] is an extension of M. Therefore, there is a $G \subset N$ such that

$$m = \mathcal{S}(p, r-1, \text{pick}(G, \langle[\text{MSG}_{r-1,1}], \cdots, [\text{MSG}_{r-1,n}]\rangle)).$$

By Lemma 5, $[\text{RAW}_{r,p}] = m$. By the definition of filter, $[\text{MSG}_{r,p}] = [\text{RAW}_{r,p}]$. Thus, $[\text{MSG}_{r,p}] \neq \lambda$. □

E. Proof of Correctness

In this subsection, we show for any A, V, $\mathcal{S}$, and $\mathcal{D}$ that if protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ solves some problem (formalized by a correctness predicate and a requirement that all correct processors eventually decide), then protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$ solves the same problem. Recall that protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ operates in the crash fault model and protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$ operates in the Byzantine fault model. Our approach is to exhibit for any execution E of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$ an execution of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ in which the correct processors "do the same thing" that they did in execution E. More specifically the property we seek is that the "eventual value" of MSG in the constructed execution of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ is equal to the eventual value of MSG in execution E of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$.

Let v_0 be an arbitrary fixed element of V. In this subsection, we will find it convenient to use v_0 as a "default input."

We now construct for any execution E of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$ a sequenced execution of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$, denoted crash($E$), in which the correct processors "do the same thing" that they did in E. Because crash(E) is a sequenced execution in the crash fault model, it is completely determined by the following three items: 1) the inputs to the processors, 2) the number, if any, of the last phase in which each processor takes steps, and 3) the order in which each operating processor receives its phase r messages for each r. We will construct crash(E) so that the following three properties hold: 1) the input to processor p is $[\text{MSG}_{1,p}]$ if $[\text{MSG}_{1,p}] \neq \lambda$ and v_0 otherwise, 2) a processor p sends a phase r message if and only if $[\text{MSG}_{r,p}] \neq \lambda$, and 3) messages are delivered to a processor p in phase r in an order that causes processor p to send the message $(r+1, [\text{MSG}_{r+1,p}])$.

Suppose $p \in N$, $r \in \mathfrak{N}$, and $\mathfrak{M}$ is a message array. Define support(p, r, M) to be the lexicographically least set $G \subset N$ such that $M_{r-1,q} \neq \lambda$ for all $q \in G$ and

$$M_{r,p} = \mathcal{S}(p, r-1, \text{pick}(G, \langle M_{r-1,1}, \cdots, M_{r-1,n}\rangle)).$$

If there is no such set G, then define support(p, r, M) to be $\emptyset$.

If $M_{1,p} = \lambda$, then define $\beta(0, p, M)$ to be the empty sequence of events; otherwise, define $\beta(0, p, M)$ to be the event (*step:* p). For all $r \geq 1$, if $M_{r+1,p} = \lambda$, then define $\beta(r, p, M)$ to be the empty sequence of events; otherwise, define $\beta(r, p, M)$ to be the sequence of events that consists of 1) the receipt by processor p of all of the (non-λ) messages in $\langle M_{r,1}, \cdots, M_{r,n}\rangle$ that are from processors in support($p, r+1, M$) $- \{p\}$ followed by 2) the receipt by processor p of $M_{r,p}$ followed by 3) the receipt by processor p of all of the remaining (non-λ) messages in $\langle M_{r,1}, \cdots, M_{r,n}\rangle$.

Suppose $E = (F, I, \sigma)$ is an arbitrary execution of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$. We define crash($E$) to be $(F, \langle i_1', \cdots, i_n'\rangle, \sigma_0'\sigma_1' \cdots)$ where

$$i_p' = \begin{cases} [\text{MSG}_{1,p}] & \text{if } [\text{MSG}_{1,p}] \neq \lambda; \\ v_0 & \text{otherwise,} \end{cases}$$

and for all r

$$\sigma_r' = \beta(r, 1, [\text{MSG}])\beta(r, 2, [\text{MSG}]), \cdots \beta(r, n, [\text{MSG}]).$$

Lemma 9: Let $p \in N$, $q \in N$, $r \in \mathfrak{N}$, and $m \in A$. Let L be a message array. Let M = filter(L). Let C be an initial configuration for protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$. Let $\sigma' = \sigma_0\sigma_1 \cdots \sigma_{r-1}$ where (for $0 \leq r' \leq r-1$)$\sigma_{r'} = \beta(r', 1, M) \cdots \beta(r', n, M)$. Suppose that σ' is a schedule that is applicable to configuration C. Suppose that processor p sends an (r, m) message to processor q in $\sigma'(C)$. Then $m = M_{r,p}$.

Proof: There are two cases.

Case 1: ($r = 1$): Let v be the input to processor p in configuration C. Clearly, $m = v$. To send the message $(1, m)$, processor p must take at least one step in schedule σ'. It follows from Lemma 1 and the definition of σ' that $M_{1,p} \neq \lambda$. By the construction of σ', $v = M_{1,p}$. Thus, $m = M_{1,p}$.

Case 2: ($r \geq 2$): Let M' be the value of the variable MSG at processor p in protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ after the application of the event that causes the sending of the message (r, m) from processor p to processor q. By Lemmas 3 and 7, message array M is an extension of message array M'. Because the message (r, m) is sent in $\sigma'(C)$, we have that support(p, r, M) $\neq \emptyset$. We claim that

$$\langle M'_{r-1,1}, \cdots, M'_{r-1,n}\rangle = \text{pick}(\text{support}\ (p, r, M), \langle M_{r-1,1}, \cdots, M_{r-1,n}\rangle).$$

The claim follows by the definition of $\beta(r-1, p, M)$. Using the claim, we calculate that

$$\begin{aligned} m &= \mathcal{S}(p, r-1, \langle M'_{r-1,1}, \cdots, M'_{r-1,n}\rangle) && \text{From the code.} \\ &= \mathcal{S}(p, r-1, \text{pick}(\text{support}\ (p, r, M), \\ &\quad \langle M_{r-1,1}, \cdots, M_{r-1,n}\rangle)) && \text{By the claim.} \\ &= M_{r,p}. && \text{By the definition of support.} \end{aligned}$$

□

Lemma 10: If $E = (F, \langle i_1, \cdots, i_n\rangle, \sigma)$ is an execution of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$, then E' = crash(E) is an execution of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$.

Proof: Suppose that $E' = (F, I', \sigma')$ where $I' = \langle i_1', \cdots, i_n'\rangle$. Partition the schedule σ' into subschedules σ_0', σ_1', etc., with σ_i' defined as it is in the definition of crash.

To show that E' is an execution of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ we verify the following three properties. 1) The schedule σ' is applicable to the initial configuration C of $\mathcal{C}(A, V, \mathcal{S}, \mathcal{D})$ in which F is the set of faulty processors and in which processor p begins in the initial state that corresponds to input i_p' for all $p \in N$. 2) All processors in $N - F$ take an infinite number of steps. 3) If processor q takes an infinite number of steps, then every message that is sent to processor q is eventually delivered.

Property 1: We prove for all $r \in \mathfrak{N}$ that all events in σ'_r are applicable. The proof is by induction on r.

Basis: ($r = 0$): All events in σ'_0 are of the form (*step:* p) for some $p \in N$. It is immediate that σ'_0 is applicable to configuration C.

Induction: Pick an arbitrary event $e = (\textit{receive: } q, p, (r, [\text{MSG}_{r,p}]))$ from the schedule σ'_r. By the induction hypothesis, the schedule $\sigma'_0 \sigma'_1 \cdots \sigma'_{r-1}$ is applicable to configuration C. The event e is in the schedule $\beta(r, q, [\text{MSG}])$. By the definition of $\beta(r, q, [\text{MSG}])$, $[\text{MSG}_{r,p}] \neq \lambda$. By the definition of $\beta(r-1, p, [\text{MSG}])$, processor p sends some message (r, m) to processor q in $\sigma'_0 \sigma'_1 \cdots \sigma'_{r-1}(C)$. By Lemma 9, $m = [\text{MSG}_{r,p}]$. Thus, the message $(r, [\text{MSG}_{r,p}])$ is placed in the buffer of processor q. It is easy to see that the event e is unique in the schedule σ'. Therefore, the message $(r, [\text{MSG}_{r,p}])$ is in the buffer of processor q in the configuration just before the event e is applied. It follows that the schedule $\sigma'_0 \sigma'_1 \cdots \sigma'_r$ is applicable to configuration C.

Property 2: We prove that all processors in $N - F$ take an infinite number of steps. Let p be an arbitrary element of $N - F$. By Lemma 8, $[\text{MSG}_{r,p}] \neq \lambda$ for all $r \in \mathcal{I}^+$. By the construction of σ', each (non-λ) message in [MSG] is delivered to processor p in the execution (F, I', σ'). There are an infinite number of such messages To receive all of these messages, processor p must take an infinite number of steps.

Property 3: Let q by any processor that takes an infinite number of steps in execution E'. We prove that every message sent to processor q is eventually delivered. Consider an arbitrary message (r, m) sent from a processor p to the processor q in E'. By property 1 and Lemma 9, $m = [\text{MSG}_{r,p}]$.

We now show that the message $(r, [\text{MSG}_{r,p}])$ is delivered to processor q in execution E'. Processor q takes an infinite number of steps in execution E'. By the definition of E', $[\text{MSG}_{r',q}] \neq \lambda$ for infinitely many r'. By Lemma 8, $[\text{MSG}_{r',q}] \neq \lambda$ for all r'. So $[\text{MSG}_{r+1,q}] \neq \lambda$. By construction, the schedule $\beta(r, q, [\text{MSG}])$ includes the event $(q, p, (r, [\text{MSG}_{r,p}]))$. Thus, the message (r, m) is delivered in the execution E'. □

Lemma 11: If E is any execution of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$, then the executions E and $E' = \text{crash}(E)$ have the same set of faulty processors and the same inputs to the correct processors.

Proof: It is immediate that executions E and E' have the same faulty processors. We now show that the correct processors have the same inputs in executions E and E'. Let p be an arbitrary correct processor. Suppose that in execution E processor p has input $v \in V$. Processor p broadcasts the message $(1, p, 0, v)$ in its first step in execution E. By Lemma 5, $[\text{RAW}_{1,p}] = v$. By the definition of filter, $[\text{MSG}_{1,p}] = v$. Thus, the input to processor p in execution E' is v. □

If E is an execution of either protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ or protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$, processor p is correct in E, and $r \in \mathfrak{N}$, then we define state(p, r, E) to be the $(r+1)$st value that processor p assigns to its copy of the variable STATE in the execution E. For example, state$(p, 0, E)$ is the input to processor p.

Lemma 12: If E is an execution of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$ and $E' = \text{crash}(E)$, then state$(p, r, E) =$ state(p, r, E') for all $(p, r) \in (N - F) \times \mathcal{I}^+$.

Proof: If $r = 0$, then the claim follows from Lemma 11. Suppose instead that $r \geq 1$. By the construction of the execution E',

$$\text{STATE}(p, r, E') = \mathcal{S}(p, r+1, \text{pick}(\text{support } (p, r, [\text{MSG}]), \langle [\text{MSG}_{r,1}], \cdots, [\text{MSG}_{r,n}] \rangle)).$$

By the definition of support

$$\mathcal{S}(p, r+1, \text{pick}(\text{support } (p, r, [\text{MSG}]), \langle [\text{MSG}_{r,1}], \cdots, [\text{MSG}_{r,n}] \rangle)) = [\text{MSG}_{r+1,p}].$$

By the definition of filter, $[\text{MSG}_{r+1,p}] = [\text{RAW}_{r+1,p}]$. By Lemma 5, $\text{STATE}(p, r, E) = [\text{RAW}_{r+1,p}]$. Thus, $\text{STATE}(p, r, E') = \text{STATE}(p, r, E)$.

Theorem 13: If $n \geq 4t + 1$, then the following two conditions hold.

- *Correctness condition:* If protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ satisfies some correctness predicate $\mathcal{P}$, then so does protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$.
- *Decision condition:* If protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ decides, then so does protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$.

Proof: We verify that the two conditions are satisfied.

Correctness condition: Suppose protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ satisfies correctness predicate $\mathcal{P}$. Let $E = (F, I, \sigma)$ be an arbitrary deciding execution of $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$. Let $E' = \text{crash}(E)$. By Lemma 10, E' is an execution of $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$. Suppose $E' = (F', I', \sigma')$. By Lemma 11, $F = F'$ and $\text{inp}(E) = \text{inp}(E')$. By Lemma 12, E' is a deciding execution and $\text{ans}(E) = \text{ans}(E')$. Therefore, $\mathcal{P}(\text{inp}(E), \text{ans}(E)) = \mathcal{P}(\text{inp}(E'), \text{ans}(E'))$ and $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$ also satisfies correctness predicate $\mathcal{P}$.

Decision condition: We prove the contrapositive of the claim. Suppose that protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$ does not decide. By the definition of decision, there is some nondeciding execution E of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$. Let $E' = \text{crash}(E)$. By Lemma 10, E' is an execution of $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$. By Lemma 12, execution E' is a nondeciding execution of $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$. Thus, protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ does not decide. □

Theorem 14: Let $I \in V^n$. If protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ has running time r for input I and $n \geq 4t + 1$, then protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$ has running time $2 \cdot r$ for input I.

Proof: By assumption, all correct processors decide by time r in all timed executions of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ with input I. We claim that all correct processors decide by the end of asynchronous round r in all executions of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ with input I. Suppose not. Then, there is some execution E of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ with input I in which a correct processor decides in asynchronous round r' for some $r' > r$. We can construct a sequenced execution E' of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ in which the sequence of values assigned to the variable STATE at each correct processor is the same as it is in execution E. It should be clear that each correct processor decides in the same asynchronous round in executions E and E'. For the sequenced execution E', consider the obvious 1-bounded timing S' where receiving an asynchronous round r'' message takes place at time r''. In the timed execution (E', S'), there is a correct processor which decides at time r'. This contradicts the assumption that there is no such execution. Thus, we have proved our claim that all correct processors decide by the end of asynchronous round r in all executions of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ with input I.

Let (E'', S'') be an arbitrary timed execution of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$ with input I. It follows from Lemma 12 that if there is a correct processor in the execution crash(E'') of protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ that decides in asynchronous round r' for some r', then the same correct processor decides in asynchronous round r' in execution E'' of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$. It follows from the claim that all correct processors in execution E'' decide by asynchronous round r.

We can show by induction on r''' that in the timed execution (E'', S'') of protocol $\mathcal{B}(A, V, \mathcal{S}, \mathfrak{D})$, all processors end asynchronous round r''' at or before time $2 \cdot r'''$. Thus, all correct processors in the timed execution (E'', S'') decide by time $2 \cdot r$. □

F. An Alternative Object Protocol

The alternative version of our compiler operates by translating an instance of Protocol 1 customized by A, V, $\mathcal{S}$, and $\mathfrak{D}$ into an instance of Protocol 3 customized by A, V, $\mathcal{S}$, and $\mathfrak{D}$. Throughout the rest of this paper, an instance of Protocol 3 customized with A, V, $\mathcal{S}$, and $\mathfrak{D}$ is denoted $\mathcal{B}'(A, V, \mathcal{S}, \mathfrak{D})$. Protocol 3 requires that the redundancy be at least three. It uses the function filter defined in Section IV-B.

The only difference between Protocol 3 and Protocol 2 is that in Protocol 3 we use a different set of communication primitives to install elements in the message array RAW. They are the communication primitives developed by Bracha [1].

Protocol 3: The Object Protocol (Byzantine Faults, $n \geq 3t + 1$):

```
Initialization for processor $p$:
    STATE $\leftarrow$ the initial value of processor $p$
    VOTE$_{l,q,i,u} \leftarrow \lambda$ for all $(l, q, i, u) \in \mathcal{I}^+ \times N \times \mathfrak{M} \times N$
    RAW$_{l,q} \leftarrow \lambda$ for all $(l, q) \in \mathcal{I}^+ \times N$
    MSG$_{l,q} \leftarrow \lambda$ for all $(l, q) \in \mathcal{I}^+ \times N$
1.  for $r \leftarrow 1$ to $\infty$ do
2.      broadcast $(r, p, 0,$ STATE$)$
3.      until MSG$_{r,p} \neq \lambda$ and $|\{q \in N$: MSG$_{r,q} \neq \lambda\}| \geq n - t$
        do
4.          receive any message $(l, q, i, m)$ from any processor
            $u$
5.          if VOTE$_{l,q,i,u} = \lambda$ then
6.              VOTE$_{l,q,i,u} \leftarrow m$
7.              NUM $\leftarrow |\{s \in N$: VOTE$_{l,q,i,s} = m\}|$
8.              if $i = 0$ and $q = u$ then broadcast $(l, q, 1, m)$
9.              if $i = 1$ and NUM $= n - t$ then broadcast
                    $(l, q, 2, m)$
10.             if $i = 2$ and NUM $= n - 2t$ then broadcast
                    $(l, q, 2, m)$
11.             if $i = 2$ and NUM $= n - t$ then RAW$_{l,q} \leftarrow m$
12.             MSG $\leftarrow$ filter(RAW)
13.     STATE $\leftarrow \mathcal{S}(p, r, \langle$MSG$_{r,1}, \cdots,$ MSG$_{r,n}\rangle)$
14.     DECISION $\leftarrow \mathfrak{D}(p, r,$ STATE$)$
15.     if DECISION $\in V$ then decide DECISION.
```

Theorem 15: If $n \geq 3t + 1$, then the following two conditions hold.

• *Correctness condition:* If protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ satisfies some correctness predicate $\mathcal{P}$, then so does protocol $\mathcal{B}'(A, V, \mathcal{S}, \mathfrak{D})$.

• *Decision condition:* If protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ decides, then so does protocol $\mathcal{B}'(A, V, \mathcal{S}, \mathfrak{D})$.

Proof sketch: We begin by proving the analogues of Lemmas 5, 6, 7, and 8. The remainder of the proof is identical to the proof of Theorem 13. □

Theorem 16: Let $I \in V^n$. If protocol $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$ has running time r for input I and $n \geq 3t + 1$, then protocol $\mathcal{B}'(A, V, \mathcal{S}, \mathfrak{D})$ has running time $3 \cdot r$ for input I.

Proof: Similar to the proof of Theorem 14. □

V. Approximate Agreement Protocols

We use our compiler to simplify the design of a new approximate agreement protocol that operates in the Byzantine fault model. In Section V-A, we review some definitions and basic results regarding multisets. In Section V-B, we give an approximate agreement protocol that operates in the crash fault model. In Section V-C, we prove our approximate agreement protocol correct in the crash fault model. In Section V-D, we apply the two versions of our compiler to the approximate agreement protocol given in Section V-B. In the Byzantine fault model, we compare the performance of our compiled protocols to the performance of the protocol of Dolev *et al.*

A. Preliminary Definitions

We give some definitions and prove some basic facts about multisets. The presentation in this subsection borrows heavily from Dolev *et al.* [5]. Lemma 19 of this subsection is very similar to Lemma 5 of Dolev *et al.*

We view a finite multiset U of reals as a function $U: \mathcal{R} \rightarrow \mathfrak{N}$ that is nonzero on at most finitely many $r \in \mathcal{R}$. Intuitively, the function U assigns a multiplicity to each real number. In the remainder of this section, the term *multiset* always refers to finite multisets of reals as described above.

The *cardinality* of a multiset U is given by $\Sigma_{r \in \mathcal{R}} U(r)$ and is denoted by $|U|$. A multiset is *empty* if its cardinality is 0; otherwise it is *nonempty*. Multiset U is a *subset* of multiset V, written $U \subset V$, if $U(r) \leq V(r)$ for all $r \in \mathcal{R}$. The *minimum* min(U) of a nonempty multiset U is given by min$(U) = \min\{r \in \mathcal{R}: U(r) \neq 0\}$. The *maximum* max$(U)$ is defined analogously. If multiset U is nonempty, let $\rho(U)$ (the *range* of U) be the closed interval [min(U), max(U)], and let $\delta(U)$ (the *diameter* of U) be max(U) − min(U).

For the remainder of this paper, let $c = \lfloor (n - 1)/t \rfloor$. The constant c, which is the floor of the redundancy, plays a role in the definition of our averaging functions and, as we will see in Theorem 25, is the convergence rate of our approximate agreement protocol. Suppose that U is a multiset with $|U| = n - t$. Let $u_0 \leq u_1 \leq \cdots \leq u_{n-t-1}$ be the elements of U in nondecreasing order. Define select(U) to be the multiset consisting of the elements $u_0, u_t, \cdots, u_{(c-1) \cdot t}$. Thus, select$(U)$ chooses the smallest element of U and every tth element thereafter. The *median* of multiset U, written median(U), is defined to be u_m where $m = \lfloor |U|/2 \rfloor$. The *mean* of multiset U, written mean(U), is defined to be

$$\text{mean}(U) = \sum_{r \in \Re} \frac{r \cdot U(r)}{|U|}.$$

In our approximate agreement protocol, we will use the two averaging functions median(U) and mean(select(U)). The next three lemmas characterize the convergence properties of these functions.

Lemma 17: If U and V are multisets such that $V \subset U$, then median(V) $\in \rho(U)$ and mean(select(V)) $\in \rho(U)$.

Proof: This is immediate from the definition of the averaging functions. □

Lemma 18: If U, V, and W are multisets such that $|V| = |W| = n - t$, $V \subset U$, $W \subset U$, and $|U| \leq n$, then median(V) $\in \rho(W)$.

Proof: Let $v_0 \leq v_1 \leq \cdots \leq v_{n-t-1}$ be the elements of V, let $w_0 \leq w_1 \leq \cdots \leq w_{n-t-1}$ be the elements of W, and let $u_0 \leq u_1 \leq \cdots \leq u_{|U|-1}$ be the elements of U. We calculate that

$$\begin{aligned}
\text{median}(V) &\geq v_t && \text{Because } |V| \geq 2t+1.\\
&\geq u_t && \text{Because } V \subset U.\\
&\geq w_0 && \text{Because } W \subset U \text{ and } |U| - |W| \leq t.\\
&= \min(W).
\end{aligned}$$

Thus, median(V) $\geq$ min(W). By a similar argument, median(V) $\leq$ max(W). It follows that median(V) $\in \rho(W)$.

Lemma 19: If U, V, and W are multisets such that $|V| = |W| = n - t$, $V \subset U$, $W \subset U$, and $|U| \leq n$, then $|\text{mean(select}(V)) - \text{mean(select}(W))| \leq \delta(U)/c$.

Proof: Let $v_0 \leq v_1 \leq \cdots \leq v_{c-1}$ be the elements of select(V) and let $w_0 \leq w_1 \leq \cdots \leq w_{c-1}$ be the elements of select(W).

We claim that $\max(v_i, w_i) \leq \min(v_{i+1}, w_{i+1})$ for $0 \leq i \leq c - 2$. Let $u_0 \leq u_1 \leq \cdots \leq u_{|U|-1}$ be the elements of U. Observe that $v_i \leq u_{(i+1)\cdot t} \leq v_{i+1}$ because $V \subset U$ and because there are at most t elements of U that are not in V. Similarly, $w_i \leq u_{(i+1)\cdot t} \leq w_{i+1}$. Thus, $\max(v_i, w_i) \leq \min(v_{i+1}, w_{i+1})$ for $0 \leq i \leq c - 2$. This concludes the proof of the claim.

Let $x = |\text{mean(select}(V)) - \text{mean(select}(W))|$. We use the claim in the calculation that follows.

$$\begin{aligned}
x &= \left| \left(\frac{1}{c} \cdot \sum_{i=0}^{c-1} v_i\right) - \left(\frac{1}{c} \cdot \sum_{i=0}^{c-1} w_i\right) \right|\\
&= \frac{1}{c} \cdot \left| \sum_{i=0}^{c-1} (v_i - w_i) \right|\\
&\leq \frac{1}{c} \cdot \sum_{i=0}^{c-1} |v_i - w_i| \qquad \text{By the triangle inequality.}\\
&= \frac{1}{c} \cdot \sum_{i=0}^{c-1} (\max(v_i, w_i) - \min(v_i, w_i))\\
&= \frac{1}{c} \cdot \Bigg(\max(v_{c-1}, w_{c-1}) - \min(v_0, w_0)\\
&\qquad + \sum_{i=0}^{c-2} (\max(v_i, w_i) - \min(v_{i+1}, w_{i+1}))\Bigg)\\
&\leq (\max(v_{c-1}, w_{c-1}) - \min(v_0, w_0))/c \qquad \text{By the claim.}\\
&\leq (\max(U) - \min(U))/c\\
&\qquad \text{Because } V \subset U \text{ and } W \subset U.\\
&= \delta(U)/c.
\end{aligned}$$

□

B. The Protocol

Our approximate agreement protocol is given as Protocol 4. A processor begins the protocol by assigning its input value to the variable VAL. The protocol is organized into a series of asynchronous rounds. In each asynchronous round, each processor that is still operating broadcasts the value of VAL, waits to receive at least $n - t$ values broadcast in the current asynchronous round, places the multiset of these $n - t$ values in the variable SAMPLE, and applies an averaging function to SAMPLE to get a new value for VAL. In the first two asynchronous rounds, the averaging function used is median. In subsequent asynchronous rounds, it is mean ∘ select, where ∘ denotes function composition. In asynchronous round 2, each processor that is operating calculates an upper bound on the number of asynchronous rounds required and stores the bound in the variable ROUNDS. When sufficient asynchronous rounds have elapsed, a processor decides on the current value of VAL as its answer.

Protocol 4: An Approximate Agreement Protocol (Crash Faults, $n \geq 3t + 1$):

```
    Initialization for processor $p$:
        VAL ← the initial value of processor $p$
1.  for $r$ ← 1 to ∞ do
2.      broadcast ($r$, VAL)
3.      wait to receive ($r$, *) from $n - t$ processors
4.      let SAMPLE be the multiset of values received in the
            previous step
5.      if $r = 1$ then
6.          VAL ← median(SAMPLE)
7.      if $r = 2$ then
8.          VAL ← median(SAMPLE)
9.          ROUNDS ← $2 + \lceil \log_c(\max(1, \delta(\text{SAMPLE})/\epsilon)) \rceil$
10.         if $r$ = ROUNDS then decide VAL
11.     if $r \geq 3$ then
12.         VAL ← mean(select(SAMPLE))
13.         if $r$ = ROUNDS then decide VAL.
```

Some straightforward translation is necessary to put Protocol 4 in standard form. We omit the details.

C. Proof of Correctness

For all $r \geq 1$, we say that a processor p *finishes* asynchronous round r if it completes the last instruction in the code for asynchronous round r.

Lemma 20: In every execution of Protocol 4, for all r, all correct processors eventually finish asynchronous round r.

Proof: An easy induction on r. □

In an execution of Protocol 4, we let X_0 denote the multiset containing the inputs to all of the correct processors and we let

X_r denote the multiset containing the value of the variable VAL at the end of asynchronous round r for all processors that finish asynchronous round r. It follows from Lemma 20 that $|X_r| \geq n - t$ for all r.

The next three lemmas help establish the convergence of our approximate agreement protocol. In proving these lemmas, we use the properties of our two averaging functions that we proved in Section V-A.

Lemma 21: If $r \geq 1$, then $\rho(X_r) \subset \rho(X_{r-1})$.

Proof: There are two cases. Either $r = 1$ or $r \geq 2$.

Case 1: ($r = 1$): Let U be the multiset of inputs to all processors. Because there are n processors, $|U| = n$. Let W be the multiset of inputs to an arbitrarily chosen set of $n - t$ correct processors. Clearly, $W \subset X_0 \subset U$ and $|W| = n - t$. Let p be an arbitrary processor that finishes asynchronous round 1. Let V be the multiset of values received by processor p in asynchronous round 1. Because there are only crash faults, $V \subset U$. From step 2 of the code, $|V| = n - t$. We have established that the multisets U, V, and W satisfy the preconditions of Lemma 18; therefore, median$(V) \in \rho(W) \subset \rho(X_0)$. Because median$(V)$ is an arbitrarily chosen element of X_1, we have that $\rho(X_1) \subset \rho(X_0)$.

Case 2: ($r \geq 2$): Let $U = X_{r-1}$. Let p be an arbitrary processor that finishes asynchronous round r. Let V be the multiset of values received by processor p in asynchronous round r. Because there are only crash faults, $V \subset U$. If $r = 2$, then let $a =$ median (V); otherwise, let $a =$ mean(select(V)). We have established that the multisets U and V satisfy the preconditions of Lemma 17; therefore, $a \in \rho(X_{r-1})$. Because a is an arbitrarily chosen element of X_r, we have that $\rho(X_r) \subset \rho(X_{r-1})$. □

Lemma 22: If Y is the multiset of values received by an arbitrary correct processor in asynchronous round 2, then $\delta(X_2) \leq \delta(Y)$.

Proof: Let $U = X_1$. Because there are n processors, $|U| \leq n$. Let p be an arbitrary processor that finishes asynchronous round 2. Let V be the multiset of values received by processor p in asynchronous round 2. Let $W = Y$. Because there are only crash faults, $V \subset U$ and $W \subset U$. From step 2 of the code, $|V| = |W| = n - t$. We have established that the multisets U, V, and W satisfy the preconditions of Lemma 18; therefore, median$(V) \in \rho(W) = \rho(Y)$. Because median$(V)$ is an arbitrarily chosen element of X_2, we have that $\rho(X_2) \subset \rho(Y)$. It is immediate that $\delta(X_2) \leq \delta(Y)$. □

Lemma 23: If $r \geq 3$, then $\delta(X_r) \leq \delta(X_{r-1})/c$. (Recall that c is the floor of the redundancy.)

Proof: Let $U = X_{r-1}$. Because there are n processors, $|U| \leq n$. Let p and q be two arbitrary processors that finish asynchronous round r. Let V be the multiset of values received by processor p in asynchronous round r and let W be the multiset of values received by processor q in asynchronous round r. Because there are only crash faults, $V \subset U$ and $W \subset U$. From step 2 of the code, $|V| = |W| = n - t$. We have established that the multisets U, V, and W satisfy the preconditions of Lemma 19; therefore, $|$mean(select$(V)) -$ mean(select$(W))| \leq \delta(X_{r-1})/c$. Because mean(select$(V)$) and mean(select$(W)$) are arbitrarily chosen elements of X_r, we have that $\delta(X_r) \leq \delta(X_{r-1})/c$. □

Theorem 24: In the crash fault model, Protocol 4 solves the approximate agreement problem.

Proof: We show that the agreement, validity, and decision conditions are satisfied.

Agreement condition: Let p and p' be arbitrary correct processors. Suppose that processor p decides v in asynchronous round r and processor p' decides v' in asynchronous round r'. Without loss of generality, assume that $r \leq r'$. Let Y be the multiset of values received by processor p in asynchronous round 2.

We claim that $\delta(Y)/c^{i-2} \geq \delta(X_i)$ for all $i \geq 2$. The proof of the claim is by induction on i. The basis ($i = 2$) is immediate from Lemma 22. The inductive step is immediate from Lemma 23.

From steps 9, 10, and 13 of the code, $r = 2 + \lceil \log_c(\max(1, \delta(Y)/\epsilon)) \rceil$. It follows that $\epsilon \geq \delta(Y)/c^{r-2}$. By the claim, $\epsilon \geq \delta(X_r)$. Clearly, $v \in X_r$ and $v' \in X_{r'}$. By repeated application of Lemma 21, $v \in \rho(X_r)$. Thus, $|v - v'| \leq \delta(X_r) \leq \epsilon$.

Validity condition: If v is the decision of some correct processor, then there is some r such that $v \in X_r$. By repeated application of Lemma 21, $\rho(X_r) \subset \rho(X_0)$. Thus, $v \in \rho(X_0)$ and there are correct processors with inputs $\min(X_0)$ and $\max(X_0)$ such that $\min(X_0) \leq v \leq \max(X_0)$.

Decision condition: Let p be an arbitrary correct processor. Processor p assigns some value—an integer greater than or equal to 2—to the variable ROUNDS in asynchronous round 2; it never changes the variable ROUNDS after asynchronous round 2. Eventually, processor p calculates that $r =$ ROUNDS and it decides on some answer in either step 10 or step 13. Thus, each correct processor eventually decides. Because there are a finite number of correct processors, all correct processors eventually decide. □

We say that an approximate agreement protocol has *convergence rate l* if there is some constant k such that in every timed execution where the multiset of inputs to the correct processors is X, all correct processors decide by time $k + \lceil \log_l(\max(1, \delta(X)/\epsilon)) \rceil$.

Theorem 25: The convergence rate of Protocol 4 is c.

Proof: For all $r \geq 1$, it is easy to see that in any timed execution of Protocol 4, asynchronous round r ends by time r. Thus, it is sufficient to show that every correct processor decides by asynchronous round $2 + \lceil \log_c(\max(1, \delta(X_0)/\epsilon)) \rceil$.

Let p be an arbitrary correct processor. Let Y be the multiset of values received by processor p in asynchronous round 2. It is clear that $Y \subset X_0$. So, $\delta(Y) \leq \delta(X_0)$ and processor p assigns the value $2 + \lceil \log_c(\max(1, \delta(Y)/\epsilon)) \rceil$ to the variable ROUNDS. Thus, processor p decides by asynchronous round $2 + \lceil \log_c(\max(1, \delta(X_0)/\epsilon)) \rceil$. □

D. Approximate Agreement with Byzantine Faults

So far in this section, we have developed an approximate agreement protocol that tolerates crash faults. We now apply the two versions of the compiler developed in Section IV to produce approximate agreement protocols that tolerate Byzantine faults.

It is possible to express Protocol 4 in the standard form defined in Section IV-A. That is, there are A, V, $\mathcal{S}$, and $\mathfrak{D}$ such that Protocol 4 can be expressed as $\mathcal{C}(A, V, \mathcal{S}, \mathfrak{D})$. For

TABLE I
COMPARISON OF PERFORMANCE

Protocol	Convergence Rate	Minimum Redundancy
$\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$	$\sqrt{c}$	4
$\mathcal{B}'(A, V, \mathcal{S}, \mathcal{D})$	$\sqrt[3]{c}$	3
Dolev et al.	$\lfloor (c-1)/2 \rfloor$	5

TABLE II
CONVERGENCE RATE FOR SPECIFIC VALUES OF c

	$c = 3$	$c = 4$	$c = 5$	$c = 6$	$c = 7$	$c = 8$
$\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$	—	2	2.24	2.45	2.65	2.83
$\mathcal{B}'(A, V, \mathcal{S}, \mathcal{D})$	1.44	1.59	1.71	1.82	1.91	2
Dolev et al.	—	—	2	2	3	3

the remainder of this paper, we let A, V, $\mathcal{S}$, and $\mathcal{D}$ be chosen in that way.

Theorem 26: In the Byzantine fault model, for $c \geq 4$, protocol $\mathcal{B}(A, V, \mathcal{S}, \mathcal{D})$ solves the approximate agreement problem with a convergence rate of $\sqrt{c}$.

Proof: Correctness follows from Theorems 13 and 24. It follows from Theorems 14 and 25 that the convergence rate is $\sqrt{c}$. □

Theorem 27: In the Byzantine fault model, for $c \geq 3$, protocol $\mathcal{B}'(A, V, \mathcal{S}, \mathcal{D})$ solves the approximate agreement problem with a convergence rate of $\sqrt[3]{c}$.

Proof: Correctness follows from Theorems 15 and 24. It follows from Theorems 16 and 25 that the convergence rate is $\sqrt[3]{c}$. □

In Tables I and II, we compare the Dolev *et al.* protocol to the two compiled versions of our approximate agreement protocol. To compare the convergence rates, we need to overcome one obstacle. For our definition of convergence rate, the Dolev *et al.* protocol, as published, has no bounded convergence rate. The difficulty lies with the method that correct processors use to estimate the number of asynchronous rounds required until decision. Faulty processors can cause this estimate to be unboundedly pessimistic. This difficulty could easily be overcome if the Dolev *et al.* protocol were modified to use an estimation method similar to the one used in our Protocol 4. To allow for a fair comparison of convergence rates, we assume that the Dolev *et al.* protocol has been modified in this way.

In Table I, we compare the convergence rates and the minimum required redundancy. We can see that the asymptotic convergence rate of the Dolev *et al.* protocol is better than the asymptotic convergence rate of either compiled version of our protocol. Our protocols, however, operate with a smaller amount of redundancy.

In Table II, we give numerical values of the convergence rate for specific small values of c. The data in Table II show that, for any system with $n \leq 7t$, there is a compiled version of our approximate agreement protocol that has a better convergence rate than the Dolev *et al.* protocol.

ACKNOWLEDGMENT

I would like to thank C. Dwork, A. Fekete, N. Lynch, S. Toueg, W. Weihl, J. Welch, and the anonymous referees for many helpful comments on various versions of this paper.

REFERENCES

[1] G. Bracha, "Asynchronous Byzantine agreement protocols," *Inform. Computat.*, vol. 75, pp. 130-143, Nov. 1987.

[2] M. F. Bridgland and R. J. Watro, "Fault-tolerant decision making in totally asynchronous distributed systems," in *Proc. Sixth Annu. ACM Symp. Principles Distributed Comput.*, Aug. 1987, pp. 52-63.

[3] B. A. Coan, "Achieving consensus in fault-tolerant distributed computer systems: Protocols, lower bounds, and simulations," Ph.D. dissertation, Massachusetts Instit. of Technol., Apr. 1987.

[4] D. Dolev, C. Dwork, and L. Stockmeyer, "On the minimal synchronism needed for distributed consensus," *J. ACM*, vol. 34, pp. 77-97, Jan. 1987.

[5] D. Dolev, N. A. Lynch, S. S. Pinter, E. W. Stark, and W. E. Weihl, "Reaching approximate agreement in the presence of faults," *J. ACM*, vol. 33, pp. 499-516, July 1986.

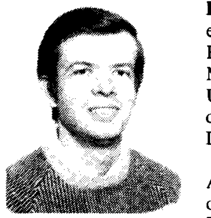
[6] A. D. Fekete, "Asymptotically optimal algorithms for approximate agreement," in *Proc. Fifth Annu. ACM Symp. Principles Distributed Comput.*, Aug. 1986, pp. 73-87.

[7] M. J. Fischer, N. A. Lynch, and M. Merritt, "Easy impossibility proofs for distributed consensus problems," *Distributed Comput.*, vol. 1, pp. 26-39, Jan. 1986.

[8] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *J. ACM*, vol. 32, pp. 374-382, Apr. 1985.

[9] S. R. Mahaney and F. B. Schneider, "Inexact agreement: Accuracy, precision, and graceful degradation," in *Proc. Fourth Annu. ACM Symp. Principles Distributed Comput.*, Aug. 1985, pp. 237-249.

[10] G. Neiger and S. Toueg, "Automatically increasing the fault-tolerance of distributed systems," in *Proc. Seventh Annu. ACM Symp. Principles Distributed Comput.*, Aug. 1988, pp. 248-262.



Brian A. Coan received the B.S.E. degree in electrical engineering and computer science from Princeton University, Princeton, NJ, in 1977, the M.S. degree in computer engineering from Stanford University, Stanford, CA, in 1979, and the Ph.D. degree in computer science from the Massachusetts Institute of Technology, Cambridge, MA, in 1987.

He has worked for Amdahl Corporation and AT&T Bell Laboratories. Currently he is a member of the Technical Staff at Bell Communications Research. His main research interests are in the theory of reliable distributed systems.