Deciding in Partitionable Networks*

Roy Friedman[†] Idit Keidar[‡] Dalia Malki[§] Ken Birman[†] Danny Dolev[‡]

November 27, 1995

Abstract

Motivated by Chandra and Toueg's work, we study decision protocols in a model that closely approximates "real" distributed systems. Our results show how the weakest failure detector and associated consensus algorithm can be adapted to a network in which omission failures can occur during periods when processes suspect one-another as faulty. For protocols in which a majority subset of the participants can reach decisions on behalf of the system as a whole, we also characterize a series of stages that necessarily arise during execution. Jointly, these findings establish a direct relationship between an extended version of the three-phase commit protocol, which makes progress even when a traditional three-phase commit would block, and the consensus protocol of Chandra and Toueg. Although we do not explore the linkage here, our results should also be applicable to other agreement protocols for systems of this sort, such as leader election and dynamic group membership.

*This work was supported by ARPA/ONR grants N00014-92-J-1866 and F30602-95-1-0047, and by grants from IBM, Siemens, and GTE Corporations. This paper is also published as Technical Report #TR95-16, Computer Science Institute, The Hebrew University of Jerusalem.

[†]Department of Computer Science, Cornell University. Email: {roy,ken}@cs.cornell.edu

[‡]Computer Science Institute, The Hebrew University of Jerusalem. Email: {idish,dolev}@cs.huji.ac.il

[§]AT&T Bell Laboratories, Murray Hill, NJ 07974.
Email: dalia@research.att.com

1 Introduction

The literature contains various forms of distributed agreement protocols with many common points and subtle differences. Examples are Consensus, agreement on membership changes in a group, and atomic commitment decisions. Chandra and Toueg, in a seminal paper [2], analyze the weakest conditions for reaching agreement in an asynchronous environment with crash failures. In the present paper, we consider the problem of reaching agreement decisions in an asynchronous environment with all possible benign failure types: crash and recovery, message omission failures, and network partitions and re-merges. We extend the definitions of failure detectors given in [3] to such environments. By these definitions, correct processes are alive and incorrect processes are crashed. Thus, there may be two processes considered correct that are unable to exchange messages because of communication problems. The first part of the paper concludes that agreement is achievable if the following holds:

1. Eventually, a majority of the processes become connected.
2. Every two permanently disconnected processes eventually suspect one another.
3. Eventually there exists a live process in a majority-connected component that is not suspected by any other process in the component.

Informally, the first condition states a majority-connected condition, and the second and third conditions characterize an *eventually strong failure detector*[1]. These conditions are defined formally in the paper. The eventually strong failure detector is analogous to $\diamond\mathcal{S}$ in [3] but is defined for the extended failure model. Similar but stronger conditions for reaching agreement on the membership of a group processes are given in the *timed asynchronous* model of Cristian and Schmuck [4].

In our terminology, protocols that achieve agreement in such an environment are called majority-deciding protocols (more precisely, $\lceil\frac{n+1}{2}\rceil$-deciding protocols, where the system contains n processes). For example, the consensus protocol of Chandra and Toueg [3] is in this category, since it reaches consensus among a majority in a crash-failure environment. We also provide an adaptation of their protocol for an omission-failure environment that is a majority-deciding protocol as well. Another protocol is the enhanced three phase commit protocol (E3PC) of Keidar and Dolev [7], which solves the atomic commitment problem in any majority-connected component. Although we do not pursue the linkage here, our findings are also relevant to protocols for primary-partition dynamic group membership problems in the asynchronous model, and in the timed asynchronous model of Cristian and Schmuck [4] (we refer to the *majority group membership* problem – the strongest they define – which guarantees total order of majority group views).

The second part of the paper characterizes all such majority-deciding protocols, showing that:

[1] We actually show that an *eventually weak failure detector* is sufficient. Such a failure detector is defined formally in the paper.

1. Before reaching a decision, a protocol must reach a configuration *close-to-deciding* a value (say v). Informally, this means that a minority of processes may extend this configuration to a v-deciding configuration without any additional interaction with other processes. This minority is called a *potential set.*

2. Furthermore, in the configuration above, the system is *far-away from deciding* $u \neq v$. Informally, this means that after this configuration is reached, the potential minority set needs to be "consulted" by a majority in order to decide $u \neq v$. The intuition here is that in order to reverse the potential minority decision v, a majority is required to verify that at least one of the potential sets has not taken its "crucial" last step.

   Note that most protocols do not start in a configuration as above, *i.e.* close to a decision and far away from another decision value, unless the initial configuration is already univalent, and therefore need to bring the system into such a configuration. In case the initial configuration is already univalent, e.g., a unilateral abort in the 3PC protocol, the initial configuration is close to deciding (in this example, abort), but is far from deciding other values, since there are no extensions which can lead to a decision on another value.

   Finally, there may be protocols that start in a configuration which is not univalent, but is close to deciding on some value v and far-away from deciding u. These protocols would have the peculiar property that at the outset, some minority set S would be enabled to decide (on v), and hence that the remainder of the system would be prevented from deciding any other value ($\neq v$) without involving processes in S. None of the protocols we study is initially configured this way.

3. If during the execution two system configurations were reached, one close to deciding v and far away from deciding u, and the other vice-versa, then one of these configurations "dominates" the other, in the following sense:

   - In one of the configurations, a majority of processes will have taken steps subsequent to the other configuration. In other words, a majority must participate in any protocol that shifts from one such configuration to the other.
   - A system configuration reachable from both configurations (their union) has the properties of the dominant one, *i.e.* is close to deciding the latter value, and far-away from the other.

To summarize, these findings establish a direct relationship between an extended version of the three-phase commit protocol, which makes progress even when a traditional three-phase commit would block, and the consensus protocol of Chandra and Toueg.

2 The System Model and Definitions

The system contains a fixed set N of n processes. The set N is known to all the processes. Processes communicate via message passing. The underlying communication network provides

FIFO datagram message delivery. That is, if two messages are sent by the same process to the same process, and both are delivered, then they are delivered in the same order in which they were sent. However, the network does not guarantee that all messages will be delivered. Also, there is no known bound on message transmission time, hence the system is asynchronous.

The datagram message delivery and the asynchrony assumption capture the following types of possible failures: Messages may be lost, failures may partition the network into several *components*[2], and previously disjoint network components may re-merge. Processes may crash and recover, but recovered processes come up with their stable storage intact and every state change in a processor is logged on stable storage. Hence, our model does not distinguish processes that crash and later recover from "disconnected" ones, and we therefore omit this case from our discussion.

We assume also that each process is equipped with a failure detector, which provides hints regarding which processes may be disconnected at any given time. The information provided by the failure detectors need not be accurate, although some restrictions on their behavior is imposed later in Section 4.1.

More formally, a process can be modeled as a (possibly infinite) automata, which take *steps* that consist of receiving `message-receive` events from the network and `suspect` events from the failure detector, do some local calculation, and then generate zero or more `message-send` events. Events of type `message-receive` may be empty, containing null messages (thus, processes may initiate operations spontaneously). A process can also receive a `crash` event, which is the last event it receives.

A *history* of a process is a sequence of events as they occur in that process, in which `crash` events are not followed by any other event. An *execution* is a collection of histories, one for each process, in which there is a mapping from `message-receive` events to `message-send` events.

An execution σ' is a *sub-execution* of another execution σ if they include histories of the same set of processes, and the history of each process p_i in σ' is a prefix of p_i's history in σ. An execution σ *extends* an execution σ' if σ' is a sub-execution of σ. Given an execution σ and a sub-execution σ' of σ, the history postfixes resulted by eliminating the history of each process in σ' from its corresponding history in σ is an *extension* of σ'. We denote this extension by $\sigma \setminus \sigma'$.

2.1 K-Connected Executions

Let σ be an infinite execution, σ' a sub-execution, σ'' a sub-execution of σ', and let τ be the extension $\sigma' \setminus \sigma''$. We use the following definitions:

alive Process p is *alive* in τ if p does not crash in σ'.

[2]A **component** is sometimes called a **partition**. In our terminology, a partition splits the network into several components.

crashed Process p is (permanently) *crashed* in τ if p crashes in σ'.

connected Processes p and q are *connected* in τ if p and q are alive in τ, p receives in σ every message that was sent from q to p in τ, and vice versa. A set of processes P is *connected* in τ, if for every two processes $p, q \in P$, p and q are connected in τ.

If τ is infinite, p and q are (P is) *permanently connected.*

detached Processes p and q are *detached* in τ if p does not receive (in σ) any message that q sends in τ and vice versa. A set of processes P is *detached* from a set of processes Q in τ if for every process $p \in P$ and every process $q \in Q$, p and q are detached in τ.

If τ is infinite, p and q (P and Q) are *permanently detached.*

connected component A set of processes P is a *connected component* in τ, if P is *connected* in τ, and P is detached from $N \setminus P$ in τ. If P has k members, it is also called a *k-connected component* in τ.

If τ is infinite, P is a *permanently connected component.*

k-connected execution If there exists a *k-connected component* in τ, and τ is infinite, then the execution σ' is *k-connected.*

3 k-decidable problems

Our work analyzes the executions of protocols that reach agreement among a group of processes. Formally, we say that a decision protocol is a *k-decidable* protocol if:

Agreement All the processes that decide decide on the same value.

Non-Triviality The protocol has different executions that decide on at least two different values.

k-decidability In every *k-connected* execution of the protocol some member of the *k-connected component* decides.

The Non-Triviality requirement above is further refined in specific problem models. For example, the requirements of an atomic commitment protocol are: The COMMIT decision can only be reached if all sites voted **Yes**, and if there are no failures and no suspicions and all sites voted **Yes**, then the decision will be to COMMIT [6]. The requirement of an agreement protocol is that it decides on the initial value of one of the processes.

Examples of $\lceil \frac{n+1}{2} \rceil$-decidable protocols are the agreement protocol of Chandra and Toueg [3] and the atomic commitment protocol, E3PC, of Keidar and Dolev [7]. These protocols use failure detectors: E3PC uses an *eventually perfect* failure detector, and Chandra and Toueg's protocol uses an *eventually weak* failure detector, these failure detectors are defined in Section 4.

Note that as required by the FLP result [5], there are executions of the above protocols which may never terminate in a real system. However, in these executions, some important messages must always be too late, or lost. The $\lceil\frac{n+1}{2}\rceil$-connected requirement characterizes an important class of executions in which this does not happen, and therefore all $\lceil\frac{n+1}{2}\rceil$-connected executions of both [3] and [7] terminate.

4 Failure Detectors

In Section 4.1 we define failure detectors in the model where messages may be lost and the network may partition. In Section 4.2 we show that $\diamond\mathcal{W}$ is reducible to $\diamond\mathcal{S}$ in our model. In Section 4.3 we show that the protocol in [3] is $\lceil\frac{n+1}{2}\rceil$-decidable with *unbounded* buffer space. In Section 4.4 we present a $\lceil\frac{n+1}{2}\rceil$-decidable protocol that requires *bounded* buffer space, and uses an eventually weak failure detector.

Chandra, Hadzilacos, and Toueg showed that $\diamond\mathcal{W}$ is the weakest failure detector to achieve consensus in an asynchronous environment with crash failures [2]. Since our model is an extension of their model, $\diamond\mathcal{W}$ is also the weakest failure detector required for $\lceil\frac{n+1}{2}\rceil$-decidable algorithms in our model.

4.1 Classes of Failure Detectors

Chandra and Toueg [3] define classes of *distributed* failure detectors: Each process has access to a local *failure detector module*; each local module maintains a list of the processes that it currently suspects to have crashed. Each process receives a `suspect` event from the local failure detector module when the list of suspected processes changes. These failure detector classes are defined in a crash-failure asynchronous environment.

If we were to use the definitions of Chandra and Toueg in an environment where messages can get lost, a message loss would render faulty either the sender of the message or the receiver (or both). In practical settings, this would render almost all the processes faulty, violating the $\lfloor\frac{n}{2}\rfloor$ resilience bound for consensus. Therefore, we extend the definitions of failure detectors to include omission failures. In the partitionable model, two processes may be detached and yet both are considered correct.

Failure detectors can be characterized using *completeness* and *accuracy* properties. Below, we provide the definitions of *completeness* and *eventual accuracy*[3] in our model. Notice the following points in our failure detectors: A failure detector suspects *detached* processes as well as *faulty* (or *crashed*) processes. Further, some non-faulty process *in every connected component* is not suspected by one or many other processes in its component. We use the following definitions:

[3] Chandra and Toueg also define *perpetual accuracy*, but this concept doesn't carry to our model because of the transient nature of failures.

Strong Completeness Eventually every process that permanently crashes or permanently detaches from p is suspected by p.

Weak Completeness Eventually every process that permanently crashes or permanently detaches from p is suspected by some process q that is permanently connected to p.

Eventual Strong Accuracy If p and q are permanently connected, then there is a time after which p does not suspect q.

Eventual Weak Accuracy In every permanently connected component P, there is a time after which some member of P is never suspected by any other member of P.

Completeness	**Accuracy**	
	Eventual Strong	Eventual Weak
Strong	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
Weak	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

Figure 1: Classes of Failure Detectors

As in [3], we define four classes of eventual failure detectors; each failure detector is characterized by the strength of its completeness and eventual accuracy properties. The failure detector classes are shown in Figure 1.

4.2 Reducing $\diamond\mathcal{W}$ to $\diamond\mathcal{S}$

We reduce $\diamond\mathcal{W}$ to $\diamond\mathcal{S}$ using the algorithm suggested by Chandra and Toueg [3], described in Figure 2. We prove that this algorithm reduces $\diamond\mathcal{W}$ to $\diamond\mathcal{S}$ in our model as well. We denote the given (input) failure detector by W, and the resulting (output) failure detector by S. We denote by W_p the suspects list of p with the failure detector W, and S_p the suspects list with S.

The algorithm works as follows: p periodically sends its input suspects list to all the processes. When p receives a suspects list from another process q, it removes q from its suspects list, and merges q's suspects list into its output suspects list.

Lemma 4.1 *If* W *satisfies* weak completeness *then* S *satisfies* strong completeness.

Proof: If q permanently crashes or permanently detaches from p, then eventually $q \in W_r$ for some r s.t. r is permanently connected to p. r periodically sends W_r to p. Since p and r are permanently connected, p receives r's message and merges W_r into S_p, and thus, eventually, $q \in S_p$. p doesn't hear from q, and therefore doesn't remove it from its suspects list. ■

Every process p does the following:

1. Initially:
   $S_p \leftarrow \emptyset$

2. p periodically sends its suspects list to all the processes:
   `send` (p, W_p) to all.

3. **when** `receive` (q, W_q) for some q,
   $S_p \leftarrow S_p \cup W_q \setminus \{q\}$

Figure 2: From $W \in \diamond\mathcal{W}$ to $S \in \diamond\mathcal{S}$

Lemma 4.2 *If W satisfies* eventual weak accuracy *then S also satisfies* eventual weak accuracy.

Proof: If there is a time after which p is never suspected by any member of its connected component P, then after this time no member of P ever sends a suspects list containing p. Furthermore, the members of P do not receive messages sent by processes that are not members of P, and therefore, there is a time after which p is never added to their suspects list. If at this time $p \in S_q$ for some $q \in P$, then since p and q are connected, eventually q gets p's suspects list and removes p from S_q. p is never again added to S_q. ∎

The theorem follows:

Theorem 4.3 *The algorithm in Figure 2 reduces $W \in \diamond\mathcal{W}$ to $S \in \diamond\mathcal{S}$.*

4.3 Agreement with Unbounded Buffers and $\diamond\mathcal{S}$

Chandra and Toueg [3] suggested an algorithm that solves consensus with a failure detector $S \in \diamond\mathcal{S}$ if there are no more than $\lfloor \frac{n}{2} \rfloor$ failures. If there are more than $\lfloor \frac{n}{2} \rfloor$ failures, the algorithm "blocks", but in no case does it allow two processes to decide on different values.

If we allow our processes to be infinite state machines, the processes can use unbounded buffer space. In this case, each process can record all the messages it sent in the past and piggyback on every message all the previous messages. Therefore, we can assume that there are no message losses between pairs of processes as long as they are not permanently detached. Furthermore, p cannot distinguish the case that it is permanently detached from q, from the case that q has crashed. Therefore, to the members of an $\lceil \frac{n+1}{2} \rceil$-permanently connected component in this model the execution seems the same as to a group of $\lceil \frac{n+1}{2} \rceil$ correct processes in the crash-recovery model.

Thus, the algorithm in [3] **unchanged** will reach a decision among members of the $\lceil \frac{n+1}{2} \rceil$-permanently connected component P. Processes that are not members of P will never reach contradicting decisions. So, the algorithm of [3] is $\lceil\frac{n+1}{2}\rceil$-decidable in the partitionable model using unbounded buffers.

4.4 Agreement with Bounded Buffers and $\diamond\mathcal{S}$

We suggest a $\lceil\frac{n+1}{2}\rceil$-decidable algorithm that uses a failure detector $S \in \diamond\mathcal{S}$. The algorithm is based on the algorithm suggested in [3], and is presented in Figure 3.

Every process p executes the following:

Initially: $estimate_p \leftarrow v_p$; $r_p \leftarrow 0$; $ts_p \leftarrow 0$

|| **while** undecided **do**
1. $r_p \leftarrow r_p + 1$; $c_p \leftarrow (r_p \bmod n) + 1$
2. **if** $(c_p = p)$ **then**
   keep sending $(p, r_p, new\_coordinator)$ periodically to all, until r_p changes
3. send $(p, r_p, estimate_p, ts_p)$ to c_p
4. **if** $(c_p = p)$ **then**
   wait until receive $(q, r_p, estimate_q, ts_q)$ from $\lceil\frac{n+1}{2}\rceil$ processes
   $ts_p \leftarrow \max ts_q$; $estimate_p \leftarrow estimate_q$ s.t. $ts_q = ts_p$
   send $(p, r_p, estimate_p)$ to all
5. **wait until** either:
   if receive $(c_p, r_p, estimate_{c_p})$ from c_p **then**
      $ts_p \leftarrow r_p$; $estimate_p \leftarrow estimate_{c_p}$
      send (p, r_p, ack) to c_p
   if suspect c_p **then**
      send $(p, r_p, nack)$ to c_p
6. **if** $(c_p = p)$ **then**
   wait until receive (q, r_p, ack) or $(q, r_p, nack)$ from $\lceil\frac{n+1}{2}\rceil$ processes
   if received (q, r_p, ack) from $\lceil\frac{n+1}{2}\rceil$ **then**
      send $(p, r_p, estimate_p, decide)$ to all

|| **when** receiving $(q, r_q, new\_coordinator)$ with $r_q > r_p$:
   $r_p \leftarrow r_q$; $c_p \leftarrow q$
   Abort the current round of **while** loop and go to step 2 above

|| **when** receiving $(q, r_q, estimate_q,$ decide$)$: decide on $estimate_q$

Figure 3: Deciding with $\diamond\mathcal{S}$ and Bounded Buffers

The algorithm uses the *rotating coordinator* paradigm. The coordinator of round r is process $(r \bmod n) + 1$. Every coordinator tries to determine a consistent decision value.

Each process maintains three variables: the round number, the current estimate, and a timestamp which is the round number when this estimate was last changed. Initially, the estimate contains the process' initial value. Each round of the algorithm is conducted in four phases. In the first phase, all the processes send timestamped estimates to the coordinator. In the second phase, the coordinator gathers $\lceil\frac{n+1}{2}\rceil$ estimates, selects one of the estimates with the latest timestamp, and multicasts it to all the processes. In the third phase, all the processes wait for the new estimate from the coordinator or until they suspect the coordinator. When a process receives the new estimate it adopts it as its new estimate, and sends an ACK to the coordinator. If a process suspects the coordinator it sends a NACK to it, and moves to the next round. In the fourth phase, the coordinator waits for $\lceil\frac{n+1}{2}\rceil$ replies from processes. If it gathers $\lceil\frac{n+1}{2}\rceil$ ACKs, it sends a "decide" message.

When a process is the coordinator, it periodically sends a *new_coordinator* message telling all the other processes that it is the new coordinator. When a process receives such a message with a higher round number than its own, it "aborts" the current round and "obeys" the new coordinator. Thus, even if there are message losses before the connected component stabilizes, which cause different processes to be "stuck" in different rounds, when the processes become permanently connected, they all shift to the latest round.

If the coordinator of some round is a member of an $\lceil\frac{n+1}{2}\rceil$-connected component P, and is not suspected by any member of P, then it does not abort the round and will succeed in reaching a decision. Thus, the algorithm is $\lceil\frac{n+1}{2}\rceil$-decidable using $S \in \diamond\mathcal{S}$.

5 Allowing a Majority to Decide

In this section, we consider protocols in which any majority subset of the system members is permitted to reach decisions on behalf of the system as a whole. We characterize a series of stages that such protocols must pass through before reaching a decision. Intuitively, these stages capture the notion that to safely reach a decision, a process must first obtain "permission" from a majority of the processes in the system. Some further motivation for the characterization of these steps as we define them in this section appear in Appendix A.

The results described in this section apply to $\lceil\frac{n+1}{2}\rceil$-decidable protocols, and are valid for any type of failure detector, including $\diamond\mathcal{P}$.

5.1 Consistent Cuts

In order to reason about executions, we need to introduce the following formal definitions: The *local state* of a process at the end of a history, or at the end of a history prefix, is the state of the automata at that point. Given an execution σ, a *consistent cut* is a collection of the local states of all processes at the end of some sub-execution σ' of σ. An execution σ which extends execution σ' also *extends* the consistent cut C' which corresponds to σ'. If the consistent cut that corresponds to the end of σ is C, the extension $\sigma \setminus \sigma'$ is also denoted $C \setminus C'$.

Let σ be an execution whose corresponding cut is C, σ' a sub-execution of σ whose corresponding cut is C', and σ'' a sub-execution of σ' whose corresponding cut is C'', and denote $\sigma \setminus \sigma'$ by τ and denote $\sigma' \setminus \sigma''$ by τ'. Then we sometimes use the notation $C'\tau$ instead of C, $C''\tau'$ instead of C', or even $C''\tau'\tau$ instead of C. Denote $\tau'\tau$ by τ''; in this case, τ' is said to be a *sub-extension* of τ''. Also, if s is a receive event for some process p_i in C, then we denote by $C - s$ the consistent cut the corresponds to the sub-execution of σ achieved by eliminating s from p_i's history in σ. (If $C - s$ results in a non consistent cut then it is undefined.)

We say that a message is sent causally after a consistent cut C if the corresponding `message-send` event does not appear in the sub-execution that corresponds to C. An extension τ of a consistent cut C is *p_k-silent* if no message that was sent by p_k causally after C is ever received during τ. An extension τ of a consistent cut C is *p_k-free* if p_k does not receive any events during τ. An extension τ of a consistent cut C is a *p_k-only extension* if only p_k receive events during τ. The definitions of p_k-silent, p_k-free, and p_k-only extensions can be extended to sets of processes in a natural way.

Let σ and σ' be two sub-executions of some infinite execution, and let C and C' be the corresponding consistent cuts, we denote by $greater(C, C')$ the set of processes whose histories in C are longer than in C'; we denote by $union(C, C')$ the consistent cut which is formed by taking the history prefixes of $greater(C, C')$ from C and by taking the history prefixes of $greater(C', C)$ from C'.

A cut C is *v-valent* if there is an extension of C in which at least one process decides, and if in every extension of C in which at least one process decides, this process decides v. We say that an extension of a cut C is *v-valent* if it leads to another cut C' which is v-valent. A consistent cut is *multivalent* if there exist two extensions τ_v and τ_u of C such that τ_v is v-valent and τ_u is u-valent, for some $u \neq v$.

Lemma 5.1 *Let C be a multivalent cut, S be a set of processes, τ be a S-free extension of C, and τ' be a S-only extension of C. Then both $\tau\tau'$ and $\tau'\tau$ are possible extensions of C and $C\tau\tau' = C\tau'\tau$.*

5.2 Characterizing deciding executions

Definition 5.1 *We say that a consistent cut C is* close to deciding v *if there exists a v-valent extension τ of C in which at most $\lfloor \frac{n}{2} \rfloor$ processes receive events. The set of process that receive events during τ is called a* potential set for deciding v in C, *and we denote this set by S_τ.*

Definition 5.2 *We say that C is* far away from deciding on u *if it is close to deciding $v \neq u$, and for one of the v-valent extensions τ of C in which at most $\lfloor \frac{n}{2} \rfloor$ processes receive events, the following holds: In any u-valent extension τ' of C, $\lfloor \frac{n}{2} \rfloor + 1$ processes receive events that causally follow events by processes in S_τ during τ'.*

Note that in particular, if a consistent cut is far away from deciding on some value u, then it is not close to deciding u.

Definition 5.3 (Decider) *A process p_i is called a* decider *in a consistent cut C if there exist two p_i only schedules τ_v and τ_u such that $C\tau_v$ is v-valent and $C\tau_u$ is u-valent.*

Claim 5.2 *If there exists a decider in some consistent cut C, then the protocol is not k-decidable for any $k < n$.*

The proof of Claim 5.2 is almost identical to the proof of Lemma 12.1.7 in [1]. We repeat the details for completeness.

Proof: Assume, by way of contradiction, that there exists a decider p_i in some consistent cut C of a k-decidable protocol $\mathcal{A}$, for some $k < n$. Since the protocol is k-decidable and $k < n$, there exists a p_i-free v-valent extension τ of C, for some value v. Since p_i is assumed to be a decider in C, there exists a value $u \neq v$ and a p_i-only u-valent extension τ' of C. By Lemma 5.1, since τ is p_i-free and τ' is p_i-only, $C\tau\tau' = C\tau'\tau$. However, since $C\tau$ is v-valent, $C\tau\tau'$ is also v-valent. Similarly, since $C\tau'$ is u-valent, $C\tau'\tau$ is also u-valent. A contradiction. ■

Note that it is possible that the initial configuration of a specific execution of a k-decidable protocol is close to two different values. Our results indicate that before the system can move to a univalent configuration, and in particular, before anyone can decide, the system must be in a configuration which is close to the value being chosen, and far away from all other possible values.

Theorem 5.3 *Let C be a v-valent consistent cut of an execution of a $\lceil\frac{n+1}{2}\rceil$-decidable protocol, for some value v, and let e_i be the last event received by some process p_i in C such that $C - e_i$ is multivalent. Then $C - e_i$ is close to deciding v, far away from deciding on any other value u.*

In the following proof, we denote the cut $C - e_i$ by C_{e_i}.

Proof: In order to prove that C_{e_i} is close to deciding v we need to show a v-valent extension of C in which at most $\lfloor\frac{n}{2}\rfloor$ processes receive events. The event e_i of p_i is an example of such an extension.

Now, assume, by way of contradiction, that there exists a u-valent extension τ of C is which less than $\lfloor\frac{n}{2}\rfloor$ processes *learn* about a new message from p_i. Here, a process learns about a message if it sends that message, or if it executes an event (causally) after the message was sent. Denote the set of processes that learn about a new message from p_i during τ by S. Since

τ is u-valent, it cannot be a p_i-free extension of C_{e_i}. Also, if no message is sent from p_i and received by another process, then there exists a cut in which p_i is a decider and by Claim 5.2 the protocol is not $\lceil \frac{n+1}{2} \rceil$-decidable.

Hence, there exists a p_i-free sub-extension τ' of τ such that $\tau = \tau'\tau''$, the histories of all processes not in S are the same in τ and in τ' and $C\tau' = C_{e_i}\tau' e_i$ is v-valent. Note that τ' is non-empty, since processes not in S cannot receive any message from a process that already received a message from p_i. Note also that there is no assumption about the number of processes whose histories are prolonged by τ. The only assumption is about the size of S. Denote the consistent cut that corresponds to $C_{e_i}\tau'$ by C'.

Recall that $\mid S \mid \leq \lfloor \frac{n}{2} \rfloor$ meaning that at least $\lfloor \frac{n}{2} \rfloor + 1$ do not learn about any new message by p_i during τ. Hence, there exists a univalent $S \cup \{p_i\}$-free extension τ''' of C'. Moreover, since τ''' is p_i-free, it must also be v-valent. However, no message is ever exchanged between processes in $S \cup \{p_i\}$ and processes not in $S \cup \{p_i\}$, both in τ'' and in τ'''. Hence, $C'\tau''\tau'''$ is a consistent cut. However, $C'\tau''\tau'''$ is both u-valent and v-valent. A contradiction. ∎

Theorem 5.4 *Let C_i and C_j be consistent cuts of an execution σ of a $\lceil \frac{n+1}{2} \rceil$-decidable protocol such that C_i is close to deciding v for some value v, and far away from deciding u for any other value u, and that $greater(C_i, C_j)$ includes a potential set for deciding v in C_i. Then the following holds:*

1. *$union(C_i, C_j)$ is close to deciding v,*
2. *$union(C_i, C_j)$ is far away from deciding on any other value u, and*
3. *$greater(C_i, C_j)$ includes a potential set for deciding v in $union(C_i, C_j)$.*
4. *If C_j was close to deciding u and far from deciding v, then $\mid greater(C_i, C_j) \mid > \lfloor \frac{n}{2} \rfloor$.*

Proof: For the rest of this proof, denote the potential set which is included in $greater(C_i, C_j)$ by S (if there is more than one such set, let S be one of them), denote the extension $union(C_i, C_j) \setminus C_i$ by τ_{ji}, and the extension $union(C_i, C_j) \setminus C_j$ by τ_{ij}.

Claim 1: By assumption, S is a subset of $greater(C_i, C_j)$. Hence, there exists a v-valent extension τ of C_i in which at most $\lfloor \frac{n}{2} \rfloor$ processes, all of them from S, receive events. Also, during τ_{ji} and during τ, no message is exchanged between any processes in S and processes not in S. By Lemma 5.1, τ is also a v-valent extension of $C_i\tau_{ji}$, or in other words of $union(C_i, C_j)$, in which at most $\lfloor \frac{n}{2} \rfloor$ processes receive events. Therefore, $union(C_i, C_j)$ is close to deciding v.

Claim 2: Note that for every extension τ of $union(C_i, C_j)$, $\tau_{ji}\tau$ is also an extension of C_i. In particular, the histories of all processes in $greater(C_i, C_j)$ are the same in $union(C_i, C_j)$ and in C_i. Thus, if there exists a u-valent extension τ of $union(C_i, C_j)$ in which less than $\lfloor \frac{n}{2} \rfloor$ processes learn about a new message of at least one process in S, then $\tau_{ji}\tau$ is also a u-valent extension of C_i in which less than $\lfloor \frac{n}{2} \rfloor$ processes learn about a new message of at least one process in S. Therefore, $union(C_i, C_j)$ is far away from deciding on any other value u.

Claim 3: Since the histories of all processes in S are the same in C_i and $union(C_i, C_j)$, S is a potential set for $union(C_i, C_j)$ as well. Hence, $greater(C_i, C_j)$ includes a potential set for deciding v in $union(C_i, C_j)$.

Claim 4: Assume by way of contradiction, that $\mid greater(C_i, C_j) \mid \leq \lfloor \frac{n}{2} \rfloor$. Also, by assumption, there exists a v-valent S-only extension τ of C_i. Thus, $\tau_{ij}\tau$ is also a v-valent $greater(C_i, C_j)$-only extension of C_j. However, since we assumed that $\mid greater(C_i, C_j) \mid \leq \lfloor \frac{n}{2} \rfloor$, then in this extension less than $\lfloor \frac{n}{2} \rfloor$ can learn about a new message by some process in any potential set for deciding u in C_j. A contradiction to the assumption that C_j is far away from deciding v. ■

6 Discussion

Our results establish two forms of linkage between what was previously known for the asynchronous computing model, and the state of practice in the development of distributed software for realistic environments. The results of the first part of this paper demonstrate that the agreement protocol of Chandra and Toueg can be adapted to an environment in which messages are lost during periods of mutual suspicion by the endpoints, which is an important contribution. The only modifications to the Chandra and Toueg protocol are due to the fact that the traditional asynchronous model assumes that every message sent between correct processors will eventually be delivered. This assumption of fully reliable delivery has long been recognized as unrealistic, since in practice it requires infinite buffering capacity. Here, we have shown how to build a weakest failure detector in an environment where message reliability matches the properties of typical real networks, and have shown that in such a setting, the intuitive findings of Chandra and Toueg remain valid.

The second part of our paper can be understood as a study of the "knowledge states" through which a protocol must pass as it moves to a decision configuration, or moves from one decision configuration to another. The significance of this characterization is that it applies to a great variety of decision protocols, including agreement but also including multi-phase commit protocols, group membership protocols, and others of a similar nature. To the degree that these protocols permit a majority of processes to force the system to a decision, our work establishes that such a majority must explicitly participate in the decision procedure, and that if the system was previously close to some other decision, must be able to order the decisions. Interestingly, the agreement protocol of Chandra and Toueg [3] and the Enhanced 3-phase commit protocol of Keidar and Dolev [7] exhibit precisely the structure needed to implement this policy.

These things said, it should also be commented that our results contain few surprises. The weakest failure detector for enabling agreement turns out to need few changes for use in our environment, and the agreement protocol itself is nearly identical to the one for a more classical asynchronous environment. The majority progress findings confirm that a structure commonly found in protocols for realistic settings is a necessary one, and that the resemblance of such protocols to the older quorum-style database protocols and multi-phase commit protocols is more than accidental.

References

[1] H. Attiya. *Lecture Notes for Course #236357: Distributed Algorithms.* Department for Computer Science, The Technion, January 1994.

[2] T. D. Chandra, V. Hadzilacos, and S. Toueg. The Weakest Failure Detector for Solving Consensus. In *ACM Symp. on Prin. of Distributed Computing (PODC)*, pages 147–158, 1992.

[3] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. *Journal of the ACM.* To appear, previous version in PODC 1991 pp. 325-340.

[4] F. Cristian and F. Schmuck. Agreeing on Process Group Membership in Asynchronous Distributed Systems. Technical Report CSE95-428, Department of Conmputer Science and Engineering, University of California, San Diego, 1995.

[5] M. Fischer, N. Lynch, and M. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *J. ACM*, 32:374–382, April 1985.

[6] R. Guerraoui. Revisiting the Relationship between non-blocking Atomic Commitment and Consensus. In *International Workshop on Distributed Algorithms (WDAG)*, pages 87–100, September 1995.

[7] I. Keidar and D Dolev. Increasing the Resilience of Atomic Commit, at No Additional Cost. In *ACM Symp. on Prin. of Database Systems (PODS)*, pages 245–254, May 1995. Previous version available as Technical Report CS94-18, The Hebrew University, Jerusalem, Isreal.

[8] D. Skeen. A Quorum-Based Commit Protocol. Technical Report TR82-483, Department of Computer Science, Cornell University, February 1982.

A Motivation of State Definitions

In this section, we provide some intuition on the definitions given in Section 5 and relate them to the agreement protocol of Chandra and Toueg [3] and the E3PC protocol of Keidar and Dolev [7]. Note, however, that this section is intended to give some intuition, and not to cover all possible cases.

Both the E3PC and the Chandra and Toueg protocols consist of rounds, each of which, if all goes well, requires 3 phases. In the first phase, the coordinator solicits initial votes from the members; In the second phase, the coordinator suggests a possible decision value to the members, and waits to hear if a majority of these members agree; finally, in the third phase, after receiving approvals from a majority of the members for the value that was suggested, the coordinator can decide, and distribute its decision. Our characterization establishes that such a structure is required in order to bring the system to a configuration in which it is close to deciding on one value and far from deciding on any other value.

Note that due to the structure of these protocols, if $\lfloor \frac{n}{2} \rfloor$ processes (other than the coordinator) receive the coordinator's suggestion in any round, then the system is already in a univalent state. Thus, after sending the suggestion, say v, the coordinator's causal cut (the cut of all events that precede the send event) is close to deciding v. Similarly, the causal cut of any member that just received the coordinator's suggestion is also close to deciding v, since if less than $\lfloor \frac{n}{2} \rfloor$ processes will also receive this suggestion, the system's state will become univalent.

On the other hand, suppose that the current round is abandoned. The three phase structure guarantees that the new coordinator has to wait for the votes of at least $\lfloor \frac{n}{2} \rfloor$ processes other then itself before it can make a new proposition. For a future round to become close to a different value u, the coordinator has to receive the vote of at least one member in every minority that was able to bring the system to a v-valent state in the round that was abandoned. Hence, the causal cut of a coordinator that sends its suggestion, as well as the causal cut of any member that just received a suggestion from the coordinator, is far away from deciding on any other value.

Both these protocols give unique numbers to rounds in such a way that if a coordinator receives votes from two members that were in two different rounds, then the vote of the member with the larger round number can be recognized as reflecting a more advanced system state, in the sense of Theorem 5.4, than the vote of the member with the smaller round number. In particular, in both protocols, whenever the coordinator receives votes from members with different round numbers, picks the vote of one of the members whose round number is the largest, an action that complies with Theorem 5.4. Recall that the original three phase commit protocol of Skeen [8] does not give numbers to rounds in this way, which is why there are $\lceil \frac{n+1}{2} \rceil$-connected executions of Skeen's protocol which never terminate.