Understanding Partitions and the "No Partition" Assumption*

Aleta Ricciardi[1]
Andre Schiper[2]
Kenneth Birman[1]

TR 93-1355
June 1993

Department of Computer Science
Cornell University
Ithaca, NY 14853-7501

*To appear in the Fourth IEEE Workshop on Future-Trends of Distributed Computing Systems, Lisboa, Portugal, September 22-24, 1993.

[1]Research supported by DARPA/ONR Grant N00014-92-J-1866, and by grants from IBM, HP, Siemens, and GTE.

[2]On leave from Ecole Polytechnique Federale de Lausanne, Switzerland. Research supported by the "Fonds national suisse" and OFEB under contract number 21-32210.9, as part of the European ESPRIT Basic Research Project Number 6360 (BROADCAST).

Understanding Partitions and the "No Partition" Assumption*

Aleta Ricciardi[1], André Schiper[2], Kenneth Birman[1]

Department of Computer Science, Upson Hall
Cornell University
Ithaca, NY 14853-7501

Abstract

The paper discusses partitions in asynchronous message-passing systems. In such systems slow processes and slow links can lead to virtual partitions that are indistinguishable from real ones. This raises the following question: what is a "partition" in an asynchronous system? To overcome the impossibility of detecting crashed processes in an asynchronous system, our system model incorporates a *failure suspector* to detect (possibly erroneously) process failures. Based on *failure suspicions* we give a definition of partitions that accounts for real partitions as well as virtual ones. We show that under certain assumptions about the process behavior, any incorrect failure suspicion inevitably partitions the system. We then show how to interpret the *"absence of partition"* assumption.

1 Introduction

The paper considers message-passing asynchronous systems in which processes fail by crashing. These systems are necessarily concerned with network partitions, and as systems grow to larger and larger numbers of nodes, handling partitions becomes more pressing. Despite this, researchers commonly assume networks will not partition because this greatly simplifies protocol development. Since protocols based on the "absence of partitions" assumption are correct only to the extent that the assumption is valid, it is vital to know whether the assumption is justified, and what the consequences are when it is not.

Unfortunately "absence of partitions" is a very imprecise specification. It is usually understood to be either (1) the absence of *link failures*, or (2) that any two operational processes p and q can always communicate. In asynchronous systems communication delays are unbounded, and local clock rates may drift arbitrarily making it impossible to determine whether a process's lack of response is caused by a link problem (failed or heavily loaded) or the process itself (crashed or very slow). In this way, slow processes and links can lead to *virtual partitions*, which are indistinguishable from real partitions. A protocol assuming the "absence of partitions" must exclude these virtual partitions as well as physical ones. This paper gives a precise definition of partition, accounting for the nature of asynchronous systems by covering virtual partitions as well as physical ones.

*To appear in the Fourth IEEE Workshop on Future-Trends of Distributed Computing Systems, Lisboa Portugal, September 22-24, 1993.

[1]Research supported by DARPA/ONR Grant N00014-92-J-1866, and by grants from IBM, HP, Siemens, and GTE.

[2]On leave from Ecole Polytechnique Fédérale de Lausanne, Switzerland. Research supported by the "Fonds national suisse" and OFEB under contract number 21-32210.9, as part of the European ESPRIT Basic Research Project Number 6360 (BROADCAST).

We are concerned with distributed fault-tolerant applications. Fault-tolerance can be understood in two ways, meaning either (1) that failures will not cause the application to take unsafe actions, or (2) that the application is able to make safe progress even if (some) processes crash. The first interpretation considers fault tolerance as a safety issue only; the second adds a liveness requirement to the safety issue. We consider the latter. To illustrate the liveness issue, consider mutual exclusion implemented using a token. If the token holder process crashes, liveness requires that the token be regenerated.

In these contexts, satisfying the liveness requirement raises the issue of detecting process failures, and leads us to introduce a mechanism for *suspecting* failures. This mechanism, associated to each process p, is called the *failure suspector* FS(p). The liveness requirement just described translates to a liveness requirement for our failure suspector: FS(p) is required to eventually detect each real crash. Unfortunately the price of liveness is accuracy: in requiring bounded-time detection of true crashes we necessarily risk erroneously suspecting non-crashed processes. While a discussion of handling inaccurate failure suspicions is beyond the scope of the paper, understanding one key issue will help in understanding the process behavior considered in Section 2. There are two ways to handle the possibility of inaccurate failure suspicions. Consider two processes p and q, and assume that p has been notified by FS(p) that q has crashed. One alternative for p is to eventually adopt the failure belief as being correct. This allows p to take any actions required by q's failure (for example to regenerate the token if q was the token holder). In this model, failure beliefs become *stable*. The other choice is for p to be permitted to "change its mind" regarding q's failure. In this second model failure beliefs are not stable, hence it is inappropriate to take actions that would be unsafe if the failure is not real. For example, in this model, p could not safely regenerate a token held by q after suspecting q's failure, because the suspicion could later prove to be incorrect.

The "stable failure" model has, for example, been adopted by the Isis system [1], whereas the "non-stable failure" model is considered in [3, 2]. The "stable failure" model is often necessary in building live, fault-tolerant applications. We saw this above in the case of token regeneration, but the same issues arise in many problems, such as primary-backup computing, and replicated data management. To achieve both liveness and safety, we must overcome the inaccuracy of failure beliefs with some form of *group-wise consistency*: if p incorrectly suspects q to have failed, and yet wants to take a safe action related to q's failure, p must be ensure that its belief in q's faultiness will be shared by other processes with which it subsequently interacts. In particular, if p observes the failure of q and then communicates with r, a consistency goal might be that r will also observe the failure of q before it delivers this message.

To summarize, the stable-failure belief model achieves liveness (in a probabilistic sense); safety is ensured by requiring some form of consistency among failure beliefs.

Further implications of failure belief stability on processes behavior is discussed in Section 2, which presents the system model and introduces our *failure suspector*. Section 3 defines partitions and proves that a single incorrect failure suspicion leads, inevitably, to a partition. This result makes the "no partition" assumption questionable. Section 4 discusses how weakening the system model while strengthening the failure suspector can prevent partitions. Unfortunately, the the stronger failure suspectors are not implementable, in even the barest model of asynchrony. Section 5 concludes the paper by relating this back to the no-partition assumption.

2 System model and Failure Suspector

The system model consists of a name space of process identifiers, $\mathsf{Proc} = \{p_1, p_2, \ldots\}$. The name space is infinite to model infinite executions in which processes continually fail and recover, though at any point in time there are only a finite number of executing processes. For the processes in this set, we assume a completely-connected network of channels. Processes communicate only by passing messages over these channels. The system has no global clock and message transmission delays are unbounded. Processes fail by crashing, which we model by the distinct event *crash*; we model the recovery of a process by assigning it a new identifier. Any process p may send a message m to any process q, $send_p(q,m)$, deliver a message m' sent by some process r, $dlvr_p(r,m')$, and perform local computation. A process *history* is a linear sequence of events with a unique *start* event, in which e_p^x denotes the x^{th} event p executes. A system *run* is a tuple of infinite process histories,[1] one for each process that executes. A *cut* is a finite prefix of a run.

As in [5], event e_q of process q *directly precedes* event e_p of process p in run ρ (written $e_q \xrightarrow{1}_\rho e_p$) if either (1) $e_q = send_q(p,m)$ and $e_p = dlvr_p(q,m)$ in ρ, or (2) $p = q$, $e_q = e_p^x$, and $e_p = e_p^{x+1}$ in p's execution history in ρ. Event e_q *precedes* event e_p in ρ (written $e_q \rightarrow_\rho e_p$) if they are the beginning and end of a chain of $\xrightarrow{1}_\rho$-related events. Hereafter, we do not note the run explicitly, unless necessary. The logical formula 1-BEFORE(e_p, e_q) holds if and only if $e_p \xrightarrow{1} e_q$, whereas the formula BEFORE(e_p, e_q) holds if $e_p \rightarrow e_q$. When e_p is an event in cut c and BEFORE(e_q, e_p) holds on c, then c is *causally consistent* if and only if e_q is also an event in c.

2.1 The Failure Suspector

Crash failures are surprisingly difficult to handle in asynchronous systems. Fischer et al. [4] show that, because it is impossible to distinguish a crashed process from one that is just very slow, any problem requiring "all correct processes" to take some action cannot be solved deterministically. One way around this is for asynchronous systems to incorporate a mechanism for *suspecting* failures and a policy for handling failure suspicions [7]. We consider a failure suspector associated to each process p, denoted FS(p). When FS(p) suspects process q, it causes p to execute the event $faulty_p(q)$. The following formulas will be useful [2]:

- ALIVE$_p(q)$ holds once p is aware of q's existence[3] and until FS(p) suspects q.
- FAULTY$_p(q)$ holds once p executes $faulty_p(q)$.
- CRASHED$_p$ holds as soon as p has executed the local event $crash_p$. It is a stable formula.

To ensure fault-tolerant applications are live, we need only require that failure suspectors eventually suspect true crashes. In asynchronous systems this takes the following form: if there is some point in p's execution after which q does not *directly* affect p, then FS(p) will suspect q faulty, or p will eventually crash. Note that FS(p) is easily implemented, for example with local time-outs. Note also, that since FS(p) operates along with p it can only guess whether q will ever directly affect p; it may use sophisticated techniques, but it can only approximate with probability whether q is crashed. As a result FS(p) will make erroneous suspicions.

[1] We make histories of crashed processes infinite by appending infinitely many *crash* events.

[2] Formulas are evaluated on consistent cuts to better model asynchronous systems. The basic formulas are propositional. Given formula φ, and consistent cut c the tense logic formulas are

- $\Box\varphi$: φ is true on c and all future cuts,
- $\Diamond\varphi$: in every run that includes c, φ holds at some future cut.

To distinguish logical formulas from events, formulas are written in SMALL CAPS. For example the formula SEND$_p(q,m)$ holds along c if and only if the last event p executed in c was $send_p(q,m)$.

[3] We do not discuss process creation and incorporation into the set of operational processes.

FS(p) Liveness. For all executions and all processes q, unless there are an infinite number of event-pairs satisfying $e_q \xrightarrow{1} e_p$, eventually FS(p) suspects process q. Formally, define the set of directly-related event pairs between q and p as :

$$Causes_p(q, \rho) = \left\{ (e_q, e_p) \mid e_q \xrightarrow{1}_\rho e_p \right\}$$

If this set is finite, eventually p suspects q or p crashes:

$$\mid Causes_p(q, \rho) \mid < \infty \Rightarrow \Diamond\text{FAULTY}_p(q) \vee \Diamond\text{CRASHED}_p$$

$\boxed{\text{END}}$

We are not concerned with how FS(p) is implemented, only that it be live.

2.2 Other System Model Properties

Processes in our system model are also subject to certain constraints. As discussed in Section 1 these arise from the liveness requirements of the applications and the impossibility of accurately detecting a process-crash. The liveness requirement led us to adopt the "stable failure" model. Stability of failure beliefs is our first process property.

Failure Belief Stability. A failure belief once adopted is true forever: $\text{FAULTY}_p(q) \Rightarrow \Box\text{FAULTY}_p(q)$.

This has an immediate consequence for channel behavior: once p believes q faulty, it neither accepts further messages from q, nor sends further messages to q. Is this reasonable? Recall the liveness requirement put on the applications we consider. Assume that $\text{FAULTY}_p(q)$ holds. This might lead p to take some actions A in order to recover from q's suspected failure. Consider now a message m from q, received by p once $\text{FAULTY}_p(q)$ holds. Accepting m might lead p to execute an action A' inconsistent with action A. To avoid this type of inconsistency (without forcing p to crash or inspect messages' contents before delivering them) p rejects any further messages from q once $\text{FAULTY}_p(q)$ holds. That p acts symmetrically in not sending to q further messages upon believing q faulty is reasonable. This lead to the second process property.

Channel Disconnect:

$$\text{FAULTY}_p(q) \Rightarrow \left(\Box\neg\text{DLVR}_p(q, m) \wedge \Box\neg\text{SEND}_p(q, m) \right).$$

Toward achieving failure belief consistency, we introduce a third process property. We motivated the need for failure belief consistency as a consequence of the inaccuracy of any live failure detection mechanism. Differing failure beliefs could easily result in unsafe (i.e. inconsistent) actions. Safety can be regained if some form of consistency among failure beliefs is achieved. This is precisely the role of the *gossip* property.

Gossip. Failure beliefs propagate along causal chains of events. For processes p, q, and r:

$$\begin{aligned}&\text{BEFORE}(\mathit{faulty}_p(q), e_p) \wedge \text{1-BEFORE}(e_p, e_r) \Rightarrow \\ &\quad \text{BEFORE}(\mathit{faulty}_r(q), e_r) \vee \text{1-BEFORE}(e_r, \mathit{faulty}_r(q))\end{aligned}$$

In summary the liveness and safety requirements put on the fault-tolerant distributed applications we have in mind, led to three system process requirements: stability of failure beliefs, channel disconnect, and gossip of failure beliefs.

3 From Incorrect Failure Notifications to Partitions

We now show that given the system model and failure suspector just described, partitions are unavoidable. We introduce the ISOLATED() property, define partitions, and then prove the result.

Definition [ISOLATED(S)] Given S a subset of Proc(c), ISOLATED(S) holds on c if and only if every process considered alive by some $p \in$ S, is also in S:

$$\text{ISOLATED}(\mathsf{S}) \stackrel{\text{def}}{=} \bigwedge_{p \in \mathsf{S}} \bigwedge_{q \in \mathsf{Proc}(c)} \Big(\text{ALIVE}_p(q) \Rightarrow q \in \mathsf{S}\Big).$$

END

If ISOLATED(S) holds, the processes in S believe themselves to be the only live processes in the system. With this definition, it is natural to declare a system partitioned exactly when there are at least two disjoint subsets that each believe themselves to be the only live processes in the system.

Definition [Partition] A *partition* exists along consistent cut c if at least two non-null, disjoint subsets of Proc(c) are isolated. END

We now show that a single incorrect failure belief partitions the system.

Proposition 3.1 *[Failure Belief Propagation] If p believes q faulty then eventually every other process r either believes q faulty, believes p faulty, or crashes:*

$$\text{FAULTY}_p(q) \;\Rightarrow \Diamond\Big(\text{FAULTY}_r(q) \vee \text{FAULTY}_r(p) \vee \text{CRASHED}_r\Big).$$

PROOF Let e_p be the event $faulty_p(q)$. Let $e_r \neq crash_r$, not be causally dependent on e_p (that is, $\neg\text{BEFORE}(e_p, e_r)$). Then either (1) there is some future event e'_r such that $e_p \rightarrow e'_r$ or, (2) there will never be a causal relation between e_p and any future event on r.

Clause (1) is the Gossip premise, in which case $\text{FAULTY}_r(q)$ holds immediately after e'_r, while clause (2) is the premise of FS(r) liveness. Thus $\text{FAULTY}_r(p)$ eventually holds, or r crashes. QED

Proposition 3.2 *[Failure Reciprocity] If p believes q faulty, then eventually either q believes p faulty, or q crashes:*

$$\text{FAULTY}_p(q) \;\Rightarrow \Diamond\Big(\text{FAULTY}_q(p) \vee \text{CRASHED}_q\Big).$$

PROOF Let e_p be the event $faulty_p(q)$. To prove reciprocity (via FS(q) Liveness) we must show that no p-event on p after e_p directly precedes an event on q. Clearly, this cannot happen without p violating Channel Disconnect by sending to q once it believes q faulty. QED

In other words, failure reciprocity is inevitable if any failure suspicion is incorrect. The following proposition shows that such a mistake partitions the system.

Proposition 3.3 *If p erroneously suspects q faulty, but neither q nor p ever crashes, then eventually there are at least two disjoint subsets, S and T, such that $p \in \mathsf{S}, q \in \mathsf{T}$, $\text{ISOLATED}(\mathsf{S})$ and $\text{ISOLATED}(\mathsf{T})$.*

PROOF Rename $q \equiv q_0$ and let c be the consistent cut along which $\text{FAULTY}_p(q_0)$ initially holds, and define $\text{A-Set}_p(c)$ to be all processes p believes alive at c.

Once $\text{FAULTY}_p(q_0)$ holds, Gossip means that eventually every $r \in \text{A-Set}_p(c)$ either adopts $\text{FAULTY}_r(q_0)$, crashes, or believes $\text{FAULTY}_r(p)$. Without loss of generality, assume only p gossips *faulty*(q_0), and let c_1 be the consistent cut at which *faulty*(q_0) is gossiped (as far as possible) to the members of $\text{A-Set}_p(c)$. Let S_1 (along c_1) be the subset of $\text{A-Set}_p(c)$ that adopted $\text{FAULTY}(q_0)$, with the others having either crashed or adopted $\text{FAULTY}(p)$:

$$\mathsf{S}_1 = \left\{ r \mid \text{FAULTY}_r(q_0) \wedge \neg\text{FAULTY}_r(p) \wedge \neg\text{CRASHED}_r \right\}.$$

If S_1 is not isolated then there is some $r \in \mathsf{S}_1$, such that $\text{ALIVE}_r(q_1)$ holds at c_1, but $q_1 \notin \mathsf{S}_1$. Since *faulty*(q_0) is fully gossiped, the only reasons this q_1 would not have adopted $\text{FAULTY}_{q_1}(q_0)$ are (1) it already believed $\text{FAULTY}_{q_1}(p)$, or (2) q_1 had crashed. In either case $\text{FAULTY}_p(q_1)$ eventually holds – in the first case by reciprocity, and in the second by FS(p) liveness.

Now, p must gossip *faulty*(q_1), so let c_2 and S_2 be c_1 and S_1 after having gossiped q_1's faultiness. We can continue in this way: any process q_i that did not adopt p's belief in the faultiness of process q_{i-1} must either be crashed or believe p faulty. Eventually, some S_k is isolated; in the worst case S_k is the singleton $\{p\}$. Take $\mathsf{S} = \mathsf{S}_k$.

Analogously, reciprocity means that $\text{FAULTY}_{q_0}(p)$ eventually holds, and we can construct T, as we did for S, from $\text{A-Set}_{q_0}()$.

To see that the two isolated sets are disjoint note that $r \in \mathsf{S} \Rightarrow \text{FAULTY}_r(q_0)$. Reciprocity means that $\text{FAULTY}_{q_0}(r)$ eventually holds, and construction of T ensures $r \notin \mathsf{T}$. $\boxed{\text{QED}}$

Since we can never guarantee the failure suspectors will not make mistakes, the "no partition" assumption is invalid in the system model considered.

4 Understanding the No Partition Assumption

Given Proposition 3.3 it is important to know how an incorrect failure suspicion partitions the system, and whether we can alter our model to prevent partitions given the inevitability of incorrect suspicions.

From the definition of $\text{ISOLATED}()$, partitions occur when failures are reciprocated. So assuming $\text{FAULTY}_p(q)$ holds erroneously, how can q be prevented from believing $\text{FAULTY}_q(p)$? Since we cannot sacrifice FS(q) liveness, we are left with three choices:

1. Force q to crash before it believes and is able to propagate $\text{FAULTY}(p)$. Lacking an omniscient observer, only p can attempt to cause q to crash because only p knows it executed *faulty*$_p(q)$. Unfortunately, the absence of synchronization mechanisms means p can never ensure that any command telling q to crash itself will arrive at q before q reciprocates with (and propagates) *faulty*$_q(p)$.[4]
2. Force failure suspectors to attain a quorum-style agreement on suspicions before actually emitting the *faulty*() suspicion. This is done in [6, 7] where the quorum is a simple majority. Processes in a

[4] This strategy will not preclude permanent partitions that arise from temporary link failures.

majority subset can take further actions, while those in the minority cannot. Whether a majority can be obtained determines whether the system can progress. This is further discussed in Section 4.1.

3. Concede failure belief stability at the expense of guaranteeing system progress. We explore this option in Section 4.2.

4.1 The Primary Partition Model

The "primary partition" model is one in which the system is allowed to partition, but one assumes that there is always an identified *primary* partition that is unique, in being the only partition so designated, and in which decisions can be made on behalf of the system as a whole, without risk that contradictory decisions will be made in other partitions. The primary partition model is often considered weaker than the "no partition" model: the former allows progress in the primary partition, while the latter would not allow progress if any partition were ever to form.

Specifically, neither the no-partitions model nor the primary-partition model can guarantee progress (of the type of distributed problems we are concerned with) in situations where consensus cannot be solved. Since the primary-partition model ensures liveness in situations where the no-partitions model would not, we recommend that the primary partition model be assumed in most algorithms that make assumptions about partitioning.

4.2 Conceding Failure Belief Stability

In conceding failure belief stability we no longer need the Channel Disconnect and Gossip properties. Channel Disconnect was introduced as a consistent consequence of failure belief stability. Gossiping is used to bring about consistency of failure beliefs, but lacking stability a process may change its belief immediately after being gossiped another's failure: consistency of failure beliefs is no longer an issue.

In this section, we assume neither stability, disconnect, nor gossip and derive additional requirements on the system's failure suspectors that would preclude partitions. In particular, partitions cannot exist if for all cuts, c, some process is believed alive by every process in $\mathsf{Proc}(c)$.

Proposition 4.1 *If on all cuts c, all failure suspectors agree on some subset of non-faulty processes, then partitions will never occur.*

PROOF (By contradiction) Let $\mathsf{F\text{-}Set}_p(c) = \overline{\mathsf{A\text{-}Set}_p(c)}$; then $\mathsf{Proc}(c) = \mathsf{A\text{-}Set}_p(c) \cup \mathsf{F\text{-}Set}_p(c)$. A partition on a cut c means that there are (at least) two disjoint sets that are both isolated on c. Call them S and T.

By definition

$$\mathsf{S} = \bigcup_{p \in \mathsf{S}} \mathsf{A\text{-}Set}_p(c) \ \text{ and } \ \mathsf{T} \ = \ \bigcup_{q \in \mathsf{T}} \mathsf{A\text{-}Set}_q(c).$$

De Morgan's Law give $\overline{\mathsf{S}} \cup \overline{\mathsf{T}} = \mathsf{Proc}(c) \ \Leftrightarrow$

$$\bigcap_{p \in \mathsf{S}} \mathsf{F\text{-}Set}_p(c) \ \cup \ \bigcap_{q \in \mathsf{T}} \mathsf{F\text{-}Set}_q(c) \ = \ \mathsf{Proc}(c).$$

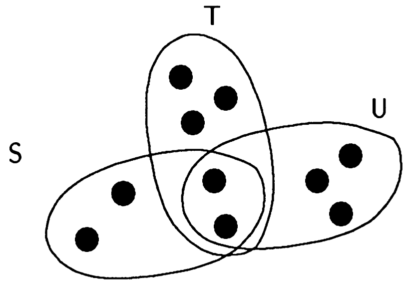

Figure 1: Intersecting Isolated Sets Must Form a 'Star'

Thus, Proc() is partitioned at c if and only if every process in the system is believed faulty by every member of some isolated set. $\boxed{\text{QED}}$

In summary, preventing partitions requires that some process in the system is not believed faulty by some member of every isolated set. The implications of this are stated in Propositions 4.2 and 4.3.

Proposition 4.2 *The intersection of isolated sets is isolated.*

PROOF Let S andT be isolated and consider $p \in \mathsf{S} \cap \mathsf{T}$, and $r \in$ A-Set$_p(c)$. Because S is isolated and $p \in \mathsf{S}$, r must also be in S. Similarly, ISOLATED(T) and $p \in \mathsf{T}$ give $r \in \mathsf{T}$. $\boxed{\text{QED}}$

Now consider three isolated sets S, T, and U such that $\mathsf{S} \cap \mathsf{T} \neq \emptyset$, and $\mathsf{T} \cap \mathsf{U} \neq \emptyset$. Unless these intersections also intersect a partition exists. In other words, a partition will not exist as long as isolated sets form a 'star', as depicted in Figure 1.

Proposition 4.3 *Let* $\mathsf{S}_1, \mathsf{S}_2, \ldots, \mathsf{S}_n$ *enumerate the isolated sets along cut c. Then no partition exists if and only if one of them is the center of a 'star'. That is,* $\exists 1 \leq x \leq n : \mathsf{S}_x = \mathsf{S}_1 \cap \mathsf{S}_2 \cap \cdots \cap \mathsf{S}_n$.

PROOF Follows easily from the definition of partition and Proposition 4.2. $\boxed{\text{QED}}$

We say Proc(c) is *degraded* if Proc(c) is carved into isolated subsets but is not partitioned. It is *fully-degraded* if $\mid \text{Proc}(c) \mid = n$, there are n isolated subsets such that $n-1$ isolated subsets are process pairs, and one isolated subset is a singleton. This represents the worst-case, non-partitioned separation (or, equivalently, belief consistency) between all process pairs.

4.3 Interpreting Partitions and Distributed Consensus

The star formation is exactly analogous to Chandra et.al.'s work on Weak Failure Suspectors [3, 2]. This work proves that if some functional process is not suspected (for a sufficiently long period of time) by every other functional process, Distributed Consensus can be solved. This corresponds, in our terminology, to the absence of a partition. Essentially, the failure suspector $\diamond\mathcal{W}$ requires eventual absence of partitions for a critical period of time (i.e. long enough to run their protocol). Note that the results of [3, 2] also hold in the primary partition of a primary partition model.

5 Conclusion

The paper has given a precise definition of partition, accounting for the nature of asynchronous systems by covering virtual partitions as well as physical ones. The paper has further considered two classes of fault-tolerant distributed applications, characterized by the stability vs. non-stability of failure beliefs. The stable-failure system model has been completed by process properties that follow logically from the stability requirement. They lead to the following result: a single incorrect failure suspicion already leads to partition the system. The only safe way to avoid partitions is to require the failure suspectors to attain a quorum agreement on suspicions before actually emitting the *faulty*() suspicion. As any so called *membership service* assumes the stable-failure system model, any membership service has to include such a quorum condition.

The paper has also shown that absence of partition is obtained by requiring that every failure suspector 'agree' on a subset of non-faulty processes. This is a valid "no partition" assumption in the "non-stable failure" system model. Finally, the paper suggests that the "no-partition" assumption be related to a "primary-partition" assumption when possible. Although a system that takes this approach will still be unable to make progress in runs for which consensus could not also be solved, such an assumption is less restrictive, more practical, and hence preferable to a no-partitions one.

References

[1] K. P. Birman. The Process Group Approach to Reliable Distributed Computing. Technical Report TR-91-1216, Cornell University, July 1991. To appear in CACM.

[2] T. D. Chandra and V. Hadzilacos andS. Toueg. The Weakest Failure Detector for Solving Consensus. In *Proceedings of the 11th Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 147–158. ACM, August 1992.

[3] T. D. Chandra and S. Toueg. Unreliable Failure Detectors for Asynchronous Systems. In *Proceedings of the Tenth Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 325–340. ACM, August 1991.

[4] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of Distributed Consensus with One Faulty Process. *Journal of the Association for Computing Machinery*, 32(2):374–382, April 1985.

[5] L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the A.C.M.*, 21(7):558–565, 1978.

[6] A. Ricciardi and K. Birman. Using Process Groups to Implement Failure Detection in Asynchronous Environments. In *Procedings of the Tenth Annual A.C.M. Symposium on Principles of Distributed Computing*, pages 341–351. A.C.M., August 1991.

[7] A. M. Ricciardi. *The Asynchronous Membership Problem.* PhD thesis, Cornell University, January 1993.