COORDINATION PROBLEMS IN DISTRIBUTED SYSTEMS
(Extended Abstract)

Danny Dolev

Ray Strong

IBM Almaden Research Center

Abstract: In this paper we provide a framework for understanding and comparing various lower bounds and impossibility results concerning the solution to coordination problems like atomic commit for distributed transactions and other consensus problems. The key to our treatment is the distinction between information that is transferred by (datagram) messages and information that can be deduced from the existence of other synchronization mechanisms. We provide an axiomatic theory of distributed systems in which such distinctions can be made; and, in the context of this theory, we describe a single paradigm involving the isolation of one process from the effects of another process in a distributed system. Using this paradigm we obtain several well known impossibility results including the impossibility of a synchronous system with a safe and timely (nonblocking) atomic commit in the presence of a possible communication partition, the impossibility of an asynchronous system that reaches consensus in the presence of at most one crash fault, and the impossibility of a solution to the Chinese Generals' Problem. We explain how these impossibility results imply the impossibility of a nonblocking consensus or atomic commit in a synchronous or asynchronous system in which all faults are eventually repaired but synchronous communication is not guaranteed.

Having established this family of limits on distributed systems, we discuss the feasibility of achieving with high probability goals that are impossible to guarantee. We suggest a family of simple atomic commit protocols that guarantee safety (consistency of data) and provide timeliness (absence of blocking) with high probability (depending on the probabilities of communication link faults and processor faults). We propose the adoption of such protocols for atomic commit and similar coordination problems when safety is necessary but timeliness and high availability are of very high priority.

1. Introduction.

In this paper we focus on some of the key problems of fault tolerant distributed systems and what makes them hard to solve. In fact we offer a unified theory that allows interpretation and comparison of many of the most important known limits on distributed systems. Among the demands placed on the designer of a distributed system, there is a competition between opposing demands for safety (consistency) and timeliness. We will argue that systems that require timeliness are generally subject to a paradigm we call *vulnerability*. Our notion of vulnerability is an attempt to capture at a very high level of abstraction the possibility that faults or delays may force some processes in a distributed system to make decisions and take actions that are not entirely coordinated with those of other processes that remain functioning. Our results are obtained by carefully distinguishing between the relation of causality that holds between events in a distributed system because of message passing and that implied by the existence of special multiprocess atomic operations or synchronizing events. By focussing on incompatibilities among actions taken by distinct processes, we capture part of the intuitive notion of a messageless information transfer in the presence of multiprocess synchronization mechanisms. We explain why messages alone are not sufficient to provide both safe and timely coordination protocols and what properties of multiprocess mechanisms are required.

The most heavily studied coordination problem is the problem of atomic commit for distributed database transactions [BHG, G]. In the atomic commit problem, participants are data management processes residing at various sites and communicating by messages. When a distributed transaction involves modification of data at

more than one site, the participants must coordinate whether and when to commit these changes. For reasons independent of faults and beyond the scope of the problem, any participant may unilaterally decide that the transaction should be aborted. But participants may give up their right to such a unilateral decision and guarantee that they will make the appropriate modifications if the transaction is committed. This guarantee is typically accompanied by expensive transmissions to some stable storage device and by the locking of resources that cannot be touched by other transactions until the decision to commit or abort is reached.

Presumably most of the time all participants agree to commit. The problem is to devise a protocol so that once all participants have made their unilateral decisions, the transaction is either committed or aborted by all processes, that continue to function correctly, within a bounded time, despite communication and process failures. Furthermore, when any failed process recovers it must be able to ascertain whether the modifications should be made or not as part of its recovery procedure. The well known two phase commit protocol solves most of the problem, but the failure of a single process can force others to wait indefinitely in what is called a blocking state, neither committing nor aborting the changes. Three phase commit protocols have been developed that tolerate process crash failures but cannot tolerate partition of communication between functioning processes [S,SS,DS]. We will show that it is the multivalent nature of the outcome together with the necessity to decide, even in isolation, that makes a complete solution to the atomic commit problem impossible.

Note that this result does not contradict the results of [SS], the difference resides in the important distinction between information transfer and messages. In [SS] an "optimistic" model is explored in which communication is synchronous and a message is guaranteed delivered or returned (but not both). This model provides a good example of messageless information transfer: both processes know when a message that was sent has not been received. In this model single partitions can be tolerated but if multiple partitions are possible, then a safe and timely commit becomes impossible.

The earliest discussed glimpse at this phenomenon and one of the most intuitive comes in the form of the Chinese Generals' Problem [G]. In this problem, we are given a situation with two generals of allied armies encamped on opposite hills above a valley full of the enemy. The generals' problem is to coordinate a time of attack. However, the only way they can communicate is by messenger through the valley of the enemy. The problem is to devise a protocol for this communication that guarantees that they either both attack (at the same time) or they both retreat (at the same time). The difficulty is that the sender cannot know if a given message arrived unless an acknowledging message is returned. As part of a solution protocol, the generals may have prearranged that if no messages get through by a given time, then they will both retreat; but there must be some possibility of attack. We will show that it is the multivalent nature of the outcome together with the possibility that a decision may have to be made in isolation that makes a solution to the Chinese Generals' Problem impossible.

One of the most famous recent impossibility results in computer science is the result of [FLP] that it is impossible to reach consensus in an asynchronous system in the presence of at most one process crash failure, even when eventual communication is guaranteed. The consensus problem is a coordination problem in the sense that all the processes that do not fail must agree on a common output. Multivalence is introduced because each process is given an input from a set of at least two values, and if all inputs agree, then all outputs must agree with the inputs. Note the similarity to the atomic commit problem: each process has some "input" that allows it to make a unilateral decision on whether to abort or allow commit; if all inputs allow commit then the agreed on output must be commit. There are only two essential distinctions between the two problems. One is the constraint placed on processes recovering from failure in the atomic commit problem. The consensus problem is easier to solve because it puts no constraints on the actions taken by processes once they have failed. The other is the asynchronous environment associated with the consensus problem. A fundamental characteristic of an asynchronous environment is that external information is only conveyed to a process when it receives a message, no information can be learned from the absence of messages. We will show that

asynchrony plus the possibility of process failure provides a multivalent system that is vulnerable to isolation and that neither consensus nor safe atomic commit is possible.

This negative result is easier than that of [FLP], because our model of a completely asynchronous system allows communication to be stopped forever. The model of [FLP] provides for guaranteed eventual communication; but allows a process to stop forever. We show how to adapt the proof of [FLP] to our paradigm. Moreover, we show that, even in a model with guaranteed eventual communication and guaranteed eventual process repair, safety requires blocking. Our contribution here is the conversion of the [FLP] result to results in other models that fall within the scope of our theory of distributed systems. We can also take much simpler proofs of impossibility in totally asynchronous systems and translate them into results about systems with eventual process repair.

Having discussed the negative results, we will offer some positive alternatives, discussing the feasibility of minimizing the probability of unwanted system behavior despite faults. It turns out that the unwanted system behavior introduced by asynchrony in the [FLP] model can be eventually overcome with probability 1 ([BO, R]). This means that even though there can be no guarantee of termination within any fixed amount of time, the probability of running forever without termination can be reduced to 0. However, the vulnerability introduced by requiring timely decisions (as for atomic commit) remains. We conjecture that the probability of unwanted system behavior cannot be reduced to zero without guaranteed communication.

Although we cannot reduce the probability of unwanted system behavior to zero, we do provide a family of algorithms for atomic commit that allows us to tune the probability somewhat. In contrast to so-called nonblocking commit algorithms like three phase commit ([S,DS]), our family can be applied to models without guaranteed communication and with arbitrary kinds of faults, although here we will concentrate on an omission fault model.

Often algorithms for atomic commit are called "nonblocking" if some subset of processes is not blocked (as in quorum based approaches). The algorithms in our family have this property; but we believe it is misleading to label such algorithms nonblocking.

Our algorithms are based on a simple notion of decentralizing the commit decision to what we call a jury. The jury consists of a set of extremely simple processes that operate concurrently with a more or less standard two phase commit protocol and make recovery much simpler than three phase commit since there is no requirement to replace any given process or choose a leader.

The remainder of the paper is organized as follows: in Section 2 we provide our formal theory of distributed systems, in Section 3 we study asynchronous systems and reproduce some famous impossibility proofs, in Section 4 we study partially and completely synchronous systems and provide our most general results about blocking, in Section 5 we describe a jury construction that is designed to reduce the probability of blocking for synchronous systems, and in Section 6 we conclude with some remarks about the practicality of messageless information transfer of the kind required to provide nonblocking atomic commit.

2. Formal Model of Coordination.

We consider systems composed of independently acting processes that can communicate with each other. During an execution of a system, each process takes an infinite sequence of actions. Associated with any action a is its *sequence number* s(a) and the process at which the action takes place @(a). Thus, in any execution, s is a function from actions to positive integers and @ is a function from actions to processes. The function @ applied on a set of actions will be the set of processes at which the actions are performed. In any execution there is a relation called the *send-receive relation* on actions. We denote this relation $\rightarrow$ and say that if $a \rightarrow b$ holds, then a is a send action and b is a corresponding receive action. We also distinguish a set of actions called faulty actions. Thus, an *execution* is a finite set of infinite (ω ordered) sequences of actions, together with a send-receive relation and a possibly empty set of faulty actions. We define a partial order $<_L$ on the actions of an execution as the transitive closure of the relation

$$(@(a)=@(b)\&s(a)<s(b)) \mid (a \rightarrow b).$$

Relation $<_L$ is often regarded as a relation of causality and precedence [L, BD, NT, PT].

A *prefix* of an execution is a set of finite prefixes of the sequences of actions of an execution with the induced send-receive relation and a set of faulty actions. If y is a prefix of an execution and x is a prefix of y, then the *extension* of x by y, denoted y-x, is the set of subsequences of actions in y and not in x. If g is an extension of prefix x, we denote by x+g the prefix such that ((x+g)-x)=g. Two prefixes are said to be *compatible* if they are prefixes of the same execution; otherwise they are said to be *incompatible.* If prefix x and prefix y are compatible, we denote by x+y the prefix that consists of the union of the actions of x and y.

In addition to the $<_L$ relation, each execution also has a *possible influence* relation on actions that we denote $\leq_K$. We need this relation to capture the intuitive notion of an *information transfer* in the presence of synchronization mechanisms that may span more than one process and allow the transfer of information without messages. Note that in an asynchronous system we can take $\leq_K \equiv <_L$. Define a *cut* of an execution as a prefix that is closed under $\leq_K$ (this means, if action b is in a cut and a $\leq_K$ b, then action a is also included in the cut). Cuts correspond to the global states of our system.

Given the preceding definitions, we formally define a *system* as a set of executions satisfying the following axioms:

A1: $<_L$ is irreflexive in each execution.

A2: Every prefix of an execution is a prefix of some execution in which there are no further failures.

A3: $\leq_K$ is transitive.

A4: $\leq_K$ extends $<_L$ (this means, if a $<_L$ b, then a $\leq_K$ b).

A5: If cut x is a prefix of both cut y and cut z and if @(y-x) is disjoint from @(z-x), then y and z have a common execution (are compatible).

A6: There are at least two processes.

Axioms A1 and A2 are fairly standard. A1 expresses the idea that a message must be sent before it is received. A2 says that each process that has not failed can always continue to operate fault free indefinitely. A6 simply removes some trivial cases from consideration. A3, A4, and A5 are quite powerful. A5 is intended to capture the independence of nonsynchronized actions taken concurrently at different processes. Thus, if y and z are each prefixes that contain all actions that influence them, then y and z are either compatible or they contain incompatible actions at some common process. Because A5 is so strong, it might seem that many systems of interest could fail to satisfy it. However, in any "system" satisfying A1 and A2 we can trivially define a $\leq_K$ relation satisfying A3, A4, and A5, making sure that the only sets of actions closed under the relation are the empty set and the entire execution.

Let E be a set of executions satisfying A1 and A2. Next we show how to construct a relation that captures the notion of information transfer (using any mechanism beyond messages or any special guarantees about messages) and satisfies A3, A4, and A5.

Let x be a prefix of an execution. Let p be one of the processes of x. Let c be the last action of p in x. If a is an action with @(a)=p and s(a)=s(c)+1 and if the extension of x by a is also a prefix of some execution in the system, then a is said to be *enabled* at x. Actions a and b are said to have *antipathy* at x if both a and b are enabled at x but no execution of which x is a prefix contains both a and b. Intuitively, antipathy reflects some form of synchronization that prevents both actions from occurring, although each one of them is enabled. Actions a and a' are said to be *similar* if @(a)=@(a') and s(a)=s(a'). If actions c and b occur in execution e, @(c)=@(b), and s(c)<s(b), then c is said to be an *antecedent* of b in e. If a and b both occur in an execution e, then we define *the similarity class of a over b,* denoted [a/b], as the set of actions similar to a that occur in some execution that contains all the antecedents of b in e. If actions a and b both occur in execution e, then a is said to *influence* action b if some action in [a/b] has antipathy with b at some prefix of e that contains all the antecedents of b in e. Intuitively, the process @(b) learns something about [a/b] by taking action b that it did not know before taking the action. We define the relation $\leq_K$ as the transitive closure of the relation (a $<_L$ b) | (a influences b). Note that the actions in a cut are not necessarily known to any process, nor can they

be determined by an observer of only one execution, since the $\leq_K$ relation is defined in terms of all executions.

It is immediate that $\leq_K$ satisfies A3 and A4. We next show that the relation defined above satisfies A5.

Theorem.2.1: If cuts are defined as prefixes closed under the transitive closure of $(<_L) \mid$ (influences), then axiom A5 is satisfied.

Proof of Theorem 2.1: Suppose that y and z are incompatible cuts that both extend prefix x. There must be a prefix u of y and an action a such that u+a is a prefix of y and u and z are compatible but u+a and z are incompatible. There must then be a prefix v of z and an action b such that v+b is a prefix of z and v and u+a are compatible but v+b and u+a are incompatible. Since v and u+a are compatible, v and u are compatible and a is enabled at u+v. Since u and z are compatible, u and v+b are compatible and b is enabled at u+v. Thus a and b have antipathy at u+v. Let e be an execution containing cut z. Let a' be the action in e similar to a. Then a' influences b in e because a is in [a'/b] and u+v contains the antecedents of b in e. Hence z contains an action similar to a but x does not. Likewise, y contains an action similar to b but x does not. Thus @(y-x) and @(z-x) both contain @(a) and @(b).
QED

A system solves a *coordination problem* if each process has a set of *output actions* from which it must take at most one action in every execution. For the rest of this paper, we assume that each system solves a coordination problem.

A system is *recoverably safe* if all process outputs must agree; it is *safe* if outputs of processes that have not failed must agree. A cut x is said to be *deciding* if some process gives its output in x without failing. An extension e of cut x is said to be *deciding* if x+e is deciding.

A system solves a *nontrivial coordination problem* if there is a cut that has two deciding extensions so some output of a nonfailing process in one disagrees with some such output in the other. Such a cut is called *multivalent*. A coordination problem is called *recoverably timely (timely)* with respect to a set of cuts s if there is a bound b on the number of actions each process takes after s and before it gives its output (or fails).

The set of all cuts in an execution is partially ordered by the prefix ordering. If cut x is a prefix of cut y and there is no cut z such that x is a prefix of z and z is a prefix of y, then y-x is said to be a *primitive extension* of cut x. Here cut y is said to be a *successor* of cut x.

If x is a prefix of a set of cuts Y, then cut z is said to be an *isolation* of Y over x if x is a prefix of z and @(z-x) is disjoint from @(Y-x), the set of processes that appear in @(y-x) for some y in Y. In this case Y is *isolated* by z at x. If Y consists of a single cut y, then z is said to be an *isolation* of y over x and y is said to be *isolated* by z at x. Let Y(p) represent the set of successors y of cut x such that @(y-x) contains process p. If, for each process p, Y(p) can be isolated at x, then x has the *isolation property.*

Theorem 2.2: If $\leq_K$ is irreflexive then every cut that has successors involving different processes has the isolation property.

Proof of Theorem 2.2:

If $\leq_K$ is irreflexive then every primitive extension involves exactly one process.
QED

Let A be any subset of the set of all executions of a system S. If the relations $<_L$ and $\leq_K$ induce relations in A that satisfy the axioms A1 through A5, then A is said to form a *subsystem* of S. We say (subsystem or system) A is *deciding* if every execution in A has a cut in which some process gives its output without failing.

A cut x is *multivalent* in a system A, if x has deciding extensions in A with disagreeing outputs. A cut is said to be *final multivalent* if it is multivalent and has no multivalent successors. A cut is said to be *univalent* if outputs from all deciding extensions agree.

A cut y is said to be *vulnerable* if it has primitive extensions b and c such that y+b and y+c are univalent with disagreeing outputs (in some extensions) and there is an isolation d of {b, c} over y such that d is either deciding or the system is deciding and d is univalent. A system is called *vulnerable* if it contains a subsystem with a vulnerable cut.

Lemma 2.3: If a system contains a deciding subsystem that has a final multivalent cut with the isolation property, then the system is vulnerable.

Proof of Lemma 2.3: Let system S contain deciding subsystem A with final multivalent cut x that has the isolation property. Since x is final multivalent in A, x has primitive extensions b and c in A such that x+b and x+c are univalent and x+b and x+c are prefixes of deciding extensions with disagreeing outputs. Thus x+b and x+c are incompatible and there must be some process p in the intersection of @(b) and @(c). Since x has the isolation property, A contains an isolation y of the set of successors of x involving p over x. But then y is univalent because x is final multivalent and A is deciding so x is vulnerable.
QED

Theorem 2.4: A system cannot be both safe and vulnerable.

Proof of Theorem 2.4: Suppose the contrary. Let S be such a safe vulnerable system. Let A be the subsystem of S that contains a vulnerable cut x, let b and c be its univalent extensions with disagreeing outputs, and let d be the deciding or univalent extension that is an isolation of b and c over x. Since S is safe, if d is deciding then it is univalent. However, x+d must be compatible with both x+b and x+c. Thus d cannot be deciding. But then A must be a deciding subsystem and x+d must be univalent, again contradicting the compatibility of x+d with both x+b and x+c.
QED

3. Asynchronous Systems.

A sequence of cuts is said to be an *ascending chain* if each cut is a prefix of the next. A system is said to have the *ascending chain property* if the union of the actions of any ascending chain of cuts is contained in an execution of the system.

In what follows we assume a message structure in addition to the send-receive relation: in particular, we assume that with each send action is associated a list of processes called the *targets* of the message sent, and with each receive action a there is a unique send action b such that $b \rightarrow a$ and @(a) is a target of b.

A system is said to be *asynchronous* if it has the ascending chain property, $\leq_K \equiv <_L$, every cut has a successor at every process, and if x is a cut containing a send action but no corresponding receive action for a target p then x has a successor containing an action at p that is a corresponding receive action and x has a successor containing an action at p that is not a corresponding receive action. (Note that the fourth requirement is not guaranteed communication. It simply says that in an asynchronous system the "next" action at the target after a send may or may not be the corresponding receive: there is no minimum or maximum message delay.)

Theorem 3.1. Safe consensus is impossible in an asynchronous deciding system with no process failures. (This is a generalization of the Chinese Generals' Problem [G].)

Proof of Theorem 3.1: Suppose S were a safe deciding asynchronous system that solved the consensus problem with no process faults. Since S solves a consensus problem, the initial empty cut is multivalent. Since S is safe, deciding, and asynchronous, there can be no infinite ascending chain of multivalent cuts. Thus there must be a final multivalent cut. Note that in an asynchronous system, every cut has successors involving different processors, so every cut has the isolation property. Thus S is vulnerable.
QED

The proof is simple because any asynchronous system has executions in which no messages are ever received. Any safe deciding asynchronous system with a multivalent cut is vulnerable, so there are no safe deciding asynchronous systems. If we restrict attention to executions in which every message is eventually received, the proof is more complex but the end result is the same. Below we reproduce arguments from [FLP] to establish Theorem 3.2 using the proof technique of Theorem 3.1.

An execution has *guaranteed eventual communication* if every message sent is eventually received by every target that does not fail. Note that the notion of guaranteed eventual communication applies also to an infinite ascending chain of cuts with the same definition. Note also that in an asynchronous system, an infinite ascending chain of cuts is an execution, provided each process takes infinitely many actions in the chain.

A *crash fault* in an execution is a process that ceases to perform any action except for a special faulty action called a *crash action,* which the process repeats for the rest of the execution. A system is *subject to crash faults* if for every prefix x of an execution and for every process p that has not failed in x, there is an execution containing x in which p crashes after its last action in x. A subsystem of such a system consisting of all executions in which no more than k processes crash is said to be *subject to at most k crash faults.*

A system is said to be *asynchronous with guaranteed eventual communication* if it consists of the subset of executions of an asynchronous system that have guaranteed eventual communication.

Theorem 3.2: (FLP) Safe consensus for a deciding asynchronous system with guaranteed eventual communication subject to at most one crash fault is impossible.

Proof of Theorem 3.2:

In an asynchronous system, every cut is the prefix of an execution with guaranteed eventual communication, and every cut has the isolation property. Moreover, the cuts are exactly the prefixes closed under the $<_L$ relation, so the subset of fault free executions of an asynchronous system forms a subsystem. Further, each primitive extension consists of exactly one action by Axiom A1.

Let S be an asynchronous system for solving a coordination problem that is subject to crash faults. Let G be the subsystem of S of executions with guaranteed eventual communication, at most one crash fault, and no other faults. Let A be the subsystem of G of executions with no faults. Suppose G is both safe and deciding.

Given a multivalent cut x in A, we will show that for each primitive extension a of x, there is an extension e of x such that (1) x+e is multivalent in A, (2) @(e) contains @(a), and (3) if a is the receipt of message m at process p, then e contains an action that is the receipt of message m at process p. Then we can take turns among all the processes and all the outstanding (sent but not yet received) messages to produce an infinite ascending chain of multivalent cuts with guaranteed eventual communication in which all processes take infinitely many actions. But this infinite ascending chain would be an execution of A and would have a deciding finite prefix, contradicting safety. Thus there can be no cut that is multivalent in A. However, in any solution to a nontrivial coordination problem, the empty cut is multivalent; and, for a consensus problem, the empty cut is multivalent in the subsystem of fault free executions. Thus a safe deciding G is impossible for consensus.

So suppose a is an extension of x such that x+a is not multivalent in A. Without loss of generality let us denote by 0 the output action produced by extensions of x+a, and let us denote by 1 the output action corresponding to a disagreeing deciding extension of x. Since x is multivalent and A is deciding, x has an extension that produces output 1. Let p = @(a). If a is a receipt of message m at p, then consider the extension of each cut in the sequence of primitive extensions leading to output 1 by a receipt of message m at p. Either one of these produces the desired multivalent cut x+e or there is some cut y in the sequence of primitive extensions of x such that y is multivalent in A and it has primitive extensions b and c with extensions of y+b in A only producing output 0 and extensions of y+c in A only producing output 1. The two extensions of y would be incompatible so @(b) = @(c) = some process q. Since G is subject to at most one crash fault and deciding, it would contain a deciding extension d of y with no actions at q. By A2, y+d is a cut of A, so y is vulnerable in A and neither A nor G could be safe by Theorem 2.4. Thus the desired extension e must exist. The argument if action a is not a receive action is identical except that any enabled action at p=@(a) can be used to extend a member of the sequence of primitive extensions of x leading to output 1. *QED*

Process q is *blocked* by set of processes P (not containing q) at cut x if x has no output action from process q, there is no extension of x that contains an output action from q unless it also contains an action from some process in P, and there is an infinite ascending chain of extensions of x with an infinite set of actions of q but no actions from processes in P. A system is said to be *blocking* if it contains a cut at which some process is blocked. (When $\leq_K \equiv <_L$, we can restrict the notion of blocking to the existence of a cut at which some process is blocked by a single other process and keep all our results about blocking.)

Corollary 3.3: A safe asynchronous system with guaranteed eventual communication that solves a nontrivial coordination problem when there are no process faults is blocking.

A cut is called *definitive* if it can be extended to two disagreeing deciding cuts y and z such that some process does not fail in either y or z.

Corollary 3.4: No safe asynchronous system with guaranteed eventual communication can be timely with respect to a definitive cut, even when there are no faults.

4. Partially Synchronous Systems.

A system is said to be *timed* if it has an infinite ascending tree of cuts such that each execution contains exactly one chain from the tree and, if $a <_L b$ in some execution, then there is some cut in the chain for that execution that contains a but not b. Such an infinite ascending tree of cuts is called a *timing*. In each execution of a timed system, the timing is an infinite ascending chain of cuts, so we can refer to the ith cut in the timing for any integer i. Thus with respect to a given timing, there is a function t that takes actions to the least integer i such that they are contained in the ith cut. With respect to a given timing, a system is said to have *synchronous processes* if there is a linear increasing function f such that $t(a)<f(s(a))$ for any action a. Note that $s(a)\leq t(a)$ for any timing. Again, with respect to a given timing, a system is said to have *synchronous communication* if there is a constant B such that, if $a \rightarrow b$ then $t(b)-t(a)<B$. A system is said to be *synchronous* with respect to a given timing if it has both synchronous processes and synchronous communication. Conjecture: these notions of partial synchrony are fairly robust. In particular, if a system has synchronous processes with respect to two timings and it has synchronous communication with respect to one of them, then it has synchronous communication with respect to the other.

A *network partition* is a communication failure that divides the set of processes into at least two sets containing processes that have not failed and prevents messages between the sets from being received. An execution in a timed system has a network partition of *duration* j if there is an integer i such that, for any send action a with $t(a)>i$, and any receive action b with $a \rightarrow b$, if @(a) and @(b) are on opposite sides of the partition, then $t(a)>i+j$, that is, no message sent between time i and time i+j is received across the partition.

The following results hold for synchronous systems in which every cut has the isolation property.

Theorem 4.1: Safe coordination that is timely with respect to a definitive cut and must tolerate a single network partition of arbitrary duration is impossible.

Proof of Theorem 4.1: Let P be the set of processes that do not fail in either of the disagreeing deciding runs from the definitive cut. Let A be the subsystem of executions containing the definitive cut in which the processes of P do not fail. Then A is deciding if the system is timely with respect to the definitive cut. Moreover, timeliness implies that there is no infinite ascending chain of multivalent cuts in any execution of A. Thus there must be a final multivalent cut and the system must be vulnerable.
QED

Corollary 4.2: Safe timely atomic commit is impossible in the presence of a possible network partition of arbitrary duration.

Corollary 4.3: Safe atomic commit is blocking.

This result does not contradict the results of [SS]. In [SS], the $\leq_K$ relation differs from the $<_L$ relation: each time a message is sent, the target receives information about actions taken by the sender, whether the message is received or not. Moreover, after a bounded time, the sender receives information about whether the message was received. Thus cuts containing actions taken a bounded time after the send must contain corresponding actions taken at the target. In such systems cuts may not have the isolation property.

However, in more realistic models without some guaranteed synchronous transfer of information, all cuts will have the isolation property and any safe atomic commit protocol must be blocking.

5. Jury: Safe and Probably Timely Atomic Commit.

In this section we provide some positive results for synchronous systems that do not have guaranteed eventual communication. An atomic commit protocol cannot be safe and timely at the same time in the absence of guaranteed synchronous information transfer. The family of protocols we present in this section has two characteristics that distinguish it from the standard two phase commit approach. One is the addition of a set of new processes called jurors to the transaction processing system. The other is the combination of timeout with unilateral requests for more time that are propagated throughout the communication structure associated with a given transaction. The basic idea is to use the jurors in order to decrease the probability of reaching an unsafe state.

The set of jurors is used to replicate the commit decision role usually reserved for a single transaction coordinator process. The timeout/time request combination is used to detect the inability of a process to communicate while not requiring any preknowledge of the time the transaction will actually require to complete processing. Together they can be used to prevent blocking by a small number of faults.

We assume as a base a transaction processing system that can support standard two phase commit. Thus, transactions can be initiated at any site and at any time and each transaction dynamically invokes a set of processes at various sites to do work on its behalf. The set of processes may depend on data at the sites. We also assume a dynamically growing communication structure (typically a tree) that allows all processes doing work for the transaction to communicate with each other, provided there are no faults in the underlying communication media.

We assume that each process knows when it has completed the work on behalf of a given transaction. Any process can unilaterally decide that the work must be aborted (for any reason) until it enters a "prepared" state. Once a process enters the "prepared" state, a process participating in standard two phase commit must wait for instructions from a coordinator on whether to commit or abort the work. Our scheme modifies two phase commit in the following ways.

When a transaction is initiated, a small set of processes called *jurors* is chosen and their names are conveyed to each process invoked to do work on the transaction. (The juror processes are invoked before any nonjuror processes are invoked.) As each juror process is invoked, it sets a timer. For simplicity, we will assume that clocks at all sites are synchronized. (When clocks are not synchronized our proposal must be modified in straightforward ways.) The synchronization need not be very precise when there is some known bound on the precision. We assume that the initiation time of the transaction is conveyed to all processes that work on it. Each juror sets its timer for a known constant time after the initiation time. This constant is chosen to provide sufficient time for most short transactions to complete. (The larger the constant the longer the time required to wait before crashed processes are detected. But if the constant is too small, processes will not have time to communicate with the jury.)

When any nonjuror process is invoked, its existence is communicated to the jury. Each nonjuror process invoked, works on the transaction until one of three events: (1) it decides to unilaterally abort, (2) it completes its work, or (3) current time comes within a known constant threshold of the timeout. In case it decides to abort, it communicates this fact to the jury and ceases to do work on behalf of the transaction. (There are many ways to optimize message traffic here, but we suppress these details in order to convey the main ideas of our protocol.) In case it completes its work, it enters a "prepared" state and communicates this fact to the jury. (We suppress details of possible optimizations. It suffices that no juror can receive a "prepared" communication from a process without knowing the identities of all processes invoked by that process; but we can also arrange that processes act as subordinate coordinators for all processes they invoke and only communicate "prepared" to their invoker(s) when all their subordinates have in turn communicated "prepared" to them.) The time threshold above is chosen to allow any process sufficient time to communicate with all to reset the timeout time. In case it crosses the time threshold, a process resets its timeout's time and sends it to all.

Each juror simply waits for one of the following events: (1) a new timeout time, (2) a unilateral abort, (3) receipt of "prepared" messages from

every process known to be working on the transaction, or (4) a timeout. In case of a new timeout time, the juror resets its timer. In case of a unilateral abort, the juror records in some stable way that it votes to abort the transaction (presumed abort of [MLO] could be used here), communicates this fact to all participants, and ceases to work on behalf of that transaction. In case all known participants have sent "prepared" messages, the juror records in some stable way that it votes to commit the transaction, communicates this fact to all participants, and ceases to do work on behalf of the transaction. In case of a timeout, the juror records in some stable way that it votes to abort the transaction, communicates this fact to all participants, and ceases to do work on behalf of the transaction.

A participant in the "prepared" state waits for one of two events: (1) the receipt of a commit vote from a majority of the jury, or (2) the receipt of an abort vote from a majority of the jury. (If the size of the jury is even, then a third event must be added: (3) receipt of a tied vote with all members of the jury voting.) It acts according to the majority vote from the jury to either commit its part of the transaction or abort it. (In case of the tie vote with all jurors voting, the transaction is aborted.)

The Jury Protocol presented below assumes an upper bound d on the worst case diffusion time and an upper bound e on the precision of clock synchronization. Let D and E be the corresponding worst clock times. Let W be a clock time sufficient for most transaction processing. We assume that when a transaction is started, a set of jurors are chosen and started. Then the nonjurors are dynamically invoked to perform the work. For transaction X, we assume communication primitives send__to__jury(X,Y) and send__to__all(X,Y) by which all participants, juror and nonjuror can send messages to all jurors or to all participants, respectively. When each process is invoked it is given the initiation time of the transaction I(X). From this it computes and maintains a timeout time T(X).

We give two protocols, one for the nonjuror and one for the juror. In the protocol the command "On Learning Y → do Z" means that at the first time event Y or condition Y holds perform Z.

NONJUROR PROTOCOL

```
Do forever;
  On learning
    transaction X initiated at I(X) → do;
      T(X) := I(X)+W+3D+E;
      log X;
      commence work on X;
      end;
    decide abort X → do;
      log abort X;
      send__to__jury(X,abort);
      abort X;
      end;
    work complete for X → do;
      log prepared X;
      send__to__jury(X,prepared);
      decide prepared X;
      end;
    current time > T(X) → do;
          /* doubles timeout time */
      T(X) := 3T(X) - 2I(X);
      send__to__all(X,new__timeout T(X));
      end;
    new__timeout T for X → do;
      T(X) := max(T,T(X));
      end;
    prepared & majority jury commit X → do;
      commit X;
      log commit X;
      end;
    prepared & majority jury abort X → do;
      decide abort X;
      end;
    end;
end.
```

JUROR PROTOCOL

```
Do forever;
   On learning
     transaction X initiated at I(X) → do;
       T(X) := I(X)+W+3D+E;
       log X;
       end;
     new__timeout T for X → do;
       T(X) := max(T,T(X));
       end;
     decide abort X → do;
       log abort X;
       send__to__all(X,abort);
       end;
     all nonjurors prepared X → do;
       log commit X;
       send__to__all(X,commit);
       end;
     current time > T(X)+D+E → do;
       decide abort X;
       end;
   end;
end.
```

The original timeout and threshold values are to be chosen so that processes have sufficient time to contact the jury to request more time. When this condition is violated, the violation is called a timing fault.

Omission faults cannot cause the above scheme to produce inconsistent results. Moreover, when there are no faults, no unilateral aborts by participating processes, and the work of the transaction is completed at all sites, then the transaction will be committed at all sites. On recovery a failed process can resolve the "in doubt" state of a transaction by consulting the jury. Finally, provided the jury has at least $2t+1$ members and at most t component failures, and provided there are no failures other than omission failures, no process that can communicate with all correctly functioning jurors will block the transaction.

To cope with Byzantine failures in the jury, the jury should reach Byzantine consensus on the vote before sending it to the prepared processes. To tolerate Byzantine failures among the processes, each process should be replicated and the replicas should reach Byzantine agreement about the work, preparation, and commitment, and should communicate by majority vote with the jury.

Omission failures in the communication media can cause blocking. However, by arranging communication protocols (such as diffusion) in a multiply connected network, a small number of such communication omission failures can be tolerated without blocking.

In general, the jurors may reside with a subset (or all) of the processes performing the work of the transaction, so long as some implementation is provided for communicating with the jury and knowing the identities of the jurors so that a majority can be recognized.

In case the jury has size 1, our scheme degenerates into a variant of two phase commit; in case the jury has size 2, it resembles schemes that use a backup coordinator (eg.[HS]) and the novelty becomes only the use of timeout and request for more time.

By providing sufficiently many jurors and sufficiently redundant communication facilities, a designer can use this Jury approach to make the probability of blocking extremely small.

6. Conclusion.

When partitions may occur with a positive probability there is no way to guarantee safety and timely operation. In most cases the probability of partition is small even though some components may be inoperative for long periods of time. Blocking prevents transient faults from causing a loss of consistency in the rare instances of partition. Much as we would like a truly nonblocking coordination protocol, Theorem 3.2 implies that guaranteed eventual communication is not enough to guarantee safety without blocking in an asynchronous environment, and Corollary 4.3 says that even in a synchronous environment, any safe atomic commit protocol is blocking.

The probability of network partition depends on the number of faults and the probability of successful transmission of single messages over links. Our jury algorithms are always recoverably safe and they are nonblocking provided synchronous

communication is guaranteed and provided less than half of the jury fails. Three phase commit techniques could be added to take care of excessive jury failure; but since synchronous communication cannot be guaranteed in most environments, the marginal improvement in the probability of blocking would likely be insignificant. The algorithms we have presented are deterministic and require both process and communication synchrony. We can generalize them to probabilistic protocols like those of [BO] that operate in asynchronous systems and require less expected time to complete in timed systems. However, these asynchronous probabilistic protocols must allow for the possibility of blocking. In a synchronous system we can combine protocols so that the expected time is constant, but when the probabilistic algorithm runs too long it switches to a deterministic one, to guarantee termination as soon as any partition is repaired. Presentation of the probabilistic algorithms and various alternatives and optimizations is outside the scope of this paper.

7. Acknowledgements.

The Jury idea was first developed by the authors and C. Mohan as a general method to provide arbitrary fault tolerance for the transaction process. The idea of unifying the various impossibility results for distributed systems arose as a result of subsequent work by the authors on the Jury idea and questions posed by Flaviu Cristian on the possibility of safe timely coordination in synchronous and partially synchronous systems. The authors would like to thank Joe Halpern for helpful comments on an earlier version of this work.

8. References.

[BD] S. Ben-David, "The Global Time Assumption and Semantics for Concurrent Systems," *Proc. 7nd ACM Symp. on PODC*, (1988).

[BO] M. Ben-Or, "Another Advantage of Free Choice: Completely Asynchronous Agreement protocols," *Proc. 2nd ACM Symp. on PODC*, (1983) 27-30.

[BHG] P. A. Bernstein, V. Hadzilacos, and N. Goodman, *Concurrent Control and Recovery in Database Systems,* Addison-Weseley, 1987.

[DS] C. Dwork and D. Skeen, "The Inherent cost of Nonblocking Commitment," *Proc. 2nd ACM Symp. on PODC*, (1983) 1-11.

[FLP] M. J. Fischer, N. A. Lynch and M. S. Paterson, "Impossiblity of Disctributed Consensus with One Faulty Process," *JACM 32* (1985) 373-382.

[G] J. N. Gray, "Notes on Database Operating Systems," *Operating Systems: an advanced course, Lecture Notes in Computer Science 60*,Springer Verlag (1978) 393-481.

[HS] M. Hammer and D. Shipman, "Reliability mechanisms for SDD-1: A system for distributed databases," *ACM Trans. Database Syst. 5*,4 (1980) 431-466.

[L] L. Lamport, "Time Clocks and the Ordering of Events in a Distributed System," *CACM 21*,7 (1978) 558-565.

[MLO] C. Mohan, B. Lindsay, and R. Obermarck, "Transaction Management in the R* Distributed Database Management System," *ACM Trans. Database Syst. 11*,4 (1986) 378-396.

[MSF] C. Mohan, R. Strong, and S. Finkelstein, "Method for distributed transaction commit and recovery using Byzantine agreement within clusters of processors," *Proceedings of the 2nd ACM SIGACT/SIGOPS Symposium on Principles of Distributed Computing*, (1983) 89-103, reprinted in *ACM/SIGOPS Operating Systems Review (1985).*

[NT] G. Neiger and S. Toueg, "Substituting for Real Time and Common Knowledge in Asynchronous Distributed Systems," *Proc. 6nd ACM Symp. on PODC*, (1987) 281-293.

[PT] P. Panangaden and K. Taylor, "Concurrent Common Knowledge: A New Definition of Agreement for Asynchronous Systems," *Proc. 7nd ACM Symp. on PODC*, (1988).

[R] M. Rabin, "Randomized Byzantine Generals," *Proc. of the 24th FOCS Symp.* (1983) 403-409.

[S] D. Skeen, "Nonblocking Commit Protocols," *Proc. ACM Conf. on Management of Data* (1982) 133-147.

[SS] D. Skeen and M. Stonebraker, "A Formal Model of Crash Recovery in a Distributed System," *Proc. 5th Berkely Workshop on Distributed Data Management and Computer Networks* (1981) 129-142.