Brewer's Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services

Seth Gilbert* Nancy Lynch*

Abstract

When designing distributed web services, there are three properties that are commonly desired: consistency, availability, and partition tolerance. It is impossible to achieve all three. In this note, we prove this conjecture in the asynchronous network model, and then discuss solutions to this dilemma in the partially synchronous model.

1 Introduction

At PODC 2000, Brewer[1], in an invited talk [2], made the following conjecture: it is impossible for a web service to provide the following three guarantees:

- Consistency
- Availability
- Partition-tolerance

All three of these properties are desirable – and expected – from real-world web services. In this note, we will first discuss what Brewer meant by the conjecture; next we will formalize these concepts and prove the conjecture;

*Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

[1] Eric Brewer is a professor at the University of California, Berkeley, and the co-founder and Chief Scientist of Inktomi.

finally, we will describe and attempt to formalize some real-world solutions to this practical difficulty.

Most web services today attempt to provide strongly consistent data. There has been significant research designing ACID[2] databases, and most of the new frameworks for building distributed web services depend on these databases. Interactions with web services are expected to behave in a transactional manner: operations commit or fail in their entirety (atomic), transactions never observe or result in inconsistent data (consistent), uncommitted transactions are isolated from each other (isolated), and once a transaction is committed it is permanent (durable). It is clearly important, for example, that billing information and commercial transaction records be handled with this type of strong consistency.

Web services are similarly expected to be highly available. Every request should succeed and receive a response. When a service goes down, it may well create significant real-world problems; the classic example of this is the potential legal difficulties should the E-Trade web site go down. This problem is exacerbated by the fact that a web-site is most likely to be unavailable when it is most needed. The goal of most web services today is to be as available as the network on which they run: if any service on the network is available, then the web service should be accessible.

Finally, on a highly distributed network, it is desirable to provide some amount of fault-tolerance. When some nodes crash or some communication links fail, it is important that the service still perform as expected. One desirable fault tolerance property is the ability to survive a network partitioning into multiple components. In this note we will not consider stopping failures, though in some cases a stopping failure can be modeled as a node existing in its own unique component of a partition.

2 Formal Model

In this section, we will formally define what is meant by the terms *consistent*, *available*, and *partition tolerant*.

2.1 Atomic Data Objects

The most natural way of formalizing the idea of a consistent service is as an atomic data object. Atomic [4], or linearizable [3], consistency is the

[2] **A**tomic, **C**onsistent, **I**solated, **D**urable

condition expected by most web services today.[3] Under this consistency guarantee, there must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time. This is the consistency guarantee that generally provides the easiest model for users to understand, and is most convenient for those attempting to design a client application that uses the distributed service. See Chapter 13 of [5] for a more complete definition of atomic consistency.

2.2 Available Data Objects

For a distributed system to be continuously available, every request received by a non-failing node in the system must result in a response.[4] That is, any algorithm used by the service must eventually terminate. In some ways this is a weak definition of availability: it puts no bound on how long the algorithm may run before terminating, and therefore allows unbounded computation. On the other hand, when qualified by the need for partition tolerance, this can be seen as a strong definition of availability: even when severe network failures occur, every request must terminate.

2.3 Partition Tolerance

The above definitions of availability and atomicity are qualified by the need to tolerate partitions. In order to model partition tolerance, the network will be allowed to lose arbitrarily many messages sent from one node to another. When a network is partitioned, all messages sent from nodes in one component of the partition to nodes in another component are lost. (And any pattern of message loss can be modeled as a temporary partition separating the communicating nodes at the exact instant the message is lost.) The atomicity requirement (§2.1) therefore implies that every response will be atomic, even though arbitrary messages sent as part of the algorithm might not be delivered. The availability requirement (§2.2) implies that

[3] Discussing atomic consistency is somewhat different than talking about an ACID database, as database consistency refers to transactions, while atomic consistency refers only to a property of a single request/response operation sequence. And it has a different meaning than the **A**tomic in ACID, as it subsumes the database notions of both **A**tomic and **C**onsistent.

[4] Brewer originally only required *almost all* requests to receive a response. As allowing probabilistic availability does not change the result when arbitrary failures occur, for simplicity we are requiring 100% availability.

every node receiving a request from a client must respond, even though arbitrary messages that are sent may be lost. Note that this is similar to wait-free termination in a pure shared-memory system: even if every other node in the network fails (i.e. the node is in its own unique component of the partition), a valid (atomic) response must be generated. No set of failures less than total network failure is allowed to cause the system to respond incorrectly.[5]

3 Asynchronous Networks

3.1 Impossibility Result

In proving this conjecture, we will use the asynchronous network model, as formalized by Lynch in Chapter 8 of [5]. In the asynchronous model, there is no clock, and nodes must make decisions based only on the messages received and local computation.

Theorem 1 *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability*
- *Atomic consistency*

in all fair executions (including those in which messages are lost).

Proof: We prove this by contradiction. Assume an algorithm A exists that meets the three criteria: atomicity, availability, and partition tolerance. We construct an execution of A in which there exists a request that returns an inconsistent response. The methodology is similar to proofs in Attiya et al. [1] and Lynch [5] (Theorem 17.6). Assume that the network consists of at least two nodes. Thus it can be divided into two disjoint, non-empty sets: $\{G_1, G_2\}$. The basic idea of the proof is to assume that all messages between G_1 and G_2 are lost. If a *write* occurs in G_1, and later a *read* occurs in G_2, then the *read* operation cannot return the results of the earlier *write* operation.

More formally, let v_0 be the initial value of the atomic object. Let α_1 be the prefix of an execution of A in which a single *write* of a value not equal to

[5]Brewer pointed out in the talk that partitions of one node are irrelevant: they are equivalent to that node failing. However restricting our attention to partitions containing only components of size greater than one does not change any of the results in this note.

v_0 occurs in G_1, ending with the termination of the *write* operation. Assume that no other client requests occur in either G_1 or G_2. Further, assume that no messages from G_1 are received in G_2, and no messages from G_2 are received in G_1. We know that this *write* completes, by the availability requirement. Similarly, let α_2 be the prefix of an execution in which a single *read* occurs in G_2, and no other client requests occur, ending with the termination of the *read* operation. During α_2 no messages from G_2 are received in G_1, and no messages from G_1 are received in G_2. Again we know that the *read* returns a value by the availability requirement. The value returned by this execution must be v_0, as no *write* operation has occurred in α_2.

Let α be an execution beginning with α_1 and continuing with α_2. To the nodes in G_2, α is indistinguishable from α_2, as all the messages from G_1 to G_2 are lost (in both α_1 and α_2, which together make up α), and α_1 does not include any client requests to nodes in G_2. Therefore in the α execution, the *read* request (from α_2) must still return v_0. However the *read* request does not begin until after the *write* request (from α_1) has completed. This therefore contradicts the atomicity property, proving that no such algorithm exists. ■

Corollary 1.1 *It is impossible in the asynchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability, in all fair executions,*
- *Atomic consistency, in fair executions in which* no *messages are lost.*

Proof: The main idea is that in the asynchronous model an algorithm has no way of determining whether a message has been lost, or has been arbitrarily delayed in the transmission channel. Therefore if there existed an algorithm that guaranteed atomic consistency in executions in which no messages were lost, then there would exist an algorithm that guaranteed atomic consistency in all executions. This would violate Theorem 1.

More formally, assume for the sake of contradiction that there exists an algorithm A that always terminates, and guarantees atomic consistency in fair executions in which all messages are delivered. Further, Theorem 1 implies that A does not guarantee atomic consistency in all fair executions, so there exists some fair execution α of A in which some response is not atomic.

At some finite point in execution α, the algorithm A returns a response that is not atomic. Let α' be the prefix of α ending with the invalid response.

Next, extend α' to a fair execution α'', in which all messages are delivered. The execution α'' is now a fair execution in which all messages are delivered. However this execution is not atomic. Therefore no such algorithm A exists. ■

3.2 Solutions in the Asynchronous Model

While it is impossible to provide all three properties: atomicity, availability, and partition tolerance, any two of these three properties can be achieved.

3.2.1 Atomic, Partition Tolerant

If availability is not required, then it is easy to achieve atomic data and partition tolerance. The trivial system that ignores all requests meets these requirements. However we can provide a stronger liveness criterion: if all the messages in an execution are delivered, the system is available and all operations terminate. A simple centralized algorithm meets these requirements: a single designated node maintains the value of an object. A node receiving a request forwards the request to the designated node, which sends a response. When an acknowledgment is received, the node sends a response to the client.

Many distributed databases provide this type of guarantee, especially algorithms based on distributed locking or quorums: if certain failure patterns occur, then the liveness condition is weakened and the service no longer returns responses. If there are no failures, then liveness is guaranteed.

3.2.2 Atomic, Available

If there are no partitions, it is clearly possible to provide atomic, available data. In fact, the centralized algorithm described in Section 3.2.1 meets these requirements. Systems that run on intranets and LANs are an example of these types of algorithms.

3.2.3 Available, Partition Tolerant

It is possible to provide high availability and partition tolerance, if atomic consistency is not required. If there are no consistency requirements, the service can trivially return v_0, the initial value, in response to every request. However it is possible to provide weakened consistency in an available, partition tolerant setting. Web caches are one example of a weakly consistent

network. In Section 4.4 we consider one of the possible weaker consistency conditions.

4 Partially Synchronous Networks

4.1 Partially Synchronous Model

The most obvious way to try to circumvent the impossibility result of Theorem 1 is to realize that in the real world, most networks are not purely asynchronous. If you allow each node in the network to have a clock, it is possible to build a more powerful service.

For the rest of this paper, we will assume a partially synchronous model in which every node has a clock, and all clocks increase at the same rate. However, the clocks themselves are not synchronized, in that they may display different values at the same real time. In effect, the clocks act as timers: local state variables that the processes can observe to measure how much time has passed. A local timer can be used to schedule an action to occur a certain interval of time after some other event. Furthermore, assume that every message is either delivered within a given, known time: t_{msg}, or it is lost. Also, every node processes a received message within a given, known time: t_{local}, and local processing takes zero time. This can be formalized as a special case of the General Timed Automata model described by Lynch in Chapter 23 of [5].

4.2 Impossibility Result

It is still impossible to have an always available, atomic data object when arbitrary messages may be lost, even in the partially synchronous model. That is, the following analogue of Theorem 1 holds:

Theorem 2 *It is impossible in the partially synchronous network model to implement a read/write data object that guarantees the following properties:*

- *Availability*
- *Atomic consistency*

in all executions (even those in which messages are lost).

Proof: This proof is rather similar to the proof of Theorem 1. We will follow the same methodology: divide the network into two components, $\{G_1, G_2\}$, and construct an admissible execution in which a *write* happens

in one component, followed by a *read* operation in the other component. This *read* operation can be shown to return inconsistent data.

More formally, construct execution α_1 as before in Theorem 1: a single *write* request and acknowledgment occur in G_1, and all messages between the two components, $\{G_1, G_2\}$, are lost. We will construct the second execution, α_2', slightly differently. Let α_2' be an execution that begins with a long interval of time during which no client requests occur. This interval must be at least as long as the entire duration of α_1. Then append to α_2' the events of α_2, as defined above in Theorem 1: a single *read* request and response in G_2, again assuming all messages between the two components are lost. Finally, construct α by superimposing the two executions α_1 and α_2'. The long interval of time in α_2 ensures that the *write* request completes before the *read* request begins. However, as in Theorem 1, the *read* request returns the initial value, rather than the new value written by the *write* request, violating atomic consistency. ■

4.3 Solutions in the Partially Synchronous Model

In the partially synchronous model, however, the analogue of Corollary 1.1 does not hold. The proof of this corollary does in fact depend on nodes being unaware of when a message is lost. There are partially synchronous algorithms that will return atomic data when all messages in an execution are delivered (i.e., there are no partitions), and will only return inconsistent (and, in particular, stale) data when messages are lost. One example of such an algorithm is the centralized protocol described in Section 3.2.1, modified to time-out lost messages. On a *read* (or *write*) request, a message is sent to the central node. If a response from the central node is received, then the node delivers the requested data (or an acknowledgment). If no response is received within $2 * t_{msg} + t_{local}$, then the node concludes that the message was lost. The client is then sent a response: either the best known value of the local node (for a *read* operation), or an acknowledgment (for a *write* operation). In this case, atomic consistency may be violated.

4.4 Weaker Consistency Conditions

While it is useful to guarantee that atomic data will be returned in executions in which all messages are delivered (within some time bound), it is equally important to specify what happens in executions in which some of the messages are lost. In this section, we will discuss one possible weaker consistency condition that allows stale data to be returned when there are

partitions, yet still place formal requirements on the quality of the stale data returned. This consistency guarantee will require availability and atomic consistency in executions in which no messages are lost, and is therefore impossible to guarantee in the asynchronous model as a result of Corollary 1.1.

In the partially synchronous model it often makes sense to base guarantees on how long an algorithm has had to rectify a situation. This consistency model ensures that if messages are delivered, then eventually some notion of atomicity is restored.

In an atomic execution, we would define a partial order of the *read* and *write* operations, and then require that if one operation begins after another one ends, the former does not precede the latter in the partial order. We will define a weaker guarantee, t-Connected Consistency, which defines a partial order in a similar manner, but only requires that one operation not precede another if there is an interval between the operations in which all messages are delivered.

Definition 3 *A timed execution, α, of a read-write object is* t-Connected Consistent *if two criteria hold. First, in executions in which no messages are lost, the execution is atomic. Second, in executions in which messages are lost, there exists a partial order P on the operations in α such that:*

1. *P orders all* write *operations, and orders all* read *operations with respect to the* write *operations.*
2. *The value returned by every* read *operation is exactly the one written by the previous* write *operation in P, or the initial value, if there is no such previous* write *in P.*
3. *The order in P is consistent with the order of* read *and* write *requests submitted at each node.*
4. *Assume there exists an interval of time longer than t in which no messages are lost. Further, assume an operation, θ, completes before the interval begins, and another operation, ϕ, begins after the interval ends. Then ϕ does not precede θ in the partial order P.*

This guarantee allows for some stale data when messages are lost, but provides a time limit on how long it takes for consistency to return, once the partition heals. This definition can of course be generalized to provide consistency guarantees when only some of the nodes are connected, and

when connections are available only some of the time. These generalizations will be further examined in future work.

A variant of the centralized algorithm described in Section 4.3 is t-Connected Consistent. Assume node C is the centralized node. The algorithm behaves as follows:

- *read* at node A:
  A sends a request to C for the most recent value. If A receives a response from C within time $2 \cdot t_{msg} + t_{local}$, it saves the value and returns it to the client. Otherwise, A concludes that a message was lost and it returns the value with the highest sequence number that has ever been received from C, or the initial value if no value has yet been received from C. (When a client *read* request occurs at C, it acts like any other node, sending messages to itself.)

- *write* at A:
  A sends a message to C with the new value. A waits $2 \cdot t_{msg} + t_{local}$, or until it receives an acknowledgment from C, and then sends an acknowledgment to the client. At this point, either C has learned of the new value, or a message was lost, or both events occurred. If A concludes that a message was lost, it periodically retransmits the value to C (along with all values lost during earlier *write* operations) until it receives an acknowledgment from C. (As in the case of *read* operations, when a client *write* request occurs at C, it acts like any other node, sending messages to itself.)

- New value is received at C:
  C serializes the *write* requests that it hears about by assigning them consecutive integer tags. Periodically C broadcasts the latest value and sequence number to all other nodes.

Theorem 4 *The modified centralized algorithm is t-Connected consistent.*

Proof: First, it is clear that in executions in which no messages are lost, the operations are atomic. An execution is atomic if every operation acts as if it is executed at a single instant; in this case, that single instant occurs when C processes the operation. C serializes the operations, ensuring atomic consistency in executions in which all messages are delivered.

Next, we examine executions in which messages are lost. The partial order, P is constructed as follows. *Write* operations are ordered by the

sequence number assigned by the central node. Each *read* operation is sequenced after the *write* operation whose value it returns. It is clear by the construction that the partial order P satisfies criteria 1 and 2 of the definition of t-Connected consistency. As the algorithm handles requests in the order received, criterion 3 is also clearly true.

In showing that the partial order respects criterion 4, there are four cases: *write* followed by *read*, *write* followed by *write*, *read* followed by *read*, and *read* followed by *write*. Let time t be long enough for a *write* operation to complete (and for C to assign a sequence number to the new value), and for one of the periodic broadcasts from C to occur.

1. *write* followed by *read*
   Assume a *write* occurs at A_w, after which an interval of time longer than t passes in which all messages are delivered. After this, a *read* is requested at some node. By the end of the interval, two things have happened. First, A_w has notified the central node of the new value, and the *write* operation has been assigned a sequence number. Second, the central node has rebroadcast that value (or a later value in the partial order) to all other nodes during one of the periodic broadcasts. As a result, the *read* operation does not return an earlier value, and therefore it must come after the *write* in the partial order P.

2. *write* followed by *write*
   Assume a *write* occurs at A_w, after which an interval of time longer than t passes in which all messages are delivered. After this, a *write* is requested at some node. As in the previous case, by the end of the interval in which messages are delivered, the central node has assigned a sequence number to the *write* operation at A_w. As a result, the later *write* operation is sequenced by the central node after the first *write* operation. Therefore the second *write* comes after the first *write* in the partial order P.

3. *read* followed by *read*
   Assume a *read* operation occurs at B_r, after which an interval of time longer than t passes in which all messages are delivered. After this, a *read* is requested at some node. Let ψ be the *write* operation whose value the first *read* operation at B_r returns. By the end of the interval in which messages are delivered, the central node has assigned a sequence number to ψ, and has broadcast the value of ψ (or a later value in the partial order) to all other nodes. As a result, the second

read operation does not return a value earlier in the partial order than ψ. Therefore the second *read* operation does not precede the first in the partial order P.

4. *read* followed by *write*
   Assume a *read* operation occurs at B_r, after which an interval of time longer than t passes in which all messages are delivered. After this, a *write* is requested at some node. Let ψ be the *write* operation whose value the first *read* operation at B_r returns. By the end of the interval in which messages are delivered, the central node has assigned a sequence number to ψ, and as a result all *write* operations beginning after the interval are serialized after ψ. Therefore the *write* operation does not precede the *read* operation in the partial order P.

Therefore, P satisfies criterion 4 of the definition, and this algorithm is t-Connected Consistent. ■

5 Conclusion

In this note, we have shown that it is impossible to reliably provide atomic, consistent data when there are partitions in the network. It is feasible, however, to achieve any two of the three properties: consistency, availability, and partition tolerance. In an asynchronous model, when no clocks are available, the impossibility result is fairly strong: it is impossible to provide consistent data, even allowing stale data to be returned when messages are lost. However in partially synchronous models it is possible to achieve a practical compromise between consistency and availability. In particular, most real-world systems today are forced to settle with returning "most of the data, most of the time." Formalizing this idea and studying algorithms for achieving it is an interesting subject for future theoretical research.

Acknowledgments

We thank Eric Brewer for his interesting PODC talk, for providing us with his talk slides and notes, and for encouraging us in writing this note. We also thank Charles Leiserson for suggesting this problem and for interesting and helpful discussions.

References

[1] Hagit Attiya, Amotz Bar-Noy, Danny Dolev, Daphne Koller, David Peleg, and Rüdiger Reischuk. Achievable cases in an asynchronous environment. In *28th Annual Symposium on Foundations of Computer Science*, pages 337–346, Los Angeles, California, October 1987.

[2] Eric A. Brewer. Towards robust distributed systems. (Invited Talk) *Principles of Distributed Computing*, Portland, Oregon, July 2000.

[3] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[4] Leslie Lamport. On interprocess communication – parts I and II. *Distributed Computing*, 1(2):77–101, April 1986.

[5] Nancy Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.