Proving PACELC

WOJCIECH GOLAB, University of Waterloo, Canada

Scalable distributed systems face inherent trade-offs arising from the relatively high cost of exchanging information between computing nodes. Brewer’s CAP (Consistency, Availability, Partition-Tolerance) principle states that when communication becomes impossible between isolated parts of the system (i.e., the network is partitioned), then the system must give up either safety (i.e., sometimes return an incorrect result) or liveness (i.e., sometimes fail to produce a result). Abadi generalized Brewer’s principle by defining the PACELC (if Partition then Availability or Consistency, Else Latency or Consistency) formulation, which captures the observation that the trade-off between safety and liveness is often made in practice even while the network is reliable. Building on Gilbert and Lynch’s formal proof of the CAP principle, this paper presents a formal treatment of Abadi’s formulation and connects this result to a body of prior work on latency bounds for distributed objects.

ACM Reference Format:

1 INTRODUCTION

Designers and implementers of distributed systems suffer many headaches over failures, concurrency, and also physical limits related to the exchange of information between components. Notably, the propagation delay for communication between data centers grows linearly with distance due to the finite propagation speed of light, which makes it difficult to build systems that are scalable in a geographic sense, and yet maintain low latency for practical workloads. To make matters worse, failures of wide-area network links can partition a system into components that can communicate internally but cannot communicate with each other. In this scenario, many systems are unable to fulfill their goals, which can be categorized broadly as ensuring safety (e.g., never producing incorrect outputs) and liveness (e.g., producing outputs eventually for all requests). Brewer’s CAP principle summarizes this observation by stating that the combination of Consistency (safety), Availability (liveness), and Partition tolerance are not achievable simultaneously.

Following Brewer’s keynote speech at PODC 2000 [3], the CAP Principle became the subject of lively discussion, raising questions such as how to define consistency and availability precisely, and how to categorize existing systems according to which correctness property they sacrifice to overcome the conjectured impossibility. This has led to some confusion, for example the “two out of three” interpretation of CAP, which treats C, A and P symmetrically and suggests that every system can be classified as AP, CP, or CA. In fact some systems (e.g., Apache Cassandra) can be tuned to provide either AP, CP, or none of the above. Moreover, the interpretation of CA (i.e., consistent and available but not partition tolerant) is questionable because lacking P seems to imply that either C or A is lost in the event of a partition, unless perhaps the system is not distributed to begin with, in which case it tolerates partitions trivially. Abadi re-visited the CAP principle by raising two technical points in his 2012 article [1]: (i) no trade-off is necessary at times when the network is reliable, meaning that an AP or CP system may in fact provide both C and A most of the time; and (ii) many practical systems sacrifice C to reduce latency.
irrespective of network failures. This observation is known as Abadi’s PACE formulation: if Partition then Availability or Consistency, Else Latency or Consistency. This formulation distinguishes P from A and C, thus discouraging the “two out of three” interpretation, and also separates the inherent C-A trade-off during a network partition from the voluntary L-C trade-off exercised during failure-free operation.

In parallel with efforts by practitioners to finesse the interpretation of CAP and related trade-offs, the theory community has sought to formalize these observations as mathematical facts. Two years following Brewer’s keynote, Gilbert and Lynch [5] brought rigor to the discussion by formalizing CAP as the impossibility of simulating a read/write register in a message passing system that simultaneously guarantees Lamport’s atomicity property [7] (consistency) and eventual termination (availability) in the presence of a network partition (arbitrary message loss). This result is commonly referred to as the CAP theorem, and is distinguished from Brewer’s informal and intuitively appealing conjecture. Naturally, the proof of the CAP theorem also validates PAC, or the first half of Abadi’s PACE formulation.

Building on the formal model adopted by Gilbert and Lynch [5], this paper aims to present a rigorous treatment of Abadi’s PACE formulation by applying and alternative proof technique based on latency bounds for shared objects. Specifically, the paper makes the following contributions:

- Section 3 discusses known latency bounds for shared objects in partly synchronous systems [2, 8], and proves an analogous bound for the asynchronous model.
- Section 4 establishes a connection between latency bounds for shared objects and CAP-related trade-offs by using the lower bound established in Section 3 to derive an alternative proof of the CAP theorem.
- Section 5 states a formal interpretation of Abadi’s PACE formulation in terms of the results presented in Sections 3 and 4.

2  FORMAL MODEL

I/O automata and their composition. Similarly to [5], this paper adopts the asynchronous system model formalized by Lynch in Chapter 8 of [9]. There are n reliable processes that communicate using point-to-point FIFO (first-in first-out) communication channels. Two varieties of such channels are considered in different parts of this paper: reliable channels that may delay messages but guarantee eventual delivery, and unreliable channels that may drop messages entirely. Both processes and channels are modeled as I/O (input/output) automata, and their composition is an automaton A representing a system that simulates a single read/write register. Process automata are denoted $P_1, \ldots, P_n$, and the channel automaton by which $P_i$ sends messages to $P_j$, $i \neq j$ is denoted $C_{i,j}$. Processes interact with channels using send and receive actions on messages. Processes also support two types of output actions, invoke and respond, by which they initiate and (respectively) compute the result of an operation on the simulated read/write register.

Executions and traces. The behavior of the system in a given run is modeled as an execution, which is an alternating sequence of states an actions, beginning with a start state determined by the initial value of the simulated register. A trace corresponding to an execution $\alpha$, denoted $\text{trace}(\alpha)$, is the subsequence of invoke and respond actions (i.e., external actions) in $\alpha$. The projection of an execution $\alpha$ (respectively trace $\beta$) onto the actions of a particular process $P_i$ is denoted $\alpha|P_i$ (respectively $\beta|P_i$). We assume that every execution is well-formed, meaning that each process executes an alternating sequence of invoke and respond actions, starting with invoke. The invoke action is enabled for each process in every start state, and also in every state where the last output action of the process was a respond.

1 Abadi explains that latency is “arguably the same thing” as availability since a network that loses messages is indistinguishable from one that delays message delivery indefinitely [1]. Thus, $L$ is in some sense synonymous with $A$.

2 The send action is an output action of each process automaton, and an internal action of the composed automaton $A$. This is accomplished by hiding send actions in $A$. 
Fairness and timing. An execution is fair if every process or channel automaton that is enabled to execute an action eventually either executes this action or ceases to be enabled to execute it. In this context, fairness means that every message sent is eventually either dropped or received, and every process eventually invokes another read or write operation if it has computed the response of its previous operation. Thus, a fair execution may in principle have a finite trace if the protocol becomes stuck with no actions enabled. Executions are timed, meaning that each event (occurrence of an action) is tagged with a timestamp from a global clock. This makes it possible to quantify the time taken for a channel to deliver a message in an execution (time of receive minus time of send, or else ∞ if receive does not occur), or the latency of a read or write operation (time or respond minus time of invoke, or else ∞ if respond does not occur).

Correctness properties. An execution α of the system automaton A is consistent if trace(α) satisfies Lamport’s atomicity property for read/write registers [7] (a special case of Herlihy and Wing’s linearizability property [6]), whose formalization is discussed in Chapter 13 of [9]. Quoting [5], atomicity is explained informally as follows:

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.

For the impossibility results presented in this paper, it suffices to adopt a weaker notion of consistency based on Lamport’s regularity property, which is easier to formalize. In the single-writer case, it states that a read must return either the value assigned by the last write preceding it in the execution, or the value assigned by some write that is concurrent with the read.

An execution α of the system automaton A is available if for every process P, any invocation action of P is eventually followed by a respond action of P (i.e., every operation invoked eventually produces a response).

An execution α of the system automaton A is partition-free if for every message m sent, the send action for m is eventually followed by a receive action for m (i.e., all messages sent are delivered eventually).

3 LATENCY BOUNDS

Prior work on simulating read/write registers in a message passing model has established bounds on operation latencies. Informally speaking, these results observe that r + w ≥ d where r and w are upper bounds on the latencies of reads and writes, respectively, and d is a lower bound on the network delay. This point was first proved by Lipton and Sandberg for the coherent random access machine (CRAM) model [8], and then formalized and strengthened by Attiya and Welch for sequential consistency [2]. Both results assume partly synchronous models, and therefore neither can be applied directly in this paper because the worst-case latencies of reads and writes are unbounded in the model defined in Section 2 due to asynchrony. In fact, the upper bounds r and w do not exist if one considers all possible executions of a system, or even all fair executions. This statement remains true even if message delays are constant (i.e., messages are delivered and processed in a timely manner) because the processes are asynchronous. For example, a process that is enabled to send a message may take an arbitrarily long time to transfer that message to a communication channel.

The known lower bound on worst-case operation latency can be recast in the asynchronous model as a lower bound over a special subset of executions. As stated in Theorem 3.1, the lower bound is asserted universally for all executions in the special subset, and implies that operation latency greater than half of the minimum message delay is inherent in the protocol rather following from asynchrony alone.

---

3 The global clock is introduced to simplify analysis, and in this version of the model processes do not have access to the clock.

4 Operation op₂ precedes operation op₁ if op₁ has a respond action and its timestamp is less than the timestamp of the invoke action of op₂.

5 Operation op₁ is concurrent with operation op₂ if neither op₁ precedes op₂ nor op₂ precedes op₁.

6 It is assumed without loss of generality that all messages sent are distinct.
**Theorem 3.1.** Let $A$ be an automaton that simulates a read/write register initialized to value $v_0$ in the asynchronous system model with at least two processes. Suppose that every execution of $A$ is consistent. Let $\alpha$ be any execution of $A$ that is available, comprises a write by some process $P_W$ of some value $v_1 \neq v_0$ and a read by some other process $P_R$, and where the two operations are concurrent. Let $r$ and $w$ denote the latencies of the read and write in $\alpha$, respectively, and let $d > 0$ be a lower bound on the message delay. Then $r + w \geq d$.

**Proof.** Since every execution of $A$ is assumed to be consistent, it follows that the read returns either $v_0$ or $v_1$. Therefore, the following case analysis is exhaustive.

**Case 1:** The read returns $v_1$.

First, note that the invoke action of $P_W'$'s write causally precedes\(^7\) the respond action of $P_R'$'s read, which implies that $P_W$ communicates with $P_R$ either directly or indirectly (i.e., by way of one or more other processes) in $\alpha$. This is because $\alpha$ is otherwise indistinguishable to $P_R$ and $P_W$ from an execution where the events are shifted so that $P_R$'s read responds before $P_W$'s write begins (i.e., $v_1$ is read before $v_1$ is written), which would contradict the assumption that all executions of $A$ are consistent. The causal relationship implies that $\alpha$ contains a set of messages $m_1, m_2, \ldots, m_k$ for some $k \geq 1$, such that $P_W$ sends $m_i$ during its write, $P_R$ receives $m_k$ during its read, and for $1 \leq i < k$ the recipient of $m_i$ sends $m_{i+1}$ after receiving $m_i$. Such a scenario is illustrated in Figure 1 in the special case where $k = 1$. Since the write begins before $m_1$ is sent and the read finishes after $m_k$ is received, it follows that the invoke action of $P_W$'s write is separated from the respond action of $P_R$'s read by $kd$ time. Moreover, since the two operations are assumed to be concurrent, it also follows that the sum of their latencies is at least $kd$. Since $k \geq 1$, this implies $r + w \geq d$, as required.

![Fig. 1. Execution $\alpha$ in the proof of Theorem 3.1.](image_url)

**Case 2:** The read returns $v_0$.

The analysis is similar to Case 1. First, note that the invoke action of $P_R$'s read causally precedes the respond action of $P_W$'s write, which implies that $P_R$ communicates with $P_W$ either directly or indirectly in $\alpha$. This is because $\alpha$ is otherwise indistinguishable to $P_R$ and $P_W$ from an execution where the events are shifted so that $P_W$'s write responds before $P_R$'s reads begins (i.e., $v_0$ is read after $v_1$ is written), which would contradict the assumption that all executions of $A$ are consistent. The causal relationship implies that $\alpha$ contains a set of messages $m_1, m_2, \ldots, m_k$ for some $k \geq 1$, such that $P_R$ sends $m_i$ during its read, $P_W$ receives $m_k$ during its write, and for $1 \leq i < k$ the recipient of $m_i$ sends $m_{i+1}$ after receiving $m_i$. Since the read begins before $m_1$ is sent and the write finishes after $m_k$ is received.

---

\(^7\) The "causally precedes" relation is the transitive closure of two rules: action $a$ causally precedes action $b$ if $a$ occurs before $b$ it the same process, or of $a$ sends a message that is received in $b$. 

---

\[\text{Vol. 1, No. 1, Article . Publication date: April 2018.}\]
\( m_k \) is received, it follows that the \textit{invoke} action of \( P_k \)'s read is separated from the \textit{respond} action of \( P_w \)'s write by \( k \) message delays or \( kd \) time. Moreover, since the two operations are assumed to be concurrent, it also follows that the sum of their latencies is at least \( kd \). Since \( k \geq 1 \), this implies \( r + w \geq d \), as required. \( \square \)

The proof of Theorem 3.1 considers two operations on a single read/write register, as opposed to [2, 8] where a weaker four operations on two registers are considered. This is a consequence of the different interpretations of consistency: this paper deals with atomicity and regularity, which assume that \textit{invoke} and \textit{respond} actions are totally ordered; [2, 8] deal with sequential consistency, which assumes that such actions are only partially ordered (by program order and the "reads-from" relation).

4 FROM LATENCY BOUNDS TO CAP

The CAP principle in the context of the model from Section 2 is stated formally in Theorem 4.1 below, which is modeled after Theorem 1 in [5].

**Theorem 4.1 (CAP).** Let \( A \) be an automaton that simulates a read/write register in the asynchronous system model with at least two processes. Then \( A \) cannot satisfy both of the following properties in every fair execution \( \alpha \) (including executions that are not partition-free):

- \( \alpha \) is consistent
- \( \alpha \) is available

 Gilbert and Lynch prove their version of Theorem 4.1 by contradiction, supposing initially that \( A \) ensures both consistency and availability. They construct an execution \( \alpha \) involving at least two processes, initially partitioned into two disjoint groups \( \{G_1, G_2\} \) that are unable to communicate with each other due to dropped messages. Letting \( v_0 \) denote the initial value of the read/write register, some process in \( G_1 \) is chosen to invoke a write operation that assigns a new value \( v_1 \neq v_0 \). Since \( A \) ensures that \( \alpha \) is available, even if it is not partition-free, this write produces a response eventually. Next, a process in \( G_2 \) is chosen to invoke a read operation, which once again produces a response eventually. However, since there is no communication between processes in \( G_1 \) and processes in \( G_2 \), \( \alpha \) is indistinguishable to processes in \( G_2 \) from an execution where the write never occurs. Therefore, the read must return \( v_0 \) instead of \( v_1 \), which contradicts the assumption that \( \alpha \) is consistent in addition to being available.

Alternatively, Theorem 4.1 can be proved using the latency bound from Section 3. Roughly speaking, the proof argues that if a system ensures consistency then operation latencies grow with message delays, and hence operations cannot terminate eventually (i.e., system cannot ensure availability) if the network is partitioned.

**Alternative proof of Theorem 4.1.** Let \( A \) be an automaton that simulates a read/write register in the asynchronous system model with at least two processes. Suppose for contradiction that \( A \) ensures that every fair execution is both consistent and available, even if the execution is not partition-free. Let \( P_w \) and \( P_r \) be distinct processes, and suppose that the network drops all messages. There exists a fair execution \( \alpha_1 \) of \( A \) where the initial value of the register is \( v_0 \), then \( P_w \) writes \( v_1 \neq v_0 \), then \( P_r \) immediately reads the register (i.e., \( P_r \)'s \textit{invoke} action is consecutive with \( P_w \)'s \textit{respond} action) and produces a response. Since \( A \) ensures consistency even if the execution is not partition-free, this implies that \( P_r \)'s read returns \( v_1 \) and not \( v_0 \). Now let \( \alpha_2 \) be the prefix of \( \alpha_1 \) ending in the state immediately following the read’s response. Since \( \alpha_2 \) is indistinguishable to all processes from an execution where the messages are merely delayed and not dropped, it is possible to extend \( \alpha_2 \) to a finite partition-free execution \( \alpha_3 \) by delivering all sent messages eventually (after the response of the read), without introducing any additional read or write operations. Now construct \( \alpha_4 \) from \( \alpha_3 \) by swapping the relative order of \( P_r \)'s \textit{invoke} action and \( P_w \)'s \textit{respond} action, which preserves the property that the execution is both consistent and available, and also makes Theorem 3.1 applicable. Let \( w \) and \( r \) denote the latencies of the write and read,
respectively, in $\alpha_4$. Suppose without loss of generality that the message delay $d$ in $\alpha_4$ is constant and greater than $r + w$, which ensures that no message sent by $P_W$ after starting its write can influence the outcome of $P_R$’s read operation. This scenario is illustrated in Figure 2 in the simplified case when $P_W$ and $P_R$ are the only two processes in the system. Then $\alpha_4$ contradicts Theorem 3.1 since this execution is both available and consistent with $d > r + w$. 

\begin{figure}[h]
\centering
\includegraphics[width=0.7\textwidth]{fig2.png}
\caption{Execution $\alpha_4$ in the alternative proof of Theorem 4.1.}
\end{figure}

5 FORMAL INTERPRETATION OF PACELC

The conjunction of Theorem 4.1 and Theorem 3.1, both of which are proved in this paper using latency arguments in an asynchronous model, constitutes a formal statement of Abadi’s PACELC formulation. Theorem 4.1 implies that for executions that are fair and not partition-free, the system cannot always guarantee both consistency and availability: if Partition then Availability or Consistency. On the other hand, Theorem 3.1 implies that for executions that are partition-free, the system cannot always guarantee both consistency and operation latency less than half of the minimum message delay, irrespective of asynchrony (i.e., even if message delays are constant and processing delays\(^8\) are zero): Else Consistency or Latency.

Attiya and Welch [2] proved that the latency lower bound $r + w \geq d$ stated in Theorem 3.1 is tight in a partially synchronous model where processes have access to local clocks that can be used as timers, and where message delays are constant. Specifically, if message delays are exactly $d$ (which implies partition-freedom), then there exists a protocol that guarantees atomic consistency and where either reads are instantaneous and writes have latency $d$, or reads have latency $d$ and writes are instantaneous. Such protocols maintain a copy of the register’s state at each process, and use timed delays to compensate for message delays. For example, in the instantaneous read protocol, a read operation returns the local copy of the state without any network communication, whereas a write operation first broadcasts the new value to all other processes, then sleeps for $d$ time, and finally updates its local state. A process updates its local copy of the state instantaneously upon receiving the broadcast value.

Practical distributed storage systems such as Amazon’s Dynamo [4] and its open-source derivatives are designed to operate in a failure-prone environment, and therefore rely on explicit acknowledgments rather than timed delays to ensure delivery (i.e., receipt and processing) of messages between processes. As a result, these systems exhibit operation latencies exceeding the lower bound in Theorem 3.1 by a factor of at least two even

---

\(^8\) In the asynchronous model with timed executions, one can define processing delay as the time between when a send, receive, or respond action is enabled and when that action is executed.
in executions where message delays are exactly $d$. For example, a quorum-replicated system such as Amazon’s Dynamo [4] can be configured for local reading but then writing requires a full network round trip, or $2d$ time, to ensure Lamport’s regularity property [7]. This is accomplished using full replication and a read-one, write-all quorum configuration. Operation latency is increased further if the system is configured to tolerate server failures, for example by using majority quorums, in which case both reads and writes require at least $2d$ time.

6 CONCLUSION

This paper presented both an alternative proof of the CAP principle and a formal treatment of Abadi’s PACELC formulation based on the inherent trade-off between operation latency and network delay. These results complement and extend the CAP theorem of Gilbert and Lynch, which was published prior to Abadi’s article, and draw a precise connection between CAP-related trade-offs and latency bounds for shared objects.

REFERENCES