SAFETY, LIVENESS, AND CONSISTENCY

Ellis Michael
How Do We Specify Distributed Systems?

- **Execution**: Sequence of events (i.e., steps taken by the system), potentially infinite.

- **Property**: A predicate on executions.

- **Safety property**: Specifies the "bad things" that shouldn't happen in any execution.

- **Liveness property**: Specifies the "good things" that should happen in every execution.

(See paper for formal definitions.)
**Theorem:** Every property is expressible as the conjunction of a safety property and a liveness property.

[Alpern and Schneider: 1987]
**Theorem:** Every property is expressible as the conjunction of a safety property and a liveness property.

[Alpern and Schneider. 1987]
Some Properties

• The system never deadlocks.
• Every client that sends a request eventually gets a reply.
• Both generals attack simultaneously.
More Properties: Consensus

$n$ processes, all of which have an input value from some domain. Processes output a value by calling $\text{decide}(v)$. Non-faulty processes continue correctly executing protocol steps forever. We usually denote the number of faulty processes $f$.

- **Agreement**: No two correct processes decide different values.
- **Integrity**: Every correct process decides at most one value, and if a correct process decides a value $v$, some process had $v$ as its input.
- **Termination**: Every correct process eventually decides a value.
**Consistency is Key!**

**Consistency:** the allowed semantics (return values) of a set of operations to a data store or shared object.

Consistency properties specify the **interface**, not the **implementation**. The data might be replicated, cached, disaggregated, etc. "Weird" consistency semantics happen all over the stack!

**Anomaly:** violation of the consistency semantics
TERMINOLOGY: STRENGTH AND WEAKNESS

- **Strong consistency**: the system behaves as if there's just a single copy of the data (or almost behaves that way).

  The intuition is that things like caching and sharding are implementation decisions and shouldn't be visible to clients.

- **Weak consistency**: allows behaviors significantly different from the single store model.

- **Eventual consistency**: the aberrant behaviors are only temporary.
Why the Difference?

- **Performance**
  - Consistency requires synchronization/coordination when data is replicated
  - Often slower to make sure you always return right answer

- **Availability**
  - What if client is offline, or network is not working?
  - Weak/eventual consistency may be only option

- **Programmability**
  - Weaker models are harder to reason against
Lamport’s Register Semantics

Registers hold a single value. Here, we consider single-writer registers only supporting write and read.

Semantics defined in terms of the real-time beginnings and ends of operations to the object.

- **safe**: a read not concurrent with any write obtains the previously written value
- **regular**: safe + a read that overlaps a write obtains either the old or new value
- **atomic**: safe + reads and writes behave as if they occur in some definite order
LAMPORT'S REGISTER SEMANTICS

Registers hold a single value. Here, we consider single-writer registers only supporting write and read.

Semantics defined in terms of the real-time beginnings and ends of operations to the object.

- **safe:** a read not concurrent with any write obtains the previously written value
- **regular:** safe + a read that overlaps a write obtains either the old or new value
- **atomic:** safe + reads and writes behave as if they occur in some definite order

\[
\text{safe} \Rightarrow r_1 \rightarrow a
\]
LAMPORT'S REGISTER SEMANTICS

Registers hold a single value. Here, we consider single-writer registers only supporting write and read.

Semantics defined in terms of the real-time beginnings and ends of operations to the object.

- **safe**: a read not concurrent with any write obtains the previously written value
- **regular**: safe + a read that overlaps a write obtains either the old or new value
- **atomic**: safe + reads and writes behave as if they occur in some definite order

\[
\begin{align*}
\text{safe} & \Rightarrow r_1 \rightarrow a \\
\text{regular} & \Rightarrow r_1 \rightarrow a \land (r_2 \rightarrow a \lor r_2 \rightarrow b) \land (r_3 \rightarrow a \lor r_3 \rightarrow b)
\end{align*}
\]
LAMPORT'S REGISTER SEMANTICS

Registers hold a single value. Here, we consider single-writer registers only supporting write and read.

Semantics defined in terms of the real-time beginnings and ends of operations to the object.

- **safe**: a read not concurrent with any write obtains the previously written value

- **regular**: safe + a read that overlaps a write obtains either the old or new value

- **atomic**: safe + reads and writes behave as if they occur in some definite order

\[
\begin{align*}
\text{safe} & \Rightarrow r_1 \rightarrow a \\
\text{regular} & \Rightarrow r_1 \rightarrow a \land (r_2 \rightarrow a \lor r_2 \rightarrow b) \land (r_3 \rightarrow a \lor r_3 \rightarrow b) \\
\text{atomic} & \Rightarrow r_1 \rightarrow a \land (r_2 \rightarrow a \lor r_2 \rightarrow b) \land (r_3 \rightarrow a \lor r_3 \rightarrow b) \land (r_2 \rightarrow b \Rightarrow r_3 \rightarrow b)
\end{align*}
\]
**Sequential Consistency**

- Applies to arbitrary shared objects.
- Requires that a history of operations be equivalent to a legal sequential history, where a legal sequential history is one that respects the local ordering at each node.
- Called *serializability* when applied to transactions.
Is It Sequential?
Is It Sequential?

$p_1 \quad w(a)$

$p_2 \quad w(b)$

$r \rightarrow a \quad r \rightarrow b$

$p_3 \quad r \rightarrow c$

$p_4$
Is It Sequential?

\[ w(a) \quad w(b) \quad r \rightarrow a \quad r \rightarrow b \quad r \rightarrow c \]

NO.
Is It Sequential?

\[ p_1 \quad w(a) \quad w(c) \]
\[ p_2 \quad w(b) \]
\[ p_3 \quad r \rightarrow c \quad r \rightarrow a \]
\[ p_4 \quad r \rightarrow b \]
Is It Sequential?

\[
\begin{align*}
&\quad \quad w(a) \quad w(c) \\
&w(b) \\
r \rightarrow c \quad r \rightarrow a \\
&\quad \quad r \rightarrow b
\end{align*}
\]
Is It Sequential?

\[ w(a) \]
\[ w(b) \]
\[ r \rightarrow a \quad r \rightarrow b \]
\[ r \rightarrow a \quad r \rightarrow a \]
Is It Sequential?

\[ \text{w(a) \ \ r \rightarrow a \ \ r \rightarrow a \ \ r \rightarrow a \ \ w(b) \ \ r \rightarrow b} \]

\[ p_1 \quad p_2 \quad p_3 \quad p_4 \]

YES!
Is It Sequential?

\[ p_1 \rightarrow w(a) \]
\[ p_2 \rightarrow w(b) \]
\[ p_3 \rightarrow r \rightarrow a \quad r \rightarrow b \]
\[ p_4 \rightarrow r \rightarrow b \quad r \rightarrow a \]
Is It Sequential?

\[ w(a) \quad w(b) \]

\[ r \rightarrow a \quad r \rightarrow b \]

\[ r \rightarrow b \quad r \rightarrow a \]

NO.
**Linearizability**

\[
\text{Linearizability} = \text{sequential consistency} + \text{respects real-time ordering.}
\]

If \(e_1\) **ends** before \(e_2\) **begins**, then \(e_1\) **appears before** \(e_2\) **in the sequential history**.

Linearizable data structures behave as if there's a single, correct copy.
Atomic registers are linearizable.
Is It Linearizable?
Is It Linearizable?

\[ w(a) \]
\[ w(b) \]

\[ r \rightarrow a \quad r \rightarrow b \]

NO.
Is It Linearizable?
Is It Linearizable?

YES!
LINEARIZABILITY VS. SEQUENTIAL CONSISTENCY

• Sequential consistency allows operations to appear out of real-time order. How could that happen in reality?
**Linearizability vs. Sequential Consistency**

- Sequential consistency allows operations to appear out of real-time order. How could that happen in reality?

- The most common way systems are sequentially consistency but not linearizability is that they allow read-only operations to return **stale data**.
STALE READS

Primary Copy

Read-only Cache
STALE READS

Primary Copy

Read-only Cache
STALE READS

- Primary Copy
- Read-only Cache
Causal Consistency

- Writes that are not concurrent (i.e., writes related by the happens-before relation) must be seen in that order. Concurrent writes can be seen in different orders on different nodes.
Causal Consistency

- Writes that are not concurrent (i.e., writes related by the happens-before relation) must be seen in that order. Concurrent writes can be seen in different orders on different nodes.

- Linearizability implies causal consistency.
Is It Causal?

$p_1 \quad w(a) \quad p_2 \quad w(b) \quad p_3 \quad r \rightarrow a \quad r \rightarrow b \quad p_4 \quad r \rightarrow b \quad r \rightarrow a$
Is It Causal?

We need to know what causes what (i.e., what messages are sent)!
Is It Causal?

Diagram showing relationships between variables $p_1$, $p_2$, $p_3$, and $p_4$ with arrows indicating potential causality between $w(a)$, $w(b)$, and $r \rightarrow a$, $r \rightarrow b$. The diagram illustrates the complexity of identifying causality in a given set of variables.
Is It Causal?

w(a)

w(b)

p_1

p_2

p_3

p_4

r \rightarrow a

r \rightarrow b

r \rightarrow b

r \rightarrow a

YES!
But not sequential.
Is It Causal?
Is It Causal?

Not causal! (or sequential)
**Cool Theorem:** Causal consistency* is the strongest form of consistency that can be provided in an always-available convergent system.

Basically, if you want to process writes even in the presence of network partitions and failures, causal consistency is the best you can do.

[Mahajan et al. UTCS TR-11-22]

*real-time causal consistency*
WE CAN GET WEAKER!

- **FIFO Consistency**: writes done by the same process are seen in that order; writes to different processes can be seen in different orders. Equivalent to the PRAM model.

- **Eventual Consistency** ≈ if all writes to an object stop, eventually all processes read the same value. (Not even a safety property! "Eventual consistency is no consistency.")
Is It FIFO?

Diagram showing operations on processes $p_1, p_2, p_3, p_4$ with the following actions:
- $w(a)$ in $p_1$
- $w(b)$ in $p_2$
- $r \rightarrow b$ in $p_1$
- $r \rightarrow a$ in $p_2$
- $r \rightarrow b$ in $p_3$
- $r \rightarrow a$ in $p_4$
Is It FIFO?

YES! (but still not causal) (or sequential)
Is It FIFO?

\[
p_1 \quad w(a) \quad w(c) \\
p_2 \quad w(b) \\
p_3 \quad r \rightarrow a \quad r \rightarrow b \\
p_4 \quad r \rightarrow b \quad r \rightarrow c \quad r \rightarrow a
\]
Is It FIFO?

Not FIFO!

\[ \begin{align*}
  p_1 & : w(a), w(c) \\
  p_2 & : w(b) \\
  p_3 & : r \rightarrow a, r \rightarrow b \\
  p_4 & : r \rightarrow b, r \rightarrow c, r \rightarrow a
\end{align*} \]
Lamport's register semantics, sequential consistency, linearizability, and causal consistency, and FIFO consistency are all safety properties.
**Using Consistency Guarantees**

Initially, both `a` and `b` are 0.

What are the possible outputs of this program?

Thread 1

```
a = 1
print("b:" + b)
```

Thread 2

```
b = 1
print("a:" + a)
```
Using Consistency Guarantees

Thread 1

\[ a = 1 \]
\[ \text{print}("b:" + b) \]

Thread 2

\[ b = 1 \]
\[ \text{print}("a:" + a) \]

Initially, both \( a \) and \( b \) are 0.

What are the possible outputs of this program?

 Depends on memory consistency!
**Using Consistency Guarantees**

Thread 1

```python
a = 1
print("b:" + b)
```

Thread 2

```python
b = 1
print("a:" + a)
```

Suppose both prints output 0.
Using Consistency Guarantees

Suppose both prints output 0.

Then there's a cycle in the happens-before graph. Not sequential!
Aside: Java's Memory Model

- Java is **not** sequentially consistent!
- It guarantees sequential consistency only when the program is *data-race free*.
- A **data-race** occurs when two threads access the same memory location concurrently, one of the accesses is a write, and the accesses are not protected by locks (or monitors etc.).
A Common (incorrect) Idiom

class Foo {
    private Bar bar = null;

    public void baz() {
        if (bar == null) {
            synchronized(this) {
                if (bar == null) {
                    bar = new Bar();
                }
                bar.doAThing();
            }
        }
    }
}
A Common (Incorrect) Idiom, Corrected

class Foo {
    private volatile Bar bar = null;

    public void baz() {
        if (bar == null) {
            synchronized(this) {
                if (bar == null) {
                    bar = new Bar();
                }
            }
            bar.doAThing();
        }
    }
}
A COMMON (INCORRECT) IDIOM, CORRECTED

class Foo {
    private volatile Bar bar = null;

    public void baz() {
        if (bar == null) {
            synchronized(this) {
                if (bar == null) {
                    bar = new Bar();
                }
            }
        }
        bar.doAThing();
    }
}

Reminder: you don't need to worry about multi-threaded access for the labs!
(Except not grabbing locks in equals and hashCode)
How to Use Weak Consistency?

- Separate operations with stronger semantics, weak consistency (and high performance) by default
- Application-level protocols, either using separate communication, or extra synchronization variables in the data store (not always possible)
Main Takeaways

- The weaker the consistency model, the harder it is to program against (usually).
- The stronger the model, the harder it is to enforce (again, usually).