Replicated Data Consistency Explained Through Baseball

In this lecture, we'll see why storage systems replicate their data and how unfettered replication can lead to anomalies. We'll then discuss how strong and weak consistency can help tame anomalous behavior before concluding with a review of the 2013 CACM article *Replicated Data Consistency Explained Through Baseball*.

Replication

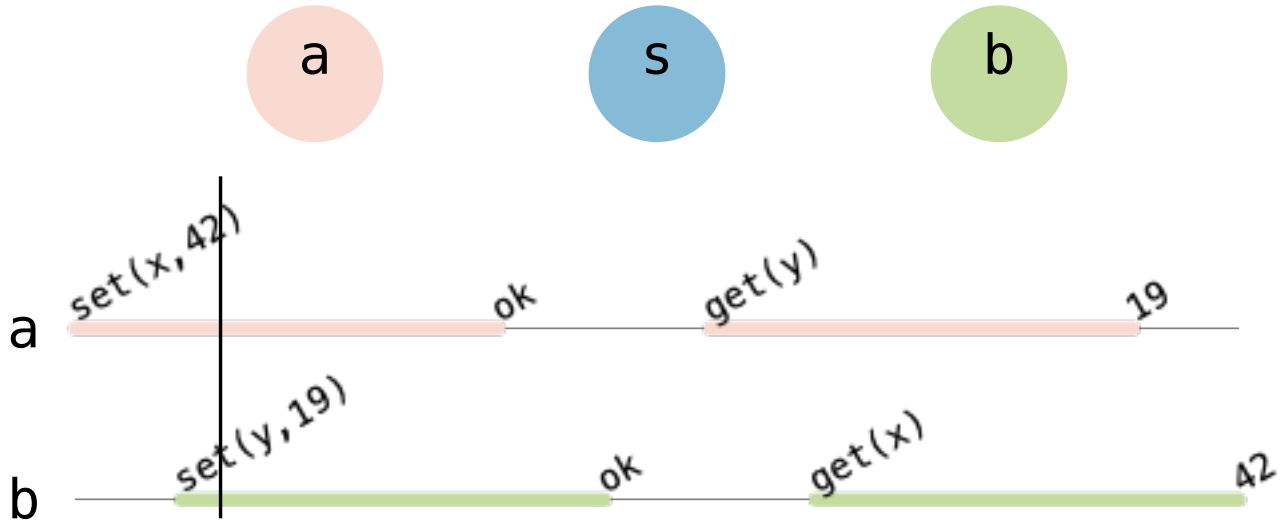
Often, distributed storage systems—like file systems, relational databases, or key-value stores—store a copy of the same data on multiple computers. This is known as **replication**. To motivate why storage systems replicate their data, we'll look at an example.

Consider a non-distributed key-value store running on a single computer. The key-value store is nothing more than a map (or dictionary) from string-valued keys to string-valued values. The key-value store supports a dirt simple interface. Clients can issue

- a `get(k)` request to retrieve the value associated with key `k` or
- a `set(k, v)` request to associate the value `v` with key `k`.

An example interaction between a client (`a`) and the key-value store (`s`) is animated below. The client first sends a `set(x,42)` request and then a `get(x)` request to the server. The top half of the animation shows how messages flow between the client and the server while the bottom half traces a timeline of every request and response.

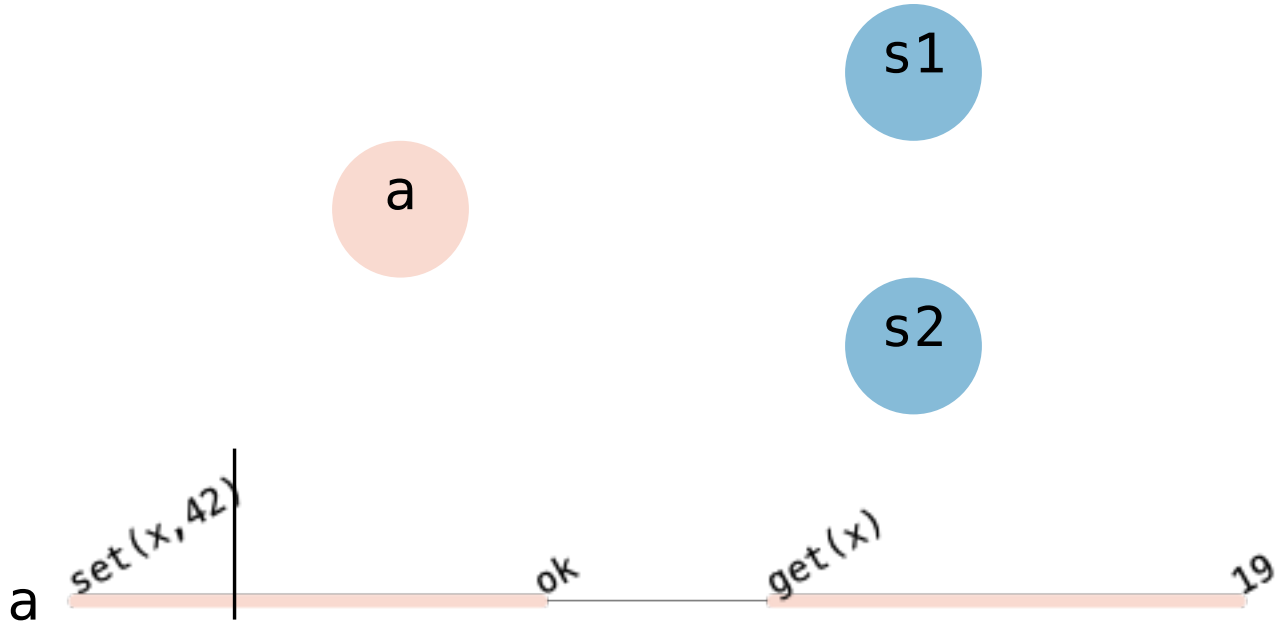
Even though there's only one key-value store, there can be multiple clients. Below, we animate an interaction where client a and client b concurrently set and then get a value from the key-value store `s`.



Unfortunately, computers don't live forever. Eventually, they crash. In our example, the entire key-value store is stored on `s`. This means that if `s` were to fail, we would irrevocably lose all of our data. That's no good! In reality, storage systems—like our key-value store—replicate data across multiple computers so that their data survives even when any single computer fails.

Strong and Weak Consistency

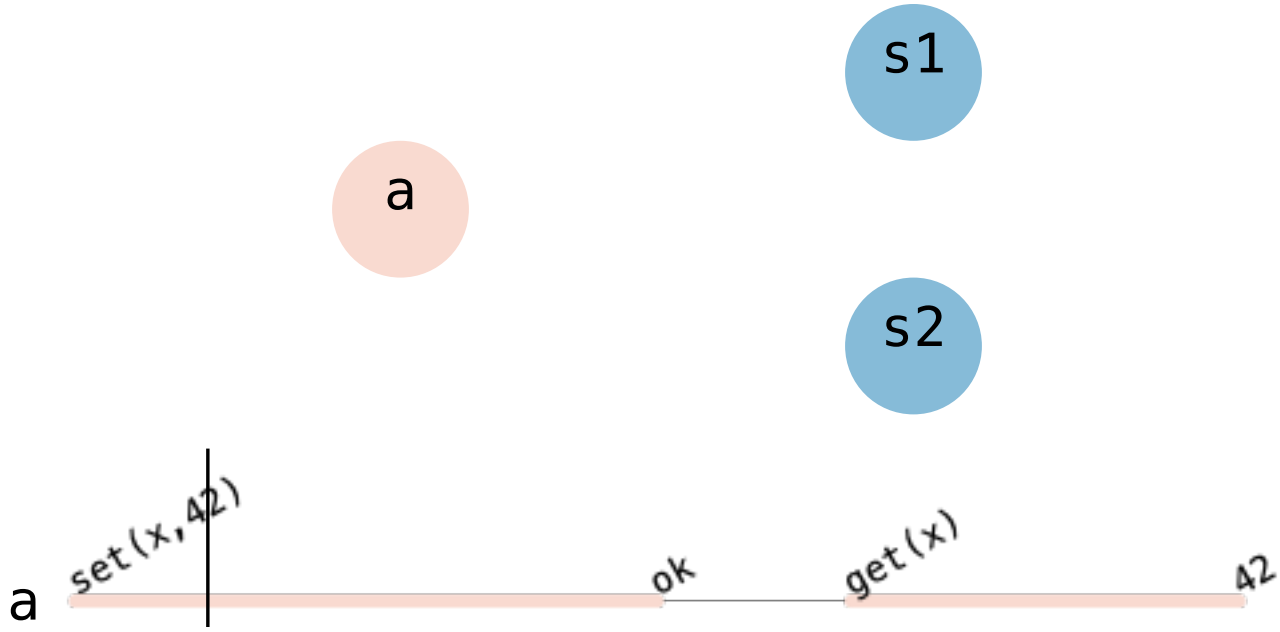
Replication allows a distributed storage system to tolerate computer failures. Unfortunately, a naively replicated storage system can behave very weirdly. For example, consider a key-value store replicated across two servers (`s1` and `s2`). If a client (a) issues a `set(x,42)` request to `s1` and then a `get(x)` request to `s2`, the `get` could return something that's not 42! This is animated below.



This behavior is bananas! If a client issues a `set(x, 42)` request followed by a `get(x)` request, we expect the key-value store to return 42. When a storage system exposes an unbridled number of anomalies like this and acts completely bananas, we colloquially say it is **inconsistent**.

Ideally, a storage system can hide the fact that it's replicated from clients and act indistinguishably from a storage system running on a single computer. The replication would increase the system's fault-tolerance but would not cause it to expose any anomalous behavior to clients. For example, a `get(x)` request following a `set(x,42)` request would always return 42. When a replicated storage system behaves indistinguishably from a storage system running on a single computer, we say it is **strongly consistent**.

Reconsider the execution above where client `a` sends a `set(x,42)` command to server `s1` and then a `get(x)` command to server `s2`. If `s1` propagates the effects of the `set(x,42)` command to `s2` before responding to `a`, then `s2` will correctly return 42 when it receives a `get(x)` request. By doing this, the system implements strongly consistency. This is animated below.



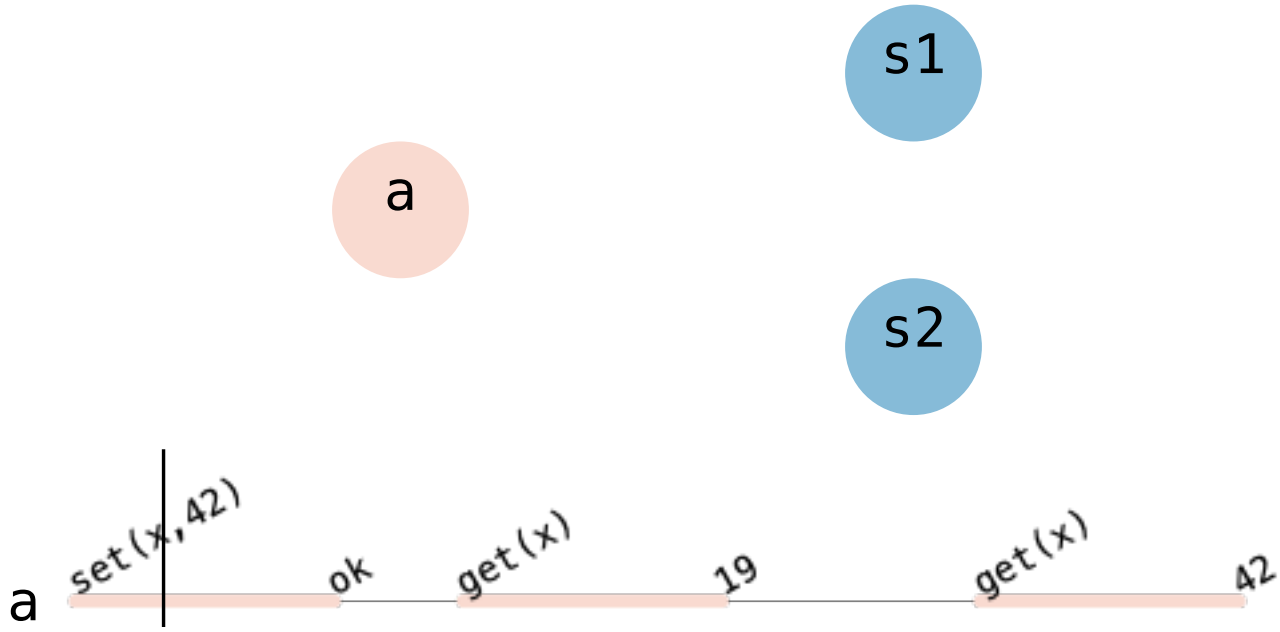
Note that the term "strongly consistent" is not well-defined. It means different things in different contexts. In some contexts, it might be used colloquially to express a general notion that a storage system doesn't act bananas. In other contexts, it might be used as a synonym for a very formally defined form of consistency like **linearizability** (which we'll define in the next lecture).

Unfortunately, implementing strong consistency—that is, implementing a system that is strongly consistent—is both challenging and costly. The algorithms used to implement strong consistency are often complex and nuanced. Worse yet, strong consistency is fundamentally at odds with low-latency and availability. We'll defer an in-depth discussion of this fact to the next lecture, but we already saw hints of it in the previous animation. When we made our key-value store strongly consistent, the `set(x, 42)` request took longer than it did when the key-value store was inconsistent.

As a consequence, systems often eschew strong consistency in favor of **weak consistency**. Weakly consistent storage systems do not behave indistinguishably from storage systems running on a single computer. While they do expose various anomalous behaviors, weakly consistent systems do not completely throw consistency out the window. Instead, they try to sit somewhere between acting completely bananas and acting with strong consistency. They provide some number of basic guarantees that are hopefully sufficient to satiate clients.

For example, consider one of the weakest forms of weak consistency: **eventual consistency**. An eventually consistent system guarantees that if all clients stop issuing requests for a while, then all the system's replicas will converge to the same state.

Let's again reconsider the example execution from above, except this time we'll look at an eventually consistent key-value store. Each server in the key-value store buffers write (i.e. `set`) requests and propagates them to the other servers every so often. As with the inconsistent key-value store, a `get(x)` requests following a `set(x,42)` request can return something other than 42. But, if a client waits long enough, *eventually* a `get(x)` request will return 42. This is animated below.



Replicated Data Consistency Explained Through Baseball

In addition to strong consistency, there are a buffet of flavors (or models) of weak consistency: eventual consistency, strong eventual consistency, causal consistency, causal+ consistency, RedBlue consistency, etc. Each consistency model exposes various degrees of inconsistency with various performance characteristics. Some simple ones can be implemented efficiently but are borderline bananas. Others are more complex but provide stronger consistency.

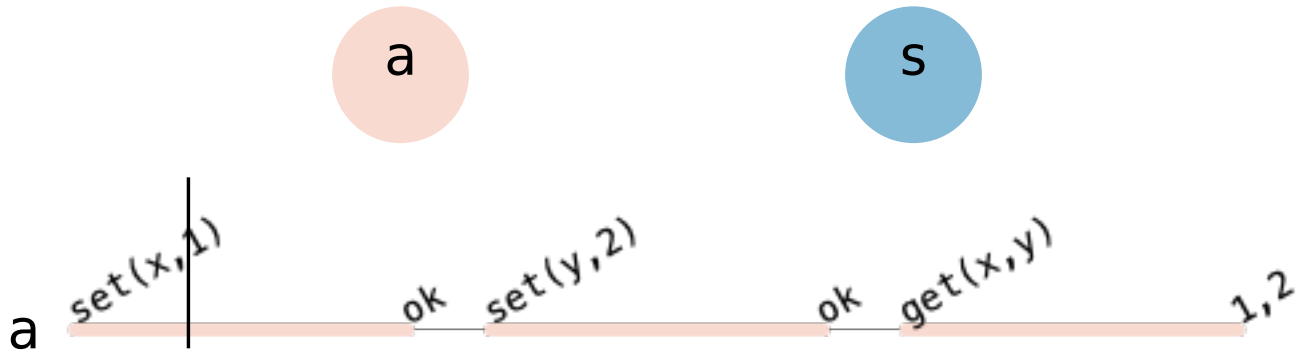
Is the baffling number of consistency models actually useful? Are the weaker

consistency models too weak? Are the stronger consistency models too strong?

Doug Terry's 2013 CACM article, *Replicated Data Consistency Explained Through Baseball*, answers these questions using the game of baseball. Terry defines six consistency models and then shows how different clients of a baseball application can benefit from each.

Consistency Models

In this section, we define six consistency models for a replicated key-value store. As with our examples above, there can be multiple clients issuing read (i.e. `get`) and write (i.e. `set`) requests to the replicated key-value store concurrently. We'll also augment `get` requests by allowing them to read multiple keys at once. For example, the request `get(x, y)` returns values for `x` and `y` simultaneously. This is animated below.



To make life easier for ourselves, we make a couple of simplifying assumptions. First, we assume that when a client issues a write request, the effects of the write are eventually propagated to all replicas. Second, we assume that all write requests are executed in the same order at all replicas (we'll see how to implement something like this in the third lecture). With these assumptions in place, the six consistency models will differ only in how read requests behave.

We briefly define each of the consistency models here. In the next section, we'll give an example illustrating the differences between the six.

1. **Strong Consistency.** With a strongly consistent key-value store, a `get(`x_1`,...,`x_n`)` request is guaranteed to return the most recently written

values of every key from x_1 to x_n.

2. **Eventual Consistency.** With an eventually consistent key-value store, a `get(`x_1`,...,`x_n`)` request is guaranteed to return values $v_1,...,v_n$ where v_i is any previously written value of key x_i. With our assumption that writes are eventually propagated to all replicas, if clients stop issuing write requests for a while, reads will (typically) return the most recently written values.
3. **Consistent Prefix.** Recall our assumption that writes are executed in the same order on all replicas. With a key-value store guaranteeing consistent prefixes, a `get` request is guaranteed to return values that are consistent with some prefix of this sequence of writes. Note that for `get` requests reading a single value, consistent prefix is equivalent to eventual consistency.
4. **Bounded Staleness.** With a key-value store guaranteeing bounded staleness, a `get(`x_1`,...,`x_n`)` request is guaranteed to return values $v_1,...,v_n$ where v_i is some value that key x_i took on during the last *t* minutes for some fixed *t*.
5. **Monotonic Reads.** With a key-value store guaranteeing monotonic reads, a client's initial read of value x is only guaranteed to return some previously written value of x (this is equivalent to eventual consistency). However, each subsequent read of x by the same client is guaranteed to return the same value of x or a more up-to-date value of x compared with the previous read of x.
6. **Read My Writes.** With a key-value store guaranteeing read my writes, if a client writes a value v to key x, then any subsequent reads of x by the same client will return v or a more recently written value of x.

Baseball as a Sample Application

The name of the game is baseball. A baseball game consists of nine rounds, called innings. During the first half of each inning (called the top of the inning), the visiting team bats until they make three outs. Then, during the second half of each inning (called the bottom of the inning), the home team bats until they make three outs. Whichever team scores more runs by the end of the game wins.

In the rest of this lecture, we'll model a baseball game using the pseudocode below. The state of a baseball game is stored in a replicated key-value store. The visiting score is stored under the key `"visitors"`, and the home score is stored under the

key "home".

```
set("visitors", 0);
set("home", 0);
for inning = 1 .. 9
  outs = 0;
  while outs < 3
    visiting player bats;
    for each run scored
      score = get("visitors");
      set("visitors", score + 1);
  outs = 0;
  while outs < 3
    home player bats;
    for each run scored
      score = get("home");
      set("home", score + 1);
end game;
```

For simplicity, we'll assume that a single client (the official scorekeeper) updates the score; no other clients write to the "visitors" or "home" key in the key-value store. To illustrate the differences between the six consistency models defined above, let's jump into an example baseball game during the middle of the seventh inning. At this point in the game, the following sequence of writes has been produced by the official scorekeeper.

- set("visitors", 0)
- set("home", 0)
- set("home", 1)
- set("visitors", 1)
- set("home", 2)
- set("home", 3)
- set("visitors", 2)
- set("home", 4)
- set("home", 5)

This sequence of writes corresponds to the following inning-by-inning line score. We see that the visiting team scored once in the third and fifth inning. The home team scored once in the first, third, and fourth inning and scored twice in the sixth inning. Thus, the visiting team is behind two runs to five. We abbreviate this score as 2-5, visiting team first.

	1	2	3	4	5	6	7	8	9	RUNS
Visitors	0	0	1	0	1	0	0			2
Home	1	0	1	1	0	2				5

Now, for each of the six consistency models defined above, we list all the possible scores that could be returned from issuing a `get("visitors", "home")` request.

Strong Consistency	2-5
Eventual Consistency	0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5
Consistent Prefix	0-0, 0-1, 1-1, 1-2, 1-3, 2-3, 2-4, 2-5
Bounded Staleness *(scores that are at most one inning out-of-date)*	2-3, 2-4, 2-5
Monotonic Reads *(after reading 1-3)*	1-3, 1-4, 1-5, 2-3, 2-4, 2-5
Read My Writes *(for the writer)*	2-5
Read My Writes *(for anyone other than the writer)*	0-0, 0-1, 0-2, 0-3, 0-4, 0-5, 1-0, 1-1, 1-2, 1-3, 1-4, 1-5, 2-0, 2-1, 2-2, 2-3, 2-4, 2-5

A few things to note:

- With a *strongly consistent* key-value store, the only possible result is 2-5: the current score of the game.
- With an *eventually consistent* key-value store, the visiting score can be read as 0, 1, or 2, and the home score can be read as 0, 1, 2, 3, 4, or 5. Note that some of the scores read—like 2-0 or 0-5—were never actually the score of the game at any point in time.
- With a key-value store guaranteeing *consistent prefixes*, the score that's read may not be the most up-to-date score, but it is guaranteed to have been the score of the game at some point in the past.

- With a key-value store guaranteeing *bounded staleness* within the last inning, the visiting score must be read as 2 because the visiting team did not score in the sixth inning. The home score can be read as 3, 4, or 5: the three home scores that occurred in the sixth inning.
- After reading 1-3 from a key-value store guaranteeing *monotonic reads*, the visiting score can be read as 1 or 2, and the home score can be read as 3, 4, or 5.
- Since there is only a single client who updates the score, *read my writes* is equivalent to strong consistency for the writer and equivalent to eventual consistency for everyone else.

Read Requirements for Participants

In this section, we'll look at six different people—a scorekeeper, an umpire, a radio reporter, a sportswriter, a statistician, and a stat watcher—and discuss the weakest consistency model each person requires. In doing so, we'll see why all six consistency models are useful.

Official Scorekeeper

As discussed above, the official scorekeeper is responsible for updating the score whenever the visiting team or home team scores a run. To ensure that the score is always correct, the scorekeeper must read the most up-to-date score just before updating it. This could be accomplished with strong consistency, but since the scorekeeper is the only one who updates the score, read my writes (which in this case is equivalent to strong consistency) suffices.

Umpire

The umpire officiates the game. After the top of the ninth inning, the home team finishes the game at bat. If the home team has more runs than the visiting team at this point, then they are guaranteed to win, no matter how poorly they perform in the bottom of the ninth inning. In this event, it is the responsibility of the umpire to end the game early. This responsibility is codified below.

```
 if first half of 9th inning complete then
```

```
  vScore = get("visitors");
  hScore = get("home");
  if vScore < hScore
    end game;
```

In order to decide whether or not to end the game, the umpire needs the current score of the game. Thus, the umpire requires **strong consistency**.

Radio Reporter

Every thirty minutes, the radio reporter announces the score of the game over the radio.

```
do {
  vScore = get("visitors");
  hScore = get("home");
  report vScore and hScore;
  sleep(30 minutes);
}
```

The reported score does not have to be up-to-date; listeners are accustomed to hearing out-of-date scores reported on the radio. However, as not to mislead listeners, the radio reporter should not report a score that never actually occurred. Thus, the radio reporter requires **consistent prefix**.

Moreover, once the radio reporter reports a score, the reporter would look foolish to announce a previous score. For example, it would be a faux pas to announce a score of 2-3 (middle of the fifth inning) and then announce a score of 0-1 (middle of the second inning) 30 minutes later. Thus, the reporter also requires either **monotonic reads** or **30-minute bounded staleness**.

Sportswriter

The sportswriter is tasked with writing an article about the game after it finishes. A typical sportswriter executes the following pseudocode.

```
while not end of game {
  drink beer;
  smoke cigar;
}
go out to dinner;
vScore = get("visitors");
hScore = get("home");
write article;
```

The sportswriter's article must be written with the correct final score. While strong consistency would suffice, the sportswriter can take advantage of a lengthy dinner. If the sportswriter eats for an hour, then **1-hour bounded staleness** is equivalent to strong consistency and meets the sportswriter's needs.

Statistician

The statistician, say for the home team, is responsible for recording season-long statistics such as the total number of runs scored. The statistician stores this statistic in the key-value store under the `"season-runs"` key. A couple hours after each game, the statistician adds the number of runs from the game to the season-long aggregate.

```
Wait for end of game;
score = get("home");
state = get("season-runs");
set("season-runs", stat + score);
```

As with the sportswriter, the statistician can use **bounded staleness** to read the home score. Since the statistician is the only person who updates the season-long number of runs, the statistician can use **read my writes** to read the latest value of `"season-runs"`, similar to how to official scorekeeper used read my writes to read the score during the game.

Stat Watcher

A stat watcher checks on their team's statistics once every day and discusses them

with friends. Because the stat watcher checks the stats infrequently and because reading up-to-date stats is not essential, **eventual consistency** suffices for a stat watcher.

```
do {
  stat = get("season-runs");
  discuss stats with friends;
  sleep(1 day);
}
```