Hercules: A Multi-View Cache for Real-Time Interactive Apps

Technical Report UW-CSE-2019-12-01

Niel Lebeck
University of Washington
Jonathan Goldstein
Microsoft Research
Irene Zhang
Microsoft Research

Abstract
Existing distributed storage systems do not meet the needs of real-time interactive apps. These apps feature small groups of users making concurrent modifications, tight responsiveness bounds, and wide-area distribution. As a result, they need a storage system that provides simultaneous access to multiple versions of shared state, where the versions trade off consistency and staleness, and there are versions representing the extreme ends of the consistency/staleness spectrum. We present Hercules, a distributed storage system that meets these needs using client-side caching. Hercules uses batching to preserve performance in high-throughput scenarios and includes recovery protocols to maintain liveness in the face of client-side failures. We experimentally evaluate Hercules’s performance and failure tolerance, and we present a set of example apps that demonstrate the versatility of Hercules’s programming model.

1 Introduction

Real-time interactive apps are an important and increasingly prominent class of distributed applications. These apps let users manipulate a set of shared data and interact with each other in real time. They run on modern client devices such as smartphones, tablets, virtual reality (VR) headsets [23], and interactive whiteboards [19], which have natural user interfaces with touch or gesture input and high-resolution visual output. Some examples are collaborative drawing and design apps, multiplayer games, messaging apps, and collaborative office apps like Google Docs.

Three properties distinguish real-time interactive apps from traditional distributed apps:

1. They feature small groups of users concurrently modifying shared state. Users continuously respond to each others’ actions and interleave operations.

2. They have tight responsiveness bounds for updating output in response to input. Human-computer interaction research has established bounds of around 100ms [20, 22], and new devices such as VR headsets have even tighter bounds of 7-20ms [1].

3. Clients are distributed over the wide area. Users participating in the same experience may be located in different regions or parts of the world, and even co-located users have client devices separated from cloud infrastructure by high-latency last-mile network connections.

Client-side caching is an essential part of distributed storage for real-time interactive apps. A cache allows these apps to quickly access shared state and meet their responsiveness bounds despite wide-area latencies. Unfortunately, existing storage systems provide limited or no client-side caching, leaving apps to construct the necessary caches themselves. This task is difficult—apps must maintain separate copies of shared state, and developers must reason about how to keep those copies synchronized with the underlying storage system. Depending on the interface provided by the storage system, maintaining the coherence of an independent client-side cache may even be impossible, or require developers to essentially build a secondary storage system on top of the first one.

This paper presents Hercules, a distributed storage system that uses client-side caching to meet the needs of real-time interactive apps. Hercules provides simultaneous access to multiple versions of shared state at different consistency levels, called views, so that apps can expose uncertainty to users. Its views trade off staleness for consistency, since apps need low-latency access to all views in order to meet their responsiveness requirements. Finally, Hercules provides novel views of state at the extreme ends of the staleness/consistency continuum, which are useful to apps but unsupported by existing systems.

Our contributions in this paper are as follows:

• We identify the unique needs of real-time interactive apps, and we identify multi-versioned client-side caching as a means of satisfying those needs.

• We describe the design and implementation of Hercules, a storage system that uses client-side caching to provide
low-latency views of shared state, including novel views not supported by existing systems.

- We present a set of example applications that showcase the use of Hercules’s different state views, and we evaluate Hercules’s performance optimizations and recovery protocols.

We implemented our Hercules prototype and example applications in C# on top of the AMBROSIA reliable RPC framework [9]. Our experiments show that apps built on Hercules have 1.7x–9x lower operation delays than apps built on a commercial cloud storage system, that Hercules’s batching maintains stable performance as a deployment scales to include more clients, and that its recovery protocol allows it to maintain liveness in the face of client failures.

2 Motivation

This section describes what real-time interactive apps need from a distributed storage system, and it shows how existing storage systems do not meet those needs.

2.1 The Needs of Real-Time Interactive Apps

The tight responsiveness bounds and wide-area distribution of real-time interactive apps mean that they need a low-latency client-side cache of shared state. This cache has three requirements, which we describe below.

Simultaneous access to multiple views. Multiple views of shared state let real-time interactive apps easily expose uncertainty when rendering client output. This uncertainty comes from the fact that clients must respond to user input immediately, but that input may conflict with other users’ concurrent actions. Apps typically handle this tension by speculatively applying a user’s own actions right away and then correcting the output once the storage system orders the actions and resolves conflicts. Some apps indicate which parts of the output are uncertain, but implementing that functionality currently

Figure 1: A sequence of screenshots from our interior design app, showing how it exposes uncertainty to the user. In (b), the user has just moved one of the chairs, and that change has not yet been synchronized with the cloud or other users, so the client highlights the chair in red.

Figure 2: An illustration of the relationship between staleness and consistency. A highly available state view cannot be both up-to-date and strongly consistent, so each state view represents a tradeoff between the two properties.

requires substantial developer effort, so many apps simply treat speculative output as authoritative and risk “lying” to the user for brief periods.

We implemented an “interior design” app as an example of how an app can indicate uncertainty in its output. The app lets users collaboratively design a room’s layout by moving around virtual items of furniture. The interior design app exposes uncertainty by highlighting furniture items whose position is uncertain (i.e., items which the user has just moved), as shown in Figure 1.

The easiest way to identify uncertainty is to compare different views of shared state representing different consistency levels. However, most existing systems provide interfaces that expose only a single state view at a time, even if they support multiple consistency levels. In order to simultaneously access multiple state views using these systems, apps must invoke read operations at multiple consistency levels and then manually cache the results.

Views that trade off staleness and consistency. The tight responsiveness bounds of real-time interactive apps mean that they must render each output frame very quickly, within tens of milliseconds. With those short frame times, apps cannot afford to make long-running queries for shared state on the output-rendering critical path.

Existing systems may provide immediate access to weakly consistent state views, but for strongly consistent views, they generally require apps to either perform a blocking query or register an asynchronous callback. These systems therefore offer a tradeoff between consistency and performance (access latency). Real-time interactive apps cannot afford to sacrifice access latency when rendering output, but they can tolerate staleness when reading strongly consistent versions of state. They need a storage system that instead exposes the tradeoff between staleness and consistency. Figure 2 illustrates this tradeoff.
New, unique views. Real-time interactive apps benefit from views of shared state that are not supported by existing storage systems. Most existing systems provide a strongly consistent, Authoritative state view representing the agreed-upon ground-truth version of state at a point in time. Many systems also provide a weakly consistent state view that includes operations that have been persisted locally on the client but have not yet propagated to cloud storage. We refer to this view as the Durable view. These two views alone are insufficient for real-time interactive apps.

Due to their responsiveness needs, real-time interactive apps need a low-latency state view that includes operations before they are locally persisted. Modern immersive client devices such as VR headsets have extremely tight response bounds of 7-20ms for updating output in response to input [1], and those bounds will decrease in the future. Apps running on these devices cannot wait for changes to shared state to be locally persisted before those changes are reflected in their output. They need a view including operations that are only stored in memory, which we call the Submitted view.

On the other end of the consistency spectrum, because real-time interactive apps have concurrent user activity, they also benefit from a view including only operations that other users have received. We refer to this view as the Visible view. Many popular apps [7, 13] provide such visibility information, but because storage systems do not track operation visibility, apps are left to implement this behavior themselves.

2.2 Current Systems

Widely available distributed storage systems do not meet the requirements outlined above. Today’s real-time interactive apps must implement the missing functionality themselves, a difficult task made harder by the opaque interfaces of storage systems. This section examines a popular distributed storage system and describes the problems it presents for these apps.

Firebase Cloud Firestore [11] is a representative example of a distributed storage system. It provides a data store organized into collections of JSON documents. Cloud Firestore allows client apps to submit writes or transactions modifying shared data, and it supports event listeners that invoke a callback when a document or query set changes. An event listener passes the most recent snapshot of shared data into its callback. Cloud Firestore optimistically applies a client’s own operations locally before sending them to the backend, and it invokes event listener callbacks in response to those optimistic invocations. Event listeners provide metadata that can be used to distinguish between notifications for local and backend events.

Apps must explicitly store operations. An app that maintains multiple views of shared state must model and store app-level operations in memory, and it must implement protocols to re-compute its views using those operations when Cloud Firestore notifies it of changes to shared state. If the app wants to use views that lag behind the most recent version of data in Cloud Firestore, such as the Visible view described above, it must also store those operations in Cloud Firestore rather than simply storing the shared state itself.

Notifications are too coarse-grained. Cloud Firestore’s event listeners can register for changes to either a single document or the results of a query. With only these two options, apps cannot subscribe just to notifications for new operations. They must instead listen to the entire set of operations, figure out which operations are new, and act on those operations. This requirement both adds an extra programming burden (developers must implement de-duplication logic) and reduces performance (the entire set of operations is delivered every time a new operation is added).

Apps must perform manual synchronization. Cloud Firestore’s event listener callbacks execute on background threads, while real-time interactive apps typically process input and render output on a dedicated, non-blocking user interface thread. Apps must therefore add synchronization logic to ensure that event listener callbacks do not access shared state or operation metadata at the same time as input-processing or output-rendering methods, and they must structure the event listener callbacks to execute without blocking, so that the added synchronization does not cause the user interface to block.

3 Hercules Overview

Hercules is a distributed storage system that exposes different views of shared state to clients, where each view reflects a particular tradeoff of consistency and staleness.

3.1 System Model

Hercules targets distributed applications with a client/server architecture, in which users interact with client apps that submit operations to a central server, and the server maintains the authoritative copy of shared state and fans out operations to other clients. Although our current Hercules prototype supports a single (possibly replicated) server, Hercules could be extended to support more sophisticated backend architectures by modifying or adding new state views.

Hercules targets real-time interactive apps, where small numbers of users (e.g., tens of users) continuously modify shared state and respond to each others’ operations. Hercules does not target distributed apps with different workloads, such as social media services that have millions of users and take minutes to fully propagate operations. Many of Hercules’s techniques could be applied to these apps in a straightforward manner, but some would require adjustment (for instance, it
might be impractical to track which of a user’s operations are visible to all other users).

### 3.2 Programming Interface

Hercules models an application’s shared state as a state machine. The state machine starts at some initial state and moves between states according to the operations in a totally-ordered operation log. Hercules supports two kinds of operations: read-write operations and read-only operations. Read-write operations can perform arbitrary deterministic computation, but they cannot return values or externalize state, because they are applied speculatively to some views. Read-only operations read a particular view of the state machine. Read-write operations are stored in the operation log, while read-only operations are unlogged.

When using Hercules, the application developer writes a state machine class whose variables define the shared state and whose methods define the read-write operations. Hercules generates a client library with two kinds of methods: operation stubs matching the read-write operations, and a `GetState()` method that returns the requested view of shared state. Applications perform read-write operations by calling the operation stubs, and they perform read-only operations by calling `GetState()` and then directly reading from the returned view.

### 3.3 Hercules State Views

The Hercules client library provides different views of the shared state, where each view is a read-only copy of the state machine resulting from a different operation log. The different views trade off consistency and staleness. In this context, staleness describes how long a client must wait for its own operations to be included in a view; a weakly consistent view allows operations to be inserted into the middle of its log, while a strongly consistent view only permits operations to be appended to the end of its log. Our Hercules design exposes four views, in order of increasing staleness:

- The **Submitted** view is a weakly consistent view including all operations that have been submitted by this client.
- The **Durable** view is a weakly consistent view that guarantees that its operations will eventually become visible to all clients.
- The **Authoritative** view is a strongly consistent view that guarantees that its operations will eventually become visible to all clients.
- The **Visible** view is a strongly consistent view that guarantees that all of the client’s own operations contained in it are visible to all clients.

Each view is defined in terms of a *global operation log*, which is agreed upon by all clients and which specifies the “master copy” of shared state. Operations submitted by clients eventually enter the global log; how exactly clients agree upon the global log depends on the precise system architecture. A client’s **Authoritative** log is a prefix of the global log, and its **Visible** log is a prefix of the **Authoritative** log. The **Submitted** log consists of the **Authoritative** log with all of the client’s submitted operations appended to the end, and the **Durable** log is the **Authoritative** log with a prefix of the client’s submitted operations appended. Figure 3 illustrates these definitions with an example.

These definitions result in a clean set of prefix relationships between the views, where each view’s operation log is the prefix of another view’s. The client’s **Visible** log is a prefix of its **Authoritative** log, which is a prefix of its **Durable** log, which is a prefix of its **Submitted** log.

Note that our definition of the **Visible** view is concerned with the visibility of a client’s own operations. Operations from other clients that are not yet visible to everyone may appear in it. We found that this definition results in a view that is intuitive to use, since users are often primarily interested in whether their own operations are visible to others. However, there may be situations where (for instance) Alice sees Bob’s operation and wants to know whether Carol has also seen that operation. In that case, a view that shows operations from any client only if they are visible to all clients would be useful. Such a view would fit cleanly into Hercules’s design but would require more messages per operation.

### 4 Hercules Design

Hercules provides clients with views of shared state that represent different tradeoffs of consistency and staleness. In the context of a concrete storage system, another interpretation of the views is that each view contains operations that have
Figure 4: The architecture of a Hercules application. Shaded areas represent components provided by Hercules.

passed beyond a certain “radius” in the system, from the point of view of the client. To provide those different views, Hercules turns each operation into multiple RPCs that provide finer-grained information about the operation’s progress through the system. The Hercules client library uses those RPCs to derive its views.

4.1 AMBROSIA Background

Hercules is built on top of AMBROSIA [9], a distributed runtime that provides reliable execution and in-order, exactly-once RPC delivery across nodes. We used AMBROSIA to build Hercules because its reliability guarantees simplify Hercules’s design.

AMBROSIA nodes are called Immortals; each Immortal consists of a package of persistent state and a set of deterministic RPC handlers that operate on that state. An Immortal can call RPCs on other Immortals or perform self-calls of its own RPCs. A special type of RPC is an Impulse Handler, which an Immortal calls on itself to accept non-deterministic external input.

4.2 System Architecture

A Hercules deployment consists of a server program and one or more client programs running on end-user devices. Hercules generates the server program and a client library; app developers write code for a client program that accepts user input and renders output using the client library. The server stores the global operation log. To save space, it maintains a checkpoint copy of shared state that represents a prefix of the global log and only explicitly stores the suffix of the log. The server stores all of its state in durable storage, whereas the client splits its state across durable storage and volatile memory. Figure 4 illustrates this architecture.

The Hercules server program runs an AMBROSIA Immortal, and each instance of the Hercules client library runs an Immortal on a background thread. All communication between the server and client library happens via RPCs between their Immortals, and each component stores its persistent state inside of its Immortal.

In the context of Hercules’s system architecture, each view has the following interpretation:

- The Submitted view includes all operations that have been submitted by this client, even those that are only stored in memory.
- The Durable view includes submitted operations from this client that have been durably written to the client’s local storage.
- The Authoritative view includes operations that have been accepted and ordered by the server.
- The Visible view includes all operations up to (but not including) the first operation from this client that has not been delivered to all other clients.

Operations in the Submitted view may be lost if the client app crashes, and operations in the Durable view may be lost if the client device suffers a persistent storage failure.

4.3 Operation RPC Protocol

When a client app performs an operation, Hercules transforms that operation into multiple RPCs that propagate it through the system and inform the client library of its progress. The operation propagation protocol, illustrated in Figure 5, has the following steps:

1. The client app calls an operation stub exposed by the Hercules client library. Once the stub call returns, the operation is Submitted.
2. The client library calls the Make-Durable ImpulseHandler RPC on itself to locally persist the operation. The operation is now Durable.
3. The client library sends a Deliver-Operation RPC containing the operation to the Hercules server.
4. The server adds the operation to its global operation log and sends a Notify-Auth RPC to the client library. The operation is now Authoritative.
5. The server sends Remote-Operation RPCs containing the operation to the other clients.
6. Each other client responds with a Remote-Ack RPC.
7. Once the server has received acks from all other clients, it sends a Notify-Visible RPC to the client library. The operation is now Visible.

Each RPC call is asynchronous and non-blocking. At any time, the client app can call the GetState() method to get the client library’s current state views.
4.4 Client View Computation

The Hercules client library uses the RPCs it receives to assemble its state views. This procedure requires some care. This section describes the basic algorithm for computing state views; Section 4.5 describes a version optimized for high-throughput scenarios.

The client library maintains a persistent copy of state corresponding to its Visible view and transient copies for the other views, and it saves computation by directly applying operations to those transient copies when possible rather than re-deriving them from scratch. Furthermore, because the full operation log of a given state view is a prefix of the log of its less-stale counterpart (e.g., the Authoritative log is a prefix of the Durable log), the client library saves space by storing operation lists that contain only the operations that are part of one log and not its less-stale counterpart (e.g., the Durable list holds the operations that have been persisted but have not yet been received by the server). It uses these operation lists to re-derive state views when necessary. Figure 6 shows the relationship between the operation logs and operation lists.

The client library updates its state views whenever it receives an RPC from the server (summarized in Figure 7). When the RPC notifies it about the progress of one of its own operations, then the operation log for one of its state views grows to be a larger prefix of its less-stale counterpart, but the operation logs for the other state views are all unchanged. In this situation, the client library can directly apply the operation to the affected view without changing any of the others.

However, if the RPC contains a remote operation from another client, then the remote operation will be inserted into the middle of several views’ operation logs. Whenever an operation is inserted into the middle of an operation log, the client library must re-compute the corresponding view by starting with a copy of its more-stale counterpart and re-applying the subsequent operations in the log. We call this re-computation a rebase. We can perform rebasing more efficiently by rebasing multiple views at once (e.g., rebasing the Durable view off of the Authoritative view, and then rebasing the Submitted view off of the Durable view).

Specifically, the client library takes the following actions when it receives RPCs corresponding to some operation $O$:

- **Make-Durable**: The Durable view changes. The client library removes $O$ from its Submitted list, appends $O$ to its Durable list, and directly applies $O$ to the Durable view.
- **Notify-Auth**: The Authoritative view changes. The client library removes $O$ from its Durable list, appends $O$ to its Authoritative list, and directly applies $O$ to the Authoritative view.
- **Notify-Visible**: The Visible view changes. The client library removes $O$ and any preceding operations from its Authoritative list and directly applies those operations to its Visible view in order.
- **Remote-Operation**:
  - If any of this client’s Authoritative operations are not yet Visible: the Authoritative, Durable, and Submitted views change. The client library appends $O$ to its Authoritative list, directly applies $O$ to the Authoritative view, and rebases the Durable and Submitted views.
  - If all of this client’s Authoritative operations are Visible: all four views change. The client library directly applies $O$ to the Visible view and rebases the other three views.

4.5 Client Rebase Batching

In the client view computation algorithm introduced in Section 4.4, a client performs a rebase whenever it receives a Remote-Operation RPC. Each rebase involves re-executing multiple operations, so rebasing can become prohibitively
Hercules makes no guarantees about the Submitted view; its operations may be lost at any time due to transient failures. It is this property that enables Hercules to quickly update the Submitted view in response to user input. The client library stores the Submitted operation list in volatile memory and adds submitted operations to it inside of a regular method rather than an AMBROSIA RPC. If the client program dies and restarts, it simply clears the Submitted operation list during recovery.

In contrast, Hercules guarantees that operations in the Durable view will eventually be sent to the server, unless the client suffers a persistent storage failure. It maintains this promise by including the Durable operation list in the client Immortal’s persistent state and only appending to it inside of an AMBROSIA RPC. If the client app suffers a transient failure, its recovery procedure restores its Durable operation list, and AMBROSIA guarantees that its Deliver-Operation RPC is eventually sent to the server. On the other hand, if the client device’s persistent storage fails, then the operations in its Durable operation list are lost. In this situation, the client app would start up as a new client instance, and any operations that it successfully delivered to the server before failing would be reflected in its Authoritative and Visible views.

The server stores its global operation log and visibility-tracking metadata in its Immortal’s persistent state, and it performs all modifications to that data inside of AMBROSIA RPCs, making it robust to transient failures. The server can use replication or another method to avoid persistent storage failures; such a failure would require the Hercules instance to start over from scratch.

**Maintaining the visibility set.** In addition to maintaining its guarantees in the face of failures, Hercules must also ensure that it remains useful when clients crash or disconnect. This concern primarily affects the Visible view, which progresses only when all clients have received a client’s operations. In practice, defining the Visible view in terms of all clients is impractical when some of those clients may experience transient connection issues or failures. As a result, Hercules defines a visibility set containing the clients that are actively receiving operations. It guarantees that a client’s operations in the Visible view have been delivered to all clients in the visibility set, and it notifies clients whenever the visibility set changes.

The Hercules server detects crashed or disconnected clients by periodically checking for clients with acks that have been outstanding for more than some timeout interval (e.g., every two seconds). Once the server detects a timeout, it updates its visibility set and sends out a Visibility-Set-Change RPC.
We implemented Hercules in C# on top of AMBROSIA [9].

with a checkpoint copy of shared state representing the prefix
(1982 LOC for Hercules itself and 1287 LOC for the exam-
ple apps), not counting any files that are generated by either
Hercules or AMBROSIA.

Our implementation consists of 3269 lines of C# code
built, in order to demonstrate the benefit of Hercules’s state
views when possible. This difference makes the code simpler
at the expense of performing a bit of unnecessary work.

6 Application Case Studies
In this section, we describe some of the example apps that we
built, in order to demonstrate the benefit of Hercules’s state views abstraction as well as to highlight the variation in the
ways different apps use that abstraction.

6.1 Interior Design App

The interior design app allows users to design the layout of a
two-dimensional virtual room. Users click and drag items of
furniture, such as tables and chairs, to move them around in
the room.

The interior design app client uses the Submitted and Visi-
ble views. It indicates to the user if the movement of an item
is not yet visible to other users by highlighting the item on-
screen. Once the item’s updated position is reflected in the
Visible view, the client changes the item’s appearance back to
normal. Figure 1 illustrates the client’s behavior.

Figure 8 shows the interior design app client’s code for
rendering output. Items in the interior design app can exist
at any point in the two-dimensional room, and it is an item’s
position itself that changes as a result of user actions, so the
interior design app uses explicit item IDs to identify differ-
ences between the state views. The client iterates over all item
IDs and checks if the position of the item with that ID differs
between the two state views. If so, it highlights the item when
drawing it.

Our implementation of the interior design app uses the Vis-
ible view to show users the ground-truth version of shared
state, but the right choice of view may differ for different
deployment scenarios. We assume that users tend to concur-
rently move around the same items, in which case they benefit
from knowing which of their operations are visible to others.
If users instead tend to move around separate groups of items,
the Authoritative view may be more appropriate, shortening

5 Implementation

We implemented Hercules in C# on top of AMBROSIA [9].
Our Hercules prototype includes the core client library and
server logic, profiling code to measure their performance, and
a code generation program that generates the client library and
server interfaces for a particular app. We also implemented
a set of example apps used to demonstrate and evaluate Her-
cules. Our implementation consists of 3269 lines of C# code
(1982 LOC for Hercules itself and 1287 LOC for the exam-
ple apps), not counting any files that are generated by either
Hercules or AMBROSIA.

Our client library implementation differs very slightly from
the description in Section 4, in that it performs a full rebase
of all three state views upon receiving a Remote-Operation
RPC, rather than rebasing only the Durable and Submitted
views.

to every other client. It then checks every operation that is
not yet visible; if it has beenacked by every client except the
just-removed client, the server marks it as visible and sends a
Notify-Visible RPC to its sender.

The server must also alert the client that it has been removed
from the visibility set (if the client is alive but suffering from
connection issues) and provide it with a way of re-joining after
it recovers. AMBROSIA’s reliability guarantees mean that a
crashed client is indistinguishable from a client with very long
RPC delays, so Hercules handles crashed and disconnected
clients with the same protocol.

When the server removes a client from the visibility set, it
 sends the client a Deregister RPC. AMBROSIA guarantees
that the client will eventually receive the RPC after it restarts
or its connection improves. Upon receiving the RPC (after
it has recovered), the client immediately sends the server a
Register RPC. The server then adds the recovering client to the
visibility set, broadcasting another Visibility-Set-Change RPC,
and “catches up” the client by sending it the data required to
reconstruct its views.

In order for the server to provide a recovering client with
the data it needs to reconstruct its views, the server has to store
the right data internally. As mentioned in Section 4, the server
explicitly stores a suffix of the global operation log along
with a checkpoint copy of shared state representing the prefix
of the log. We select the checkpoint prefix to be the prefix
containing only operations that are visible to all clients. (Note
that this prefix does not necessarily correspond to the Visible
view of any client; also, note that one or more operations
in the server’s suffix list may be visible to all clients, even
though the first operation in the suffix list is not.)

To “catch up” a recovering client, the server sends the client
its copy of state, its suffix list of operations, and the sequence
number of the latest operation from the client that is visible.
The client uses these items to reconstruct its Visible copy of
state and its Authoritative and Durable operation lists.

Figure 8: The interior design app’s output-rendering code.
the latency window in which they see uncertainty. Or, if users work with separate items and use the interior design app on reliable workstations, the Durable view could even be suitable. Thanks to Hercules’s interface, customizing the view used would require modifying only a single line of code.

6.2 Chat App

The chat app provides a shared chat room. Users send messages that are appended to the log, and clients display the most recent messages in the log. Figure 9 shows a screenshot of the chat app client, and Figure 10 shows the client’s code for rendering output.

The chat app uses all four state views. When displaying messages that have been added to the Durable, Authoritative, or Visible views, the client draws an icon alongside the message to indicate its status.

The chat app takes advantage of the log structure of its shared state to identify differences between the views. It renders the ten most recent messages in the Submitted view’s chat log. If those messages are present in the other views, then they will be located at the same index of the chat log in those views. As a result, the client identifies the index of the first message to display, checks for any messages at or after that index in the Visible view, and renders those messages with a “visible” icon next to them. The client then moves onto the Authoritative view, starting at the index where it left off, and repeats the same process, before moving onto the Durable and Submitted views.

6.3 Spreadsheet App

The spreadsheet app provides a shared spreadsheet that users can collaboratively edit. Each user has a cursor, which they move with the arrow keys, and their edits affect the cell underneath their cursor. The client shows the position of the user’s cursor by drawing the selected cell with a red border, and it shows the position of other users’ cursors with a blue border.

The spreadsheet app uses the Submitted and Visible views. The client indicates to the user when their modifications to a cell are not yet visible to other users, by drawing the cell’s text in red, and when their cursor movement is not yet visible, by

```csharp
// Display the 10 most recent messages
int i = submittedMsgs.Count - 10;
for (; i < visibleMsgs.Count; i++) {
    DrawMessage(visibleMsgs[i], /* message */
                i - firstIndex, /* position */
                VISIBLE); /* status */
}
for (; i < authMsgs.Count; i++) {
    DrawMessage(authMsgs[i],
                i - firstIndex,
                AUTHORITATIVE);
}
for (; i < durableMsgs.Count; i++) {
    DrawMessage(durableMsgs[i],
                i - firstIndex,
                DURABLE);
}
for (; i < submittedMsgs.Count; i++) {
    DrawMessage(submittedMsgs[i],
                i - firstIndex,
                SUBMITTED);
}
```

Figure 10: The chat app’s output-rendering code.

```
Part | Qty | Price
--- | --- | ---
Gear |    |     
Bolt |

(a)

Part | Qty | Price
--- | --- | ---
Gear | 4   |     
Bolt |

(b)

Part | Qty | Price
--- | --- | ---
Gear | 4   |     
Bolt | 2   |

(c)
```

Figure 11: A sequence of screenshots from the spreadsheet app client as two users edit adjacent cells. In (b), the user has entered a value in the “Qty” column and moved the cursor to the “Price” column, but those actions are not yet visible to other users. (Colors edited for better greyscale contrast.)
drawing the old cursor position with a thinner border along-
side of the updated cursor. Figure 11 shows this behavior.
Like the interior design app, the specific views used by this
app can be easily customized.

Figure 12 shows the spreadsheet app client’s code for draw-
ing the display. The spreadsheet app takes advantage of the
two-dimensional tabular structure of the shared state when
identifying differences between views. The client iterates over
each cell in the spreadsheet and figures out whether the cell
is selected by its user in the Submitted view, by its user in
the Visible view, or by another user, and draws the cell border
appropriately. To determine how to render the cell text, the
client simply checks if the cell contents differ between the
two views.

Cloud Firestore version. We built a version of the spread-
sheet app on top of Firebase Cloud Firestore [11] to compare
the Hercules programming experience with that of a popular
distributed storage system. Our Cloud Firestore version of the
spreadsheet client app required 2.4x as many lines of code
as the Hercules version (718 LOC vs. 302 LOC). The Cloud
Firestore version provides the equivalent of the Submitted,
Authoritative, and Visible views. The Cloud Firestore C# li-
brary does not provide the optimistic local invocation feature
described in Section 2, so we did not implement the Durable
view.

Most of the additional lines of code implement protocols
for maintaining the Submitted and Visible views. The Cloud
Firestore version defines in-memory representations of shared
state and operations, maintains a list of submitted operations,
and reads and writes an authoritative operation list and a set
of per-operation visibility records in Cloud Firestore. It uses
event listener callbacks to recompute its views of shared state,
mark operations as visible, and update its submitted operation
list when its operations are acknowledged by the backend.

Although our Cloud Firestore app is already much larger
than the Hercules app, it is missing several features that Her-
cules provides for free. It does not garbage-collect old oper-
ations and visibility records, it does not have the ability to
recover from a client failure, and it requires users to manu-
ally join and leave the visibility set. Furthermore, it is brittle
and less efficient than the Hercules version. Cloud Firestore
claims to only support one update per second for a given docu-
ment [10], and based on our experience, the backend enforces
that limit by arbitrarily dropping updates when updates are
too frequent. To avoid losing writes to the document holding
visibility records, we changed the client to mark operations
as visible in batches every two seconds. This change makes
the Visible view slower, and it also does not scale well—we
tested it with two clients, but a higher number of clients would
require an even longer batching period.

7 Evaluation

Our evaluation showed that Hercules’s baseline performance
exceeds that of comparable apps built on standard cloud stor-
age systems, that Hercules’s side-by-side batching enables it to
deliver good performance in high-throughput scenarios, and
that its recovery protocol allows the system to progress in the
presence of client failures.

For many of our experiments, we used a benchmark ap-
lication in which the shared state is a large byte array, and
clients invoke an operation that increments elements of the
array. Each client runs an open loop in which it invokes the op-
eration, calls GetState(), and then sleeps for some amount
of time. The size of the byte array, number of array increments
performed by the operation, and loop sleep interval are all
configurable parameters.

7.1 Setup

We ran our quantitative experiments on Google Compute
Engine virtual machines. We used an n1-highcpu-4 VM (4
virtual cores, 3.6 GB RAM) located in the us-west1 region
and an n1-highcpu-8 VM (8 virtual cores, 7.2 GB RAM)
located in the us-east1 region. Each VM ran Windows Server
2019. The round-trip latency between the two VMs was 67ms.
Unless otherwise noted, we ran Hercules clients on the n1-
highcpu-8 VM and the server on the n1-highcpu-4 VM.
7.2 Baseline Performance

We compared the normal-case performance of our spreadsheet app to the Cloud Firestore version. For each version of the app, we ran two clients, and on one of those clients, we performed a series of inputs simulating normal user behavior, in which we entered text in every other row in two columns. We measured the delay between when the client submitted each operation and when the operation was reflected in its state views.

We located our Cloud Firestore instance in the same Google Cloud region as our Hercules server. There was no option to locate the Cloud Firestore instance in the us-west1 region, so for this experiment, we put the Cloud Firestore instance in the us-east1 region, and we swapped the VMs so that the n1-highcpu-4 VM ran all clients and the n1-highcpu-8 VM ran the Hercules server. We used Remote Desktop to control the client apps.

Figure 13 shows the average delay for the Hercules and Cloud Firestore versions of the spreadsheet app, where the averages were computed after discarding the first and last 25% of operations. The figure only reports Authoritative and Visible delays, because the Cloud Firestore version does not implement the Durable view; the average delay for the Durable view in the Hercules app was 2.2ms.

For the Hercules app, the Authoritative delay is roughly the RTT between the client and server, and the Visible delay is around two times the RTT, as expected. The Cloud Firestore app’s Authoritative delay is 1.7x higher than the Hercules app’s delay, although this is not quite an apples-to-apples comparison—the Cloud Firestore backend may be replicating updates or performing other useful work with the added time. Its Visible delay is 9x higher, due to the batching described in Section 6 that is required to avoid dropped writes.

7.3 Effect of Client Rebase Batching

Client rebase batching allows Hercules deployments to maintain good performance as system load increases. Figure 14 shows how rebase batching prevents performance from collapsing as a Hercules deployment expands to include more clients. The graph plots the time to complete an experiment in which each client submits operations and then waits for all operations from all clients to be reflected in its Visible view. For this experiment, we used our benchmark application, with a 100KB shared byte array; each client submitted 1000 operations, sleeping for 20ms between each operation, and each operation performed 500 array increments.

Without rebase batching, each individual client eventually becomes overwhelmed as the number of incoming operations from other clients increases. Each client must perform a rebase before acknowledging each remote operation, so when the time required for a rebase exceeds the rate at which clients submit operations, a vicious cycle results, where operations pile up in the Authoritative lists and further increase rebase times. With rebase batching enabled, each client must execute the operations in its operation lists at a lower, fixed frequency, instead of every time a remote operation arrives. As a result, performance remains stable as system load increases.

7.4 Recovery from Client Failures

Figure 15 shows how one client’s failure impacts another client’s state views. The graph plots the delay between when a client submits an operation and when it sees that operation reflected in its state views. For this experiment, we ran the benchmark application with four clients. The shared byte array was 10KB in size, and each operation performed 1000 array increments; each client slept for 20ms between operation invocations.

At around the 20-second mark, another client is killed. The measured client’s Visible view experiences a delay spike at that point, as the system waits for the visibility set timeout to kick in, but the Authoritative view is unaffected. Note that even though there is a longer delay before updates make it into the Visible view, the client can still read that view with low latency.

8 Related Work

Global Sequence Protocol (GSP) [4] is a model for replicated shared data that represents shared state as a sequence
of updates. Clients distinguish between a *known* prefix of globally-agreed-upon updates and a *pending* sequence of local updates. Hercules’s model of shared state resembles GSP’s, and Hercules makes use of GSP’s insight that “including application-specific update operations in the data model is a powerful trick.” In contrast to GSP, Hercules exposes additional sub-sequences of updates, one of which (the *Visible* view) requires substantial backend support, and Hercules emphasizes providing its state views on-demand with low latency.

The Correctables abstraction [12] introduces the concept of executing operations on shared state with multiple consistency levels and incrementally providing feedback to applications. It provides that feedback in the form of per-operation callbacks, which we argue is the wrong abstraction. With per-operation feedback alone, apps cannot determine which portions of shared state reflect uncertainty. Also, if apps need immediate access to strongly consistent snapshots of state when rendering output, then they need to manually schedule strongly consistent read operations in the background and cache the results.

Bayou [25] introduces many ideas that Hercules builds upon. It provides two views of its data store reflecting different consistency levels, and its sample applications use those views to expose uncertainty to users. The differences between Hercules and Bayou result from changes in the mobile computing landscape over the last several years. Bayou assumes that network partitions are a common occurrence, and its concerns with latency are limited to ensuring that clients can make progress during partitions. In contrast, Hercules targets an environment in which full connectivity is the normal case, and client read latency is of paramount importance. These assumptions motivate Hercules’s *Visible* and *Submitted* views, respectively.

Multiple systems build on Bayou to provide flexible consistency guarantees [3, 27], but these systems do not ensure low client read latencies, and they do not provide multiple simultaneous views of the same data item. Other systems that provide multiple consistency levels [17, 18] similarly expose only one view at a time.

Some systems use speculation techniques to mask network latency and improve the responsiveness of distributed apps in particular domains. VNC/SRD [15] adds client-side speculation to a VNC remote desktop system, and Outatime [16] applies server-side speculation and client-side error correction to a cloud gaming system. Instead of exposing uncertainty to users, these systems directly display a tentative view, although Outatime includes multiple techniques for compensating for mispredictions in the client.

Several storage systems or frameworks target distributed mobile applications, such as Simba [8] and Diamond [28]. These systems support multiple consistency levels, but they only provide a single view of any particular data item to application clients, preventing them from exposing uncertainty to users. They also require a client’s own operations to go through durable storage before being reflected in its reads, which risks violating the responsiveness requirements of real-time interactive apps, and they do not provide any information about the visibility of one client’s operations to other clients. Other work targets distributed file systems [21, 26, 29], where accesses are more sporadic and coarse-grained, and the focus is on conserving bandwidth.

Many areas of active research in distributed systems are complementary to Hercules. Geo-distributed storage systems with strong consistency guarantees [2, 5, 14] could be used as the backend component of a Hercules deployment, and even systems that sacrifice consistency for performance [6] could be used if they provide stale, strongly consistent snapshots. On the client side, techniques from conflict-free replicated data types [24] could minimize the computing burden by avoiding the need to re-compute state views.

## 9 Conclusion

Real-time interactive apps require a client-side cache of shared state that simultaneously exposes multiple views of shared state, that trades off staleness for consistency instead of latency, and that provides new views at the extreme ends of the consistency/staleness continuum. Hercules meets these needs with a system that is performant and robust to failures.

Hercules leaves open many avenues for future work. Human-computer interaction research could explore which views of shared state best fit user preferences for how distributed apps should behave. Future distributed systems work could examine how to adapt Hercules’s state views to more complex backend architectures, which may enable new state views or require different types of performance optimizations. Given that some but not all of Hercules’s features would also be useful for traditional distributed apps, it is worth investigating whether a flexible system could support both real-time interactive apps and traditional apps, changing its guarantees and performance characteristics depending on app requirements.
References


