Mobius: Unified Messaging and Data Serving for Mobile Apps

Byung-Gon Chun
Yahoo! Research
Santa Clara, CA
bgchun@yahoo-inc.com

Carlo Curino
Yahoo! Research
Santa Clara, CA
krl@yahoo-inc.com

Russell Sears
Yahoo! Research
Santa Clara, CA
sears@yahoo-inc.com

Alexander Shraer
Yahoo! Research
Santa Clara, CA
shralex@yahoo-inc.com

Samuel Madden
MIT
Cambridge, MA
madden@csail.mit.edu

Raghu Ramakrishnan
Yahoo! Research
Santa Clara, CA
ramakris@yahoo-inc.com

ABSTRACT

Mobile application development is challenging for several reasons: intermittent and limited network connectivity, tight power constraints, server-side scalability concerns, and a number of fault-tolerance issues. Developers handcraft complex solutions that include client-side caching, conflict resolution, disconnection tolerance, and backend database sharding. To simplify mobile app development, we present Mobius, a system that addresses the messaging and data management challenges of mobile application development. Mobius introduces MUD (Messaging Unified with Data). MUD presents the programming abstraction of a logical table of data that spans devices and clouds. Applications using Mobius can asynchronously read from/write to MUD tables, and also receive notifications when tables change via continuous queries on the tables. The system combines dynamic client-side caching (with intelligent policies *chosen on the server-side*, based on usage patterns across multiple applications), notification services, flexible query processing, and a scalable and highly available cloud storage system. We present an initial prototype to demonstrate the feasibility of our design. Even in our initial prototype, remote read and write latency overhead is less than 52% when compared to a hand-tuned solution. Our dynamic caching reduces the number of messages by a factor of 4 to 8.5 when compared to fixed strategies, thus reducing latency, bandwidth, power, and server load costs, while also reducing data staleness.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Design, Algorithms, Experimentation, Performance

MobiSys'12, June 25–29, 2012, Low Wood Bay, Lake District, UK.

Keywords

Mobile cloud computing, data management, messaging, push notification, caching, mobile apps

1. INTRODUCTION

Increasingly, applications for mobile devices create and publish content that must be shared with other users in a timely fashion, or retrieve data personalized to a user's location or preferences from a cloud. Examples include Yahoo! applications like Yahoo! News, Sports, Finance, Messenger, Mail, Local, and Livestand, shop review applications like Yelp, crowd-sourced traffic collection applications like Waze, and social networking applications like Facebook, Foursquare and Twitter. Developing such applications atop create/read/update/delete (CRUD) APIs is challenging for a number of reasons: 1) cellular networks have high and variable network latencies and costly bandwidth, 2) mobile phones have limited battery-power, 3) network disconnections are common, and 4) server-side scalability and fault-tolerance are a requirement.

This forces developers to implement complex networking and data management functionality and optimizations, including: asynchronous interactions (to hide the cost of remote operations), client-side read caching (to reduce bandwidth and latency, while bounding data staleness), batching of write operations (to reduce power costs), protocols to arbitrate concurrent and disconnected operations on shared data, timeouts and reconnection strategies (to handle network coverage issues), notification mechanisms (to allow data to be pushed from the server side), and manual sharding of data and replication (to guarantee server side scalability and fault-tolerance).

We believe that developers should not have to worry about these issues when writing mobile applications. Unfortunately, existing frameworks provide limited support for addressing these challenges. On the one hand, RESTful backend storage services focus on well-connected hosts [16, 12] and provide scalability, but provide little support for data sharing across users, caching, write batching, and so on. For example, iCloud [5] only provides mechanisms for applications to backup a user's private state on servers; developers are left to implement the rest themselves. On the other hand, replication systems for mobile computing such as Coda, Ficus, Bayou, PRACTI, and Cimbiosys [37, 34, 49, 22, 41] support replication of data for sharing and disconnected operations, but they are not designed to manage data and notifications for a large number of users.

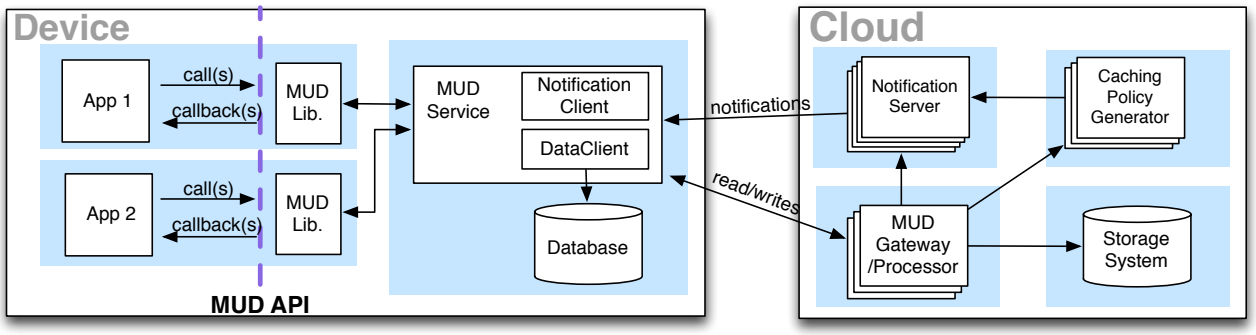

Figure 1: Mobius architecture.

In this paper, we bridge this gap with Mobius, a reusable data management and messaging platform that provides a simple, unified abstraction for mobile apps and addresses the challenges transparently underneath the abstraction.

Mobius employs a number of features that are motivated by data consumption, creation, sharing, and messaging requirements of Yahoo! mobile applications/platforms, which can be applicable to a broad set of applications. The requested features are:

- Reading/writing data and messaging
- Flexible predicate-specified real-time notification
- Writer's control on the visibility of its data
- Disconnected operation handling for reading and writing
- Transparent, intelligent caching/prefetching
- Hosted, scalable backend service

The first novel feature of Mobius is the *MUD* programming abstraction, short for "Messaging Unified with Data". MUD presents a table-oriented interface that supports asynchronous, low-latency access for messaging as well as for predicate-based lookups for data-intensive applications. Applications can read/write data or send/receive messages by reading and writing a shared logical table through the unified read/write API. Mobius implements the features that are needed by both networking and data stacks around this abstraction: disconnection-tolerance, support for changing IP addresses, caching, push notifications, and so on.

MUD tables are partitioned across mobile nodes and one or more server back-ends. From the perspective of the mobile app developer, accesses are done to local tables. These accesses may cause data to be fetched on an as-needed basis from servers, depending on the application's tolerance to stale data and the previous records that have been fetched. We support several kinds of read operations with a range of freshness guarantees.

The MUD API provides two less-conventional means for accessing data. First, it provides "writer predicates" where the visibility of writes in the table is constrained to readers who satisfy a particular predicate. For example, a user's writes may only be visible to other users in his family, or to users in a certain geographical region. This provides a form of predicate-based communication or late binding, where the exact recipients of a message are not known when the message is created. Second, the API provides a notification interface that sends applications "push notifications" when a particular record or range of a table is updated. This is useful both for inter-device messaging and for providing continuous queries, where the result of a query is updated as the records that satisfy it change. Our design and implementation of Mobius unifies back-end storage and processing for event notification and data serving, which can scale to a large number of concurrent push clients.

The second novel aspect of Mobius is support for *server-directed* caching and prefetching. To express preferences about what data should be cached, and to enable prefetching of data the application is likely to need in the future, the MUD API allows applications to specify "utility functions" that expose to our system the tradeoffs between various characteristics of a query answer, e.g., freshness, completeness, latency, bandwidth cost, and power cost.

Using this preference information, and knowledge of workload and system status, a server component can automatically devise intelligent caching strategies that maximize utility for each user. For example, knowing the frequency of updates, a caching policy can reduce power costs by suggesting that when asking for traffic data from the middle of the Nevada desert, it is best to serve data from the local cache since the chance of missing an update is minimal. Mobius provides several off-the-shelf utility functions that are designed to make it easy for programmers to implement common caching and prefetching values; e.g., programmers can provide a simple freshness threshold to ensure that content is more recent than the threshold.

We evaluate the performance of Mobius on several microbenchmarks to demonstrate that it does not incur substantial overhead over conventional APIs for data access and messaging, and to illustrate the potential for significant performance optimizations. In our initial prototype, remote read and write latency overhead is less than 52% when compared to a hand-tuned solution. We also show that our server-controlled client-side caching, based on simple machine learning based strategies, can reduce the number of queries pushed to the server by 4× to 8.5× for reads and 3× to 6× for writes, on real-world user traces from an iPhone crowd-sourcing application deployed to over 600 cars.

The rest of the paper is organized as follows. Section 2 presents the Mobius system architecture. Section 3 introduces the MUD abstraction, API, and usage. Section 4 presents Mobius write/read protocols, Section 5 presents Mobius continuous read/notification protocols, and Section 6 discusses server-directed, dynamic, context-based caching. We describe our implementation and experimental evaluation of the prototype in Section 7. Finally, we survey related work in Section 8, and conclude in Section 9.

2. MOBIUS ARCHITECTURE

Mobius is a unified messaging and data serving system for mobile apps that addresses the challenges discussed in Section 1. Mobius is motivated by data consumption, creation, sharing, and messaging requirements of current and future mobile apps; in particular, our design is guided by discussions with Yahoo! mobile app/platform developers. Many of them use their own custom layer

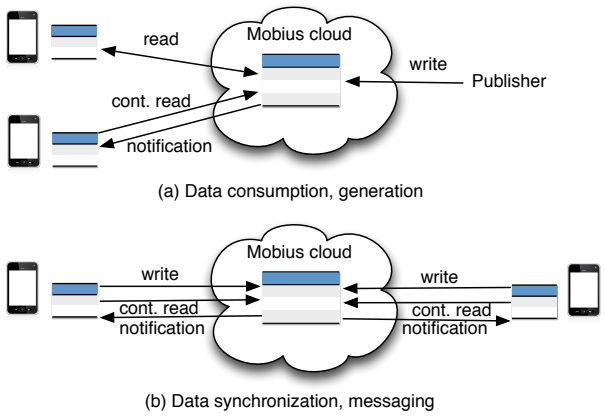

Figure 2: MUD usage scenarios.

to deal with caching, disconnection handling, push notification, or messaging in a limited fashion. Mobius provides all these features through the MUD abstraction.

Mobius consists of both client-side and back-end (cloud) infrastructure, as shown in Figure 1.

Client-side infrastructure: Services on the client include the client-side storage system, the MUD Service and application-level library code. The MUD Service is composed of a *notification client* and a *data client*. The notification client receives push notifications from the Mobius back-end infrastructure and forwards them to the appropriate MUD Library instance. The data client is responsible for sending data and queries to the back-end and for processing data sent in response. It is also responsible for query and update processing using client-side storage and as a runtime for the caching and prefetching policies devised by the server.

Back-end infrastructure: Services on the back-end include the back-end storage system, notification servers, gateways, MUD processors and caching policy generators.[1] Each *notification server* maintains a network connection to multiple client devices and is responsible for pushing notification messages to these clients. Device identifiers are used to partition clients among notification servers. The *gateways* accept client requests, forward them to MUD processors, and relay responses back to the clients. The *processor* performs query and update processing, including continuous queries, which are maintained as soft-state. Continuous query results are routed back to clients via the notification server. The back-end storage system stores records and provides scan operations that enable clients to re-sync when they connect. The storage system is partitioned by primary key ranges; each partition is handled by a single MUD processor, although each processor may be responsible for multiple partitions. Furthermore, storage is geo-replicated across multiple data-centers, each of which may run instances of notification servers and gateways. Our prototype targets PNUTS [30] as the back-end store, but other standalone storage systems may be used. The *caching policy generator* creates caching and prefetching policies for *each client* by mining data access patterns of *all clients* (this information is obtained from MUD processors). Clients use these policies to decide what to cache or prefetch and when. Next, we describe each component in detail, beginning with a description of the MUD data model and API.

[1]We also assume the existence of a naming service, which could be implemented atop the underlying storage service, or be provided using a pre-existing standalone system, such as ZooKeeper [17].

Key	Value				Predicate
	C1	C2	C3	C4	
k1	2	CA			F1 == 1
k2	3		Red		
k3		MI	Blue	100	F1> 100

Table 1: MUD table example.

3. THE MUD ABSTRACTION

MUD relieves mobile app developers from dealing with messaging and data management issues by providing them with a unified programming interface for both messaging and data. This interface is based on the simple abstraction of shared logical tables that can span devices and back-end clouds. Apps can read/write data or send/receive messages by reading and writing a shared logical table. In addition, apps can register continuous read operations and get notified when relevant updates are made to a table. Figure 2 shows different MUD usage scenarios. We also mediate access to tables stored solely on the device local storage, but this is trivial and not discussed further in this paper.

In the MUD data model, each logical table is a collection of records. Each record contains a (required) primary key, a sequence of attribute and value pairs, a predicate, and metadata (such as the record's version). Each attribute corresponds to a column in the logical table. Schemas are flexible: a record can have values for any subset of table columns. Mobius supports dynamically adding and removing columns from a table. An example of a table is shown in Table 1.

A unique feature of the abstraction is the symmetric nature of read and write operations. As usual, reads include a predicate specifying the attributes of records to be read. In addition, in MUD a writer (or sender) includes a *writer predicate* with the data. A reader (or receiver) includes *reader data* with its queries, such as authentication tokens, current location, time, or other context information. Writer predicates are then evaluated against the reader data. Hence, writers can control which readers may observe the written data. For example, an advertiser can send special coupons only to its premium users, and use user location and preferences to filter out uninterested users.

In its simplest form, a writer predicate does not require the *authenticity* of reader data. A writer predicate may be simply used to limit which readers can receive the data as an optimization (e.g., discount coupons for users in a particular location). These context-based predicates allow writers to *late bind* to readers that can see their updates, such as when the list of users in a particular location is not known at update time, or when adding a user to a group for advertisements. In more complex cases, a writer predicate can be an access control list (ACL) specifying clients that may read the record (note that, as mentioned in Section 3.1, each table has an ACL as well) when reader data's authenticity is verifiable. For example, only clients with proper identities can read the record (e.g., user messages in the writer's friend list).

We provide simple and yet well-defined consistency semantics (per-record *sequential* [39] and *fork-sequential* [40] consistency), and let app developers choose the appropriate consistency for their application. We achieve this by exposing read operations with different consistency levels. Applications can gracefully degrade consistency whenever latency or service availability are more important. Informally, our consistency guarantees define the order in which clients see updates, while allowing clients to return cached updates. In Section 4 we describe the different types of read and write operations.

Table manipulation
create(*tb*, *properties*)
drop(*tb*)
alter(*tb*, *properties*)
Table update
write(*tb*, *key*, *type*, *data*, *predicate*, *utility*)
delete(*tb*, *key*, *type*, *utility*)
Table query
read(*tb*, *predicate*, *type*, *data*, *utility*)

Table 2: MUD API.

Obviously, the back-end store must itself provide some level of consistency so that the system as a whole may be consistent. Interestingly, for simple read and write operations the back-end store need only guarantee eventual propagation of updates. That is, all non-faulty back-end storage replicas should eventually receive all record updates made to their partition, but the updates do not need to be sequentially consistent. In practice, we use PNUTS, Yahoo!'s geographically distributed data serving platform [30]. PNUTS provides *timeline consistency*: each PNUTS replica applies increasingly newer updates. We note that other systems, such as Amazon's SimpleDB, also provide the required back-end store functionality [12], while others, such as Microsoft's SQL Azure provide even stronger primitives, such as single-partition transactions [25].

3.1 MUD API

Operations in MUD are categorized into three groups: table manipulation, update, and query. Table 2 lists the supported operations (for brevity, in a simplified form). All operations are asynchronous and their results are returned through a provided callback interface.

Table manipulations include three methods: *create*, *drop*, and *alter*. *create* makes a logical table *tb* with the supplied *properties*, which may include an access control list (ACL), an optional schema, etc. An app that creates a table may later invoke *drop* to delete it. Finally, *alter* is used to modify the table's properties.

The ACL defines a security level of the MUD table, which can be public, app, or user. Our authorization follows OAuth [9], a well-known authorization protocol for publishing and interacting with protected data. For a user to access the table, the user must be authorized at the level of the table or higher. The public security level does not require any authorization. The app security level requires at least app authentication, and the user security level requires user authentication.

The *write* and *delete* operations get as a parameter the table identifier *tb*, a primary key of the record to be written or deleted, the type of the update operation (blind or conditional, described in Section 4) and a utility function. (We discuss utility in Section 6.) In addition, a *write* operation is passed the data to be written and a writer (or sender) *predicate*, that constrains the visibility of the written data to readers whose reader-data satisfies the predicate. The predicate follows the same structural rules as a read's predicate discussed below. If no record exists in *tb* with the given primary key, the write inserts it and otherwise updates the record.

A read operation passes a *predicate*, which is a boolean expression over the rows of a logical table *tb*, similar to a SQL SELECT statement. Specifically, the predicate may be a conjunction or a disjunction of conditions. Each condition imposes a boolean constraint on record key and attributes using binary operators: $=$, $\neq$, $<$, $>$, $\leq$, $\geq$, and a limited number of boolean functions and regular expressions on strings. In addition, a *predicate* may include user-defined functions (UDFs), such as aggregation or limited joins (for example, a UDF can summarize the values of a given attribute for all matching records). Such functions help conserve network bandwidth in cases where transferring records from back-end to the client may be easily avoided. The language is similar to Yahoo! Query Language (YQL) [16].

A *read* additionally takes *data* as parameter which typically contains (but is not limited to) information about the end-user or device. For example, *data* may include the reader's location, identifier and so on. A read retrieves all tuples that match its predicate, and whose write-predicate matches the read's *data*. Similar to updates, a *read* additionally includes a *utility* and a *type*. The read utility is analogous to the write utility, and is used for caching.

We support three different "one-time" read types (committed_wio, committed and uncommitted), described in Section 4, which guarantee different consistency semantics. A fourth type of read is the continuous read, which sets up a continuous query. A continuous read gets an optional *sync-timestamp* parameter (which does not appear in Table 2), that instructs it to perform a one-time read returning records no older than *sync-timestamp*, in addition to registering the continuous query. This may be used if currently stored records are of interest in addition to future updates or to restablish a continuous query after a failure or device disconnection. A continuous read always returns a *handle*, which may be later used to unregister or modify the subscription. To this end, we provide two auxiliary methods:

unregister(*handle*)
update-data(*handle*, *data*)

For example, consider a client interested in restaurants within 1 mile of its location. Such a client may register a continuous query, including its current location in the *data* of the read. As it moves and its location changes, the client invokes *update-data* to update its subscription with its latest location (although a subscription may be updated by unregistering and re-submitting it, this would require more messages as well as querying for any updates missed in the process).

For the convenience of app developers, higher level APIs are provided as a layer on top of the basic MUD API we presented above, as indicated in Figure 1. For example, we provide blocking operation APIs as well as a simple pub/sub API. In the future, even richer interfaces could be considered, such as an SQL-like declarative query language.

3.2 Using MUD

To showcase the benefits of Mobius and illustrate how to use the MUD API, we present three sample applications, social news reading, experience sharing and continuous journal editing, that combine messaging and data interactions and need late binding and caching.

Social News Reading: This application is a social news-reader application supporting instant social sharing and updates. The application supports reading news articles of users' interests, being instantly notified of any urgent news, writing user-generated content, and exchanging interesting news articles and messages with friends in real time. The social news-reader application can be easily implemented with the MUD API. The app developer first creates two tables, *news (topic, article)* and *chat (group, msg)*, using *create*. Suppose that Jane uses this app for sharing articles with her friends. Jane's application requests news articles on sports by issuing *read(news, topic='sports', committed, ...)*. It also registers Jane's interest in art articles by invoking a continuous read *read(news, topic='arts', continuous, ...)*. Whenever a new article becomes available (e.g., when the application's cloud service dis-

covers a new article online and makes it available to the app clients by invoking *write(news, unique article id, blind, [topic='arts', article='new exhibition'], ...)*), Jane's application is instantly notified. When Jane requests to share an article with her friends, the app performs *write*(*chat*, *unique message id*, *blind*, *[group='Jane's friends',msg=article id], ...)*, assuming her friends have already registered continuous reads on the *chat* table. The same table can be used for exchanging discussion messages. When Jane wants to send a local event message to her friends located within 1 mile from her current location, the app can perform *write(chat, unique message id, blind, [group='Jane's friends', msg=event url], $|receiver.loc - this.loc| \leq 1$ mile, ...)* where *this.loc* is Jane's location and *receiver.loc* is the location of a friend who sets up a continuous read to receive Jane's messages.

Experience Sharing: This application is aimed at enhancing the experience at large events, such as fairs or concerts. The first set of data consists of map overlays, schedules and other information provided by the organizers. These are queried by the users and can be cached, since their content is likely to be stable. This is achieved by issuing reads such as: *read(overlaymaps,lon>y1 and lon<y2 and lat>x1 and lat<x2, ..., freshness<24h, ...)* where *freshness<24h* represents a simple utility function that allows for heavy caching. Organizers also push timely news about schedule changes, and other unpredictable events to the users. They can do this by issuing simple writes. For example to notify every person over 21 years old of a free beer event one could write: *write(flashnews, pk1, blind, "free beer at south kiosk", receiver.age≥21,...)* to a table over which user devices have posed subscriptions, i.e., continuous reads. In this context predicates can be used to avoid broadcasting to the entire crowd, and can be applied to user profiles and context. Users contribute by messaging each other, observing friends location, sharing comments, tweets and pictures. All these use cases are easily supported with some combination of reads, writes and continuous reads. Furthermore, "relive the event" user experiences can be supported by issuing rich historical queries such as: "What was the name of the band playing when I sent that comment?," or, upon reading a geo-tagged tweet: "Tell me if anyone uploaded pictures taken at this time and place." Users can operate seamlessly on local data under disconnections, and the system syncs as soon as a connection is available.

Continuous Journal Editing: This application supports keeping context-tagged journals for the users. The application is implemented on multiple platforms — smartphones, tablets, and desktop machines. The users can update journals in any of the devices they use. For example, a user edits a journal on a tablet disconnected from the network. She then edits the journal with her smartphone on the way home and writes it to the back-end. When the user reads the journal at the tablet at home, she should be able to see both changes; i.e., the edits made on the smartphone should not be overwritten by the updates locally queued on the tablet when the tablet comes back online. This is achieved by using conditional writes such as: *write(journal, pk1, conditional, [content="about cafe A,..."], ...)*. Unlike blind writes, which simply overwrite the content, a conditional write succeeds only if the version of the record at the back-end is identical to the one locally stored on the device at the time the update is performed. Thus, whenever two conditional writes are made concurrently to the same shared journal, they conflict and one of the updates (in our case the one performed on the tablet) fails. The application issuing the failed update would then provide the user with information about the conflict so that she can reconcile them.

Given this high-level description of Mobius and the MUD API, we proceed to describe Mobius design in more detail. We start with write/one-time read operations, and then discuss continuous reads and caching. After that, we present the results of some initial experiments with our Mobius prototype.

4. WRITE/READ PROTOCOLS

Mobius provides app developers with the freedom to choose the appropriate consistency semantics for their application. The weakest consistency model our system provides is *fork-sequential consistency* [40], which is the guarantee provided when mobile devices execute operations without blocking under disconnection. As we restrict the types of records the client can read, we support stricter consistency semantics. Table 3 summarizes the consistency semantics Mobius provides. In the following, we describe different types of write and read operations, their consistency semantics, and protocols to implement them.

4.1 Operations and Consistency Semantics

Writes: Our system supports two record write operations: a "blind" write, which simply overwrites a record, and a "conditional" write, which requires the version of the record at the back-end to match the version in the client's local store. As demonstrated by the Continuous Journal Editing application in Section 3.2, conditional writes are especially useful for mobile applications.

Each record in Mobius has monotonically increasing versions, as described in detail later in this section. If the record being updated is not in the local store, a conditional write succeeds only if the record does not exist at the back-end store. When the record is written successfully at the back-end store, the record is *committed*.

The type of the write affects its liveness guarantee: A blind write invoked by a non-faulty client (that eventually becomes connected) is guaranteed to succeed regardless of the operations of other clients. Conditional writes have weaker liveness; while system-wide progress is guaranteed (some client's write will succeed), progress is not guaranteed for each individual client, as other clients may simultaneously update the same record causing a conditional write to fail.

Reads: We support three different one-time *read* operations: uncommitted, committed, and committed_wio.

A committed read returns a "recently" committed update; the notion of recent is briefly explained below. This read type provides sequential consistency [39]. Sequential consistency is equivalent to the concept of one-copy serializability found in database literature [23]. It guarantees that the real execution is equivalent to a sequential one, where the program order of each client is preserved.

With sequential consistency, the view of each client must preserve the precedence order of its own operations. However, as clients may execute operations asynchronously, without waiting for previous operations to complete, precedence order is only a partial order. Hence, the definition of sequential consistency does not restrict the order in which writes executed concurrently by a client may take effect.[2] We thus additionally support a slightly stronger notion of consistency, which requires the *invocation order* of writes to be respected. A committed_wio (committed with write invocation order) read thus requires that the returned committed record reflects all non-failed writes previously invoked by the client (and is therefore executed after all such writes complete, which may require blocking).

Intuitively, an uncommitted read is allowed to return uncommitted writes, even writes that may later fail (if some global constraints are not met). For example, a client reading and writing data while

[2]In fact, all definitions of sequential consistency we are aware of, assume that each client executes one operation at a time.

Read type	Semantics	Write visibility
uncommitted	fork-sequential consistency	pending writes
committed	sequential consistency	committed writes
committed_wio	sequential consistency with write invocation order	committed writes, blocks till client's previously invoked writes complete

Table 3: Consistency semantics

disconnected, or a client accessing data for which it is the only writer, may be fine with seeing data that it has previously written before it has been committed. Clients may also use uncommitted reads if they value latency more than consistency. This read type provides fork-sequential consistency [40]. Fork-sequential consistency allows client views to diverge when they execute operations locally, and yet guarantees that clients see writes in the same order. Specifically, it guarantees that if a client observes a write issued by another client, then their views are identical up to the write.[3] Formal definitions of the properties are presented in Appendix A.

All read types preserve the "timeline" of updates at the client, that is, a client is never returned an old version of a record after having seen a newer one. Note that this requires a client to maintain the last version identifier of each record it has seen, even if the record itself is deleted from its local store. This information may be garbage-collected when the client is sure that older versions of the record may not be returned by the back-end (for example, when the back-end indicates that all replicas are up to date). We note that while each read type returns increasingly newer versions of each record, this is not guaranteed across read types (for example, an uncommitted read may return a newer update than a subsequent committed read).

Stronger semantics: As we have already mentioned, we target PNUTS as the back-end store. PNUTS provides *timeline consistency* [30] on a per-tuple basis. Timeline consistency is a weaker variant of sequential consistency; it guarantees that each tuple will undergo the same sequence of transformations at each back-end store replica.[4] With some additional support from the back-end, it is possible to guarantee stronger consistency with only minor changes to our protocols. Specifically, if the back-end supports queries of the *latest* value of each record, the protocols guarantee linearizability [35] instead of sequential-consistency and fork-linearizability [24] instead of fork-sequential consistency. The only change required on the client-side is that committed and committed_wio reads can no longer be served locally; linearizability requires a read to return the latest complete update. Note that even if the back-end supports querying for the latest update, such queries usually require contacting the record master or multiple back-end replicas, and are therefore more expensive than queries that do not have to return the latest update.

4.2 Protocols

Next, we describe the protocol for executing reads and writes. In the text below, we denote client C's local store by DB_C.

Writes: A write is executed in two stages at the client:

1. insert R = *(key, data, predicate, ver, pver, ts, committed)* to table *tb* in DB_C, where *ver* is a version of the update, *pver* is the previous version of this record in DB_C (if any), *ts* is the local time of this update and *committed* is *false*. Once insert completes, the operation returns.
2. send the record R to a MUD gateway. Upon receiving a response *callback* is invoked and, if the update was successful, *R.committed* is set to *true*, *R.ts* to the current local time, which is used for local freshness checking, and *R.ver* is set to the global version assigned to this update by the back-end store.

When an update is submitted, *ver* is assigned a local version. Committed updates have versions assigned by the back-end, which we call *global* versions (for example, in PNUTS, global versions are simply update counters). Note that a local version is itself a triple *(base, netid, seq)*, where *base* is a global version of the last committed update to the record in DB_C (or 0 if *key* is not in DB_C), *netid* is the client identifier, and *seq* is the number of pending (uncommitted) updates issued by the client. *pver* is the version (whether global or local) of the previous update to the record in DB_C. *pver* of an update is *ver* of the preceding update to the same record stored in DB_C (0 if *key* is not in DB_C—in this case the conditional write is treated as a conditional insert).

For a conditional write to succeed, the writer must have the necessary permissions,[5] and *pver* must be equal to the current version of the record at the back-end. This check is performed atomically and therefore requires back-end support for compare-and-swap updates as provided, e.g., in SimpleDB, Windows Azure Storage or PNUTS.

While a client is disconnected from the system, its updates are stored locally and then sent to the back-end when the client reconnects. Sending updates to the back-end can be done for every update separately, or periodically, batching multiple updates together. Updates are sent to one of the MUD gateways, which forwards each request to the appropriate MUD processor responsible for the partition to which the updated record belongs. Each MUD processor is responsible for many partitions. The MUD processor forwards the write and read requests to the back-end store, maintaining any ordering constraints required for the write and committed_wio read requests. Once a write is executed against the store, the completion status is sent back to the MUD gateway which in turn returns a response to the invoking client.

In case a conditional write fails because of a conflicting update (the current version of the record at the back-end is different than *pver*), the client is given a chance to resolve the conflict. Specifically, the client is sent (a) the conflicting update found at the back-end; (b) last update to the record in DB_C and (c) the previously committed update in DB_C (*base* when the update was invoked). Given this information the client can resolve the conflict and issue

[3]Originally, "forking" consistency conditions were used in the Byzantine system model, where a malicious server may "fork" the views of two clients. We are not aware of previous work that used the conditions to model disconnected client operations in a non-malicious setting.

[4]PNUTS does not guarantee that the "timeline" of updates is preserved *at the client*, as a client may connect to different back-end replicas.

[5]Interactions between conditional writes and writer predicates have security implications, and must be handled with care.

a new update, or simply discard its local changes. This method is akin to the conflict resolution employed by software revision control systems such as SVN. Note that when such conflicts occur, the pending update is deleted from DB_C and the conflicting update returned from the back-end is stored locally.

Deletes are a special type of update. Locally, they are treated just like any other write, except that the data being written is a special *tombstone*, which marks the record for deletion. When such updates are executed on the back-end, they are translated to blind or conditional deletes.

Reads: Like updates, read operations involve two stages. First, the client attempts to answer the read locally. It evaluates the query against DB_C and identifies the primary keys of matching records. If rules to decide local serving do not allow serving some of the matching keys locally, or if it is possible that there are records which may match the query but are not stored in DB_C, the query is sent to a MUD gateway.

To illustrate the flow of read operations with the rules, we use a maximal acceptable record age (expressed as a parameter *max-age*) as a simple utility function (in Section 6 we discuss more general utility functions). The following auxiliary procedure is used to decide whether a record R in DB_C is "fresh" enough (*time* is the current local time):

```
1: procedure isFresh(R)
2:     if R.committed=false or time−R.ts<max-age then
3:         return true
4:     else return false
```

As an extension, one can use utility functions such as the ones employed in bounded inconsistency models (e.g., numerical error, order error, staleness) to define what records are acceptable in an application-specific fashion [51].

For each such key k of matching records, let R be the latest update to k in DB_C (a pending update is always considered more recent than a committed update for the choice of R) and R_{base} the last *committed* update to k in DB_C. For each of the read types, we now specify the conditions for serving k locally:

- *uncommitted*: return R if *isFresh*(R).
- *committed*: return R_{base} if *isFresh*(R_{base}).
- *committed_wio*: return R_{base} if *isFresh*(R_{base}) and $R{=}R_{base}$.

When the MUD gateway receives the read, the gateway breaks the read into multiple sub-queries, each corresponding to a different partition, and forwards each sub-query to the MUD processor responsible for that partition. The read type determines how the processor handles the read: *committed_wio* reads are queued after writes and other committed_wio reads previously received by the processor. Uncommitted and committed reads are maintained separately—such reads may be executed right away without waiting for writes to complete. When processing a client's read, the processor queries the back-end store and sends back all matching records to the MUD gateway through which this read was submitted. The gateway then forwards any returned data to the invoking client. When all processors to which sub-queries corresponding to a read have returned a response, the gateway sends a special completion message to the client.

Clients filter received records corresponding to updates older than they have already seen. This check is performed using the version *ver* of a record. If a newer record is received, it is both returned to the application through the MUD Library, and stored in DB_C (in practice, if the data is large it does not have to be stored locally—we may store just its version information). In addition to any pending updates, at most one committed version of each record is kept in DB_C (the one having the highest observed version).

5. CONTINUOUS READ PROTOCOL

In this section, we describe the processing of continuous reads and push notifications sent to clients. The notification flow is exercised when a registered continuous read for a logical table matches changes to the table's records.

As with a normal read, a continuous read is submitted to one of the MUD gateways, which splits it into one or more sub-queries, each of which corresponds to a different partition and is sent to the corresponding MUD processor. A continuous read is first executed against the back-end store just like any committed or uncommitted read. A minor difference is that, unlike normal reads, a continuous read includes an optional *sync-timestamp* parameter, which determines the maximal age of existing records which is of interest to the client (if *sync-timestamp* is set to zero, no existing records are queried). These records are returned through the response flow of a normal read. Continuous reads are then maintained as soft state at MUD processors.

When a write operation updates a record, the MUD processor of the corresponding partition is notified. It then matches the updated record and all relevant continuous reads. Finally, for each such continuous read, it looks up the notification server responsible for push notifications to the client that registered the read, using a unique client identifier. The MUD processor then forwards the updated record to all relevant notification servers, including both the client identifier, *netid*, and the registering app identifier, *appid*. Each notification server in turn forwards the write to notification clients. When a client is not reachable (e.g., it is turned off or disconnected from the network), the notification is stored in a message queue for a limited amount of time. Eventually, the continuous read is terminated by the server. Upon reconnecting, the client detects this, and re-submits the read if needed.

A *netid* is a unique and fixed client device identifier (unrelated to a client's IP address, which may change over time). Since multiple apps may share the same device (and the notification channel), the pair (*netid*, *appid*) is used to uniquely identify the origin of a continuous read. We call such pair an *endpoint*. A notification server exposes a low-level API to other servers, allowing them to send messages to endpoints. Each server is responsible for a range of *netid*s. This mapping is maintained by the name service and cached at each notification server.

Each client maintains a persistent connection to a notification server for push-notification messaging. The client locates a notification server by performing a DNS resolution of a publicly known notification service domain name. It is possible that this server is not responsible for the client's *netid*, in which case the client is redirected to the appropriate notification server. After establishing a channel, the notification client sends periodic heartbeat messages (e.g., once every 20 minutes). Such heartbeats are used by the notification server to detect a client's failure and garbage-collect its registered continuous reads. Whenever the IP address of the client device changes, the client re-establishes a persistent connection to the notification server. When the notification server pushes a message to the client, the notification client routes the message to the appropriate MUD Library using the *appid* included in the message.

6. CACHING

Our consistency semantics restrict the allowed ordering of updates seen by clients, and yet are amenable to caching, allowing

our system to choose when to serve a record using the device's local store and when to fetch a record from the back-end, and similarly when to propagate writes to the server. Hence, appropriate caching policies need to be devised to complement the consistency semantics with proper control over the tradeoffs between performance and answer quality.

We model this tradeoff as a utility maximization problem and exploit a developer-provided *utility function* in order to choose the appropriate execution plan for each query (e.g., whether to serve from client-side storage or fetch remotely), in a way that maximizes utility.

Given a query Q, and an execution plan P, a utility function is defined as: $Utility(Q, P) = Value(Q, P) - Cost(Q, P)$ where $Value$ is a function that assigns a numerical value to various quality characteristics (e.g., freshness, completeness, bias) of the answer to query Q as produced by plan P, and $Cost$ is a function assigning a numerical value to the costs (e.g., latency, bandwidth, power) of executing query Q according to plan P.

We aim to provide developers with a library of common utility functions to choose from, while allowing advanced developers to specify their own. An example of a common utility function is the classic notion of cache *freshness*, where $Value$ is defined as a step function—assuming value zero when Q is answered using locally cached data older than some threshold time t_0 and a large constant value otherwise—and $Cost$ is defined as a linear function of query latency. Under this utility function, answering from the client-side store yields higher utility whenever locally stored data is "fresh enough", while fetching data remotely is only beneficial when the required information does not exist locally or is too stale (otherwise we incur unnecessary costs). More complex utility functions accounting, for example, for battery and communication bandwidth can be easily devised.

Caching policies are most often implemented as fixed heuristics, hard-coded in the application logic at design time. Our system does not follow this convention and instead supports dynamic, context-aware, and globally-selected caching policies. Our architecture is designed so that the client runtime (specifically, the caching decision logic in the MUD data client) is a simple executor of cache policies that are determined at the server-side back-end. A server-side *caching policy generator* has a unique vantage point to determine the most efficient caching strategies since it has access to the usage patterns of all users (and can potentially collect such information across applications). Usage or access patterns include (but are not limited to) read/write ratios, temporal access correlations, current system conditions (e.g., network congestion, bandwidth cost, etc.), and user context (e.g., location). Taking such information into account allows for more informed caching decisions. For example, for a check-in application such as FourSquare, the server-side policy generator can use the density of check-in reads and updates in a certain area to determine how frequently a user's cache should expire and how often a user should push updates to the back-end. For example, when traveling to a destination where the user has no friends, a user's device could push his check-ins lazily and answer mostly from its local store, significantly reducing battery and bandwidth costs with no significant impact on utility.

Note that MUD notifications may be leveraged for pushing caching policies *dynamically* to clients. For example, in a map/-traffic application, traffic may increase dramatically following an accident, in which case it is important that users increase their frequency of traffic reports and get updated frequently on the developing conditions. A caching policy generator can react by increasing the poll and update rate for all devices in the vicinity of the accident.

Dynamically and globally coordinated caching policies have the potential to significantly outperform hard-coded heuristics (typically used today). In our current prototype, we cast the read cache policy selection problem as a classification problem: the features include the query predicates and user context, and the classification labels indicate whether a query should be served from the local cache or be sent to the server. In this initial implementation, the utility function is provided in the form of a cost-matrix, that indicates to the system the relative impact on utility of stale data versus the cost of unnecessary queries sent to the server. In our experiments, we use a real trace of user GPS locations and queries, derived from the CarTel application [36], to train a cost-sensitive classifier based on a J48 decision tree, and demonstrate that even our simple initial solution leads to up to a $4\times$ reduction in the number of queries sent to the server (providing a significant improvement in read latency, and reducing bandwidth), and reduces by $3.5\times$ the number of queries serving stale data (thus increasing the perceived data quality), when compared to a rather common baseline caching strategy.

The output of the policy selector is a trained decision tree model that is shipped to the client through our notification channel. For each query the client-side runtime tests the decision tree, providing the current user context (e.g., location and time) and the query predicates (e.g., geographical area of interest and time window). The decision tree determines whether to answer locally or remotely.

We applied similar techniques to automatically devise a write caching strategy. The goal is to batch as many write operations as possible in a single network message, reducing the cost of communication [21], while trying to guarantee that most reads will observe all previously generated writes. Again, we can cast this as a classification problem, using cost-sensitive decision tree classifiers, where we try to estimate the maximum delay acceptable for each write operation if it is to be delivered to the server in time to be observed by the earliest subsequent read. The utility function provided here specifies the relative cost of delaying a write for too long (thus impacting quality of reads) versus delaying it insufficiently, thus missing batching opportunities. Section 7.3 shows experimentally that this leads to a $3\times$ reduction of the number of messages sent, while maintaining the number of queries receiving partially stale answers unchanged.

The fact that these techniques do not require application-specific models (e.g., we do not model user mobility), but instead use read-/write ratios and their correlation to predicates and contextual variables suggests it is possible to build general purpose cache policy selectors.

In future work, our architecture will allow us to explore many other interesting optimization strategies, such as:

- cross-app synchronization of recurring reads and writes
- cross-app prefetching and piggybacking
- dynamic cache eviction policies
- gracefully degrading answer quality when disconnected
- push-based cache maintenance
- dynamically changing caching strategies
- compaction of outstanding updates stored locally

Addressing such challenges at the platform level provides much more than simple factoring of the functionalities in a common layer. We have a vantage point that is unique, and allows us to exploit machine learning and statistical techniques to improve the performance of our system in a much more globally-optimal, context-aware, and dynamic way. The simple strategies we implemented in

our prototype are only used to demonstrate some of the opportunities in this space, and a more complete investigation is part of our future research agenda.

7. EVALUATION

In this section, we describe a series of experiments we ran to evaluate our Mobius prototype. The goal is to demonstrate that:

1. The Mobius architecture and MUD API do not introduce significant additional client-side latency when sending messages or accessing local storage.

2. Utility-based caching using a server-generated caching policy can offer substantial reductions in access latency and total amount of network I/O versus simpler, local-only caching policies.

3. The Mobius back-end scales to large numbers of partitions, allowing developers to finely partition (and, by extension, parallelize) workloads.

7.1 Implementation and Experiment Setup

We implemented our prototype of Mobius using an event-driven, asynchronous architecture. We built Android and Java MUD client libraries and created an Android service that handles notification and data serving, which is shared by multiple apps running on the device. The app-side library talks to the service through Binder. The service stores data locally in SQLite [14]. Requests are handled by asynchronous Futures, and responses are returned by local callbacks. Requests from clients to gateways are encoded as JSON and are sent over HTTP. The MUD gateway, processor, and notification server are all built on JBoss Netty [6]. The protocol between notification client and server is a custom protocol optimized for this purpose.

Our prototype stores application data in JSON format, and supports queries written in a subset of the MVEL [8] expression language. We use a bLSM index for server-side storage [43]. Rather than attempting to optimize read queries using conventional database optimization techniques, we assume the underlying data (and associated queries) are finely partitioned, and chose an implementation designed to scale well with the number of partitions. This prototype comprises approximately 9,000 lines of Java code.

We ran end-to-end experiments by running servers on a machine located at UC Berkeley. The machine is equipped with two Xeon X5560 CPUs and 12GB RAM, running Ubuntu server 11.04. We ran our mobile apps for benchmarks on a Samsung Galaxy Nexus running Ice Cream Sandwich with a Verizon 4G LTE connection and a Nexus S phone running Gingerbread with a T-mobile 3G connection. We ran the storage backend experiments on a machine with a single Intel Core i7 2600 (3.4GHz) CPU, 16GB of RAM, and Ubuntu 11.04. This machine is equipped with a two-disk RAID-0 consisting of 10K RPM SATA disks.

7.2 Latency

To understand the latency overheads, we compared Mobius with baseline systems that directly execute local or remote SQL statements without considering consistency, scalability or fault tolerance. We ran microbenchmarks to measure latencies of local/remote writes, and local/remote reads. The baseline system for local writes/reads performs operations directly on local SQLite. For remote writes/reads, the baseline system sends requests to an HTTP server that executes the operations on a MySQL server at the backend. Note that the baseline systems are representative of typical developer choices. In contrast, Mobius also maintains a local copy of both read and written data. This allows local query serving.

Table 4 shows the latency overhead of Mobius relative to the baseline systems. We measured the mean latency of 500 requests.

Galaxy Nexus	Mobius Latency (ms)	Baseline Latency (ms)
Local write	94.10	64.99
Local read (cache)	40.58	2.89
Remote write (4G)	281.74	202.66
Remote read (4G)	301.94	198.08
Nexus S		
Local write	141.69	102.02
Local read (cache)	51.47	10.54
Remote write (3G)	540.07	1364.45
Remote read (3G)	364.02	1276.38

Table 4: Latency of MUD operations.

With a 4G connection, for remote writes, Mobius has up to 39% overhead, whereas for remote reads, Mobius has up to 52% overhead. While we plan to optimize our prototype to further reduce latencies, we believe that even the current overhead is acceptable, especially given that many applications are designed to deal with much higher latencies, and higher variances of latencies that occur on wireless connections or as users move from one location to another. Local writes and reads are slower with Mobius since Mobius performs these operations with asynchronous Futures unlike synchronous baseline SQLite systems. With the 3G connection, Mobius's remote writes and reads are much faster than the baseline systems. This demonstrates that simple baseline systems are hard to get right when environments change. The reason Mobius does better is that it maintains persistent HTTP connections, but the baseline system does not, re-establishing a connection each time a request is made. On 3G the additional round-trips for connection establishment incur more latency than the additional latency Mobius itself adds.

7.3 Intelligent Server-directed Caching

In Section 6 we argued in favor of server-generated caching strategies. We now present some initial results that demonstrate the benefits that one can obtain from this approach—a simple initial solution based on decision tree classifiers. Exploring more sophisticated and robust machine-learning and statistical techniques is part of our research agenda.

For these experiments, we use data traces from the CarTel [36] iPhone application. The traces consist of vehicles location reports and geo-spatial queries (in this case, for traffic data in a region). Specifically, the data comprises

- *Writes*: GPS locations of about 600 cars, reported at regular intervals (approximately 3.7M data points).
- *Reads*: User-initiated queries for data points within a bounding box (20x20km) and a two hour window.

We first report experiments on read caching strategies. We compare: 1) a *baseline* based on the actual policy implemented in the CarTel application, which answers queries from the local store (cache) whenever the query is no further than 1km from a previously cached query not older than 10 min, and 2) a *decision-tree-based* policy, which analyzes the execution trace and devises a policy based on the current query and cache contents.

As input, the decision tree takes the timestamp and GPS coordinates of the current and previous query. We generate the decision tree using Weka's cost-sensitive J48 implementation. The classifier is trained to predict whether a query should be answered from the local cache or sent to the server. In certain geographical regions our data is very sparse, and the classifier learns that issuing a query

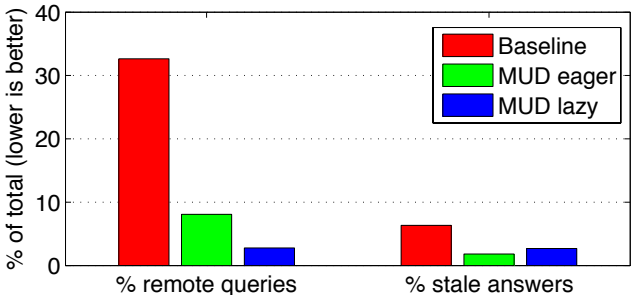

Figure 3: Mobius server-determined read caching vs Baseline on CarTel dataset

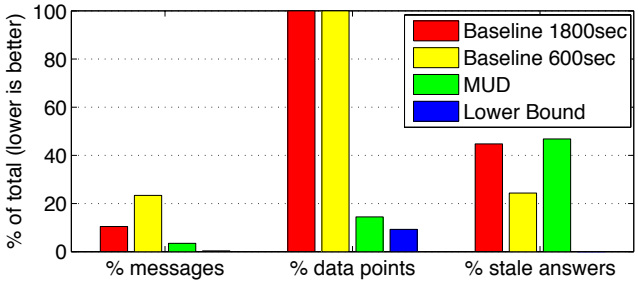

Figure 4: Mobius server-determined write batching vs Baseline on CarTel dataset

to the server if the client is interested in sparse areas is inconsequential even if the local cache is empty. We use standard 10-fold cross-validation in this experiment.

Figure 3 shows the results comparing the baseline described above and the decision tree classifier, for two different utility functions provided by the developer (one more sensitive to latency of the answer, and the other more sensitive to freshness of the data). In both cases, the decision tree significantly reduces the number of remote queries, improving perceived latency, bandwidth usage, power consumption and sever load, as well as result quality—fewer queries return stale data to the end user.

The second experiment focuses on write caching (or write batching). We compare two baseline policies that delay writes up to 10min and 30min respectively, our decision-tree based approach and a lower-bound computed *a posteriori* on the dataset.

We train the decision tree classifier to predict the maximum delay we can accept for a write without impacting a query. The classifier receives as input the GPS location (content of the write) and the time at which the write was performed in the trace. During training, we also include the time of the first subsequent read of the modified data. We quantize the delay into 10min buckets from 0 to 120min, and include an "infinite delay" bucket that accounts for unobserved writes. During training, we heavily penalize overestimates of maximum delay (which cause queries to receive stale data), and lightly penalize underestimates, increasing the cost progressively with increased error. For this experiment, we sample 10% of the data for training and use 90% for testing.

Figure 4 compares our approach with the baseline according to three metrics: the percentage of messages sent, the percentage of data points sent (this is to demonstrate the opportunity to not send data if we are confident the data will never be read), and the percentage of queries impacted by at least one missing data point. We show that our system's query results are comparable to the 30min fixed heuristic in exchange for a 7× reduction in the number data points sent (this translates in bandwidth savings and server side load reduction), and a 3× reduction in number of messages (which mostly translates into battery power reduction). When compared to a more aggressive 10min baseline we produced more stale results but up to 6× fewer messages.

While the presented approach is not completely general, it is also a somewhat naive use of classification. Even so, it is a significant improvement over reasonable baselines, showing that cloud-based caching policies can be very effective. In the context of the CarTel application, the policy selector determined that in certain regions, the density of writes is so low that pushing queries to the server would return no new results. Thus, based on location and time of the day the system is able to suggest that the client be more or less eager at pushing queries to the server.

7.4 Backend Scalability

For this experiment, we use the backend server to persist messages to disk and forward them to a single subscriber. The goal is to show impact of partitioning for a one-to-one messaging application. To stress test predicate evaluation, we group subscriptions into partitions, forcing the query processor to evaluate multiple subscription predicates for each incoming message. We focus on the gateway and query processor performance here, notification server scalability is well-studied, e.g., Google Thialfi [18].

The query processor processes subscriptions as it would in a real deployment: it decides correct recipient(s) for each message, and forwards the message to the notification server(s) for delivery. Each time a new tuple arrives, it is evaluated against each active subscription in its partition. Scenarios with only a few subscriptions per partition perform a small number of such evaluations per published tuple, while scenarios with many subscriptions per partition must perform many predicate evaluations for each published tuple.

The results of our experiment are shown in Figure 5. In the ideal case, each subscription is assigned to a separate partition and the query processor forwards approximately 5,000 messages per second. The cost of forwarding messages increases linearly as the number of subscriptions per partition increases. In our simple experiment, each message will be delivered to one subscriber; the work performed by the other predicate evaluations is wasted. Increasing the number of predicate evaluations while holding the other costs constant gives us a good idea of the performance of predicate evaluation: our simple prototype can evaluate approximately 70,000 predicates per second.

The query evaluation strategy performs well when most predicate evaluations succeed, but is unable to efficiently "prune" irrelevant queries as it processes incoming messages. We could avoid this linear slowdown in the number of subscriptions by implementing an index over query predicates, as in database continuous query processing systems like NiagaraCQ [28] or the ARIEL trigger processing system [29]. However, we observe that our architecture presents an interesting tradeoff to application developers: In order to improve performance, one can increase the sophistication of the predicate evaluation system, or one can resort to increasingly sophisticated partitioning schemes, reducing the number of subscriptions that must be considered when a tuple arrives. On the one hand, static partitioning schemes are unwieldy, but can be trivially parallelized and have good (and predictable) worst case performance. On the other hand, sophisticated query processing schemes are complex, but automatically improve the performance of a wide range of applications. We suspect that both techniques will be useful in practice.

Regardless of the underlying query processing scheme, subscription patterns that cause certain messages to be broadcast to large

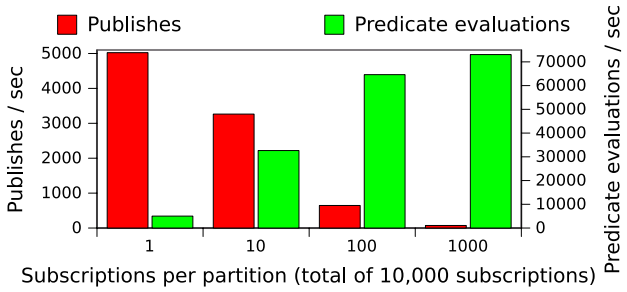

Figure 5: Impact of partitioning on one-to-one messaging via notifications. With small partitions, network and I/O overheads dominate. With large partitions, our stress test becomes CPU bound due to unnecessary predicate evaluations.

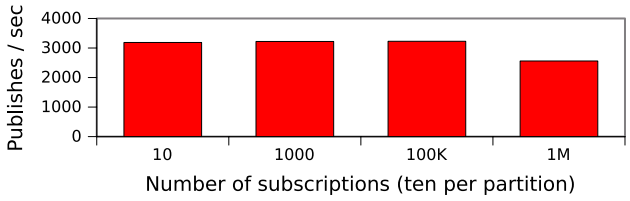

Figure 6: Notification throughput as subscriber count varies

numbers of users are likely to lead to hotspots, overwhelming some of the backend machines. Two common solutions to this problem are replication of partitions and read caching. Both approaches have the same effect: the computational and network load associated with hot partitions is spread across multiple backend servers.

Our simple prototype ignores such issues, and focuses on scaling up to applications with large numbers of small subscription groups. To this end, we ran the same publish / subscribe benchmark as above, holding the number of subscriptions per partition constant (Figure 6). With ten subscriptions per partition, the prototype is able to establish approximately one million subscriptions before running out of RAM to store active subscriptions. Although we suspect that other performance issues will manifest before we hit this limitation in practice, it is worth noting that we could spill the query cache to the backing bLSM index.

This design scales linearly by partitioning data and subscriptions across multiple servers. Partitioning schemes can be provided by application developers, but we also plan to further explore these issues by adapting our previous work on automatic and fine-grained partitioning [31, 46] to this context.

8. RELATED WORK

Mobius and MUD build on prior work from many fields, including mobile systems, pub-sub systems, messaging, continuous query processing and scalable "key-value" storage, but differ from existing work in several ways. First, the MUD API unifies messaging with data serving through the abstraction of a logical table that spans devices and clouds. Another novel aspect of MUD is that it introduces "writer predicates" to control writes, in addition to "reader predicates", in contrast to conventional pub-sub systems and database continuous queries. Second, Mobius allows utility functions to decide caching/prefetching decisions whose policies are derived by mining access patterns across clients at backend servers. Finally, Mobius's design integrates multiple components in a non-trivial way to provide app developers with a unified messaging and data platform, and also provides cloud operators with linear scaling in the number of applications, users and in the amount of data.

Existing cloud services are exposed to mobile apps through REST APIs or wrapper libraries of the APIs. Amazon Mobile SDKs for Android and iPhone [1] provide a wrapper library over the REST APIs of Amazon cloud services such as storage (S3), database (SimpleDB), and simple notifications service (SNS). SimpleDB is a non-relational data store that provides eventual or strong consistency, and SNS is a topic-based pub/sub system. SQL Azure database [13] and Google CloudSQL [3] provide cloud RDBMSs. In contrast, Mobius provides an integrated messaging and data serving platform that addresses challenges of mobile apps such as disconnection, mobility and intelligent caching for mobile devices.

Much research has been done on replication protocols for mobile computing [47]. The protocols can be categorized based on communication topologies (client-server vs. peer-to-peer), partial or full replication, and consistency guarantees. Coda and iCloud are example systems with client-server replication protocols; clients cannot directly share updates each other. Coda [37] is a distributed file system with which clients can perform disconnected operations. Coda caches a set of files specified by a hoarding profile. iCloud [5] provides synchronization of files among a set of apple devices (iPhone and iPad) and clouds, which supports a narrow app interface and semantics. Several projects have explored peer-to-peer replication protocols. Ficus [34] is a peer-to-peer replicated file system. Bayou [49] is a fully replicated database system that provides eventual consistency with session guarantees for choice of consistency [48]. PRACTI [22] supports topology independence, partial replication, and arbitrary consistency. To support partial replication, it separates invalidation messages from body messages with updates. CouchDB [2] is a schema-free "document store" with eventual consistency guarantees. Cimbiosys [41] and Perspective [42] use filters to selectively replicate content at mobile devices. Cimbiosys supports eventual filter consistency. With regard to data replication, Mobius has a client-server model since our main applications consume data from cloud, write data to cloud, and share data through cloud. Mobius provides partial replication since its cache contains partial records of tables. By avoiding direct peer-to-peer communications between devices, Mobius provides stronger consistency guarantees, and requires a simpler architecture. More importantly, it inherits all of the advantages of existing cloud storage architectures, such as analytics, multi-tenancy, elasticity, fault-tolerance, scalability, automatic application upgrades (to avoid legacy support issues), and so on.

Cloud to device push notification services such as Apple APNS [20], Google C2DM[4], Microsoft MPNS [11] provide a network channel to send small messages from cloud to device by maintaining a persistent connection between device and cloud. Similarly, our notification client/server provides channels for push notification. Thialfi [18] is a recent notification service that reliably propagates object version changes from cloud to client. In contrast, Mobius provides a high-level abstraction for data management and messaging. Unlike Mobius, Thialfi, does not deal with client-side caching, disconnected operations, consistency semantics, and so on. In addition, Mobius supports writer predicates which allow late binding of subscribers or readers to published content.

Communication service abstractions for mobile apps have also been explored. Project Hawaii [10] provides a collection of services and libraries, including a message relay service and a rendezvous service. Contrail [44] is a communication platform for decentralized social networks that uses the cloud to relay encrypted data. Junction [7] provides a switchboard service with which devices talk

each other, especially in peer-to-peer settings. These services are low-level communication services. Mobius provides higher-level abstraction for messaging and data management.

Various systems provide asynchronous, distributed write/read primitives. Pub/sub systems provide communication between publishers which produce information and subscribers which are interested in receiving it. Depending on the expressive power of subscriptions, pub/sub systems are classified into topic-based or content-based ones [27, 32]. Continuous query databases such as NiagaraCQ [28] and YFilter [33] evaluate queries when data is written to databases. Tuplespaces such as Linda [26] and TSpace [15] provide distributed associative memory spaces; applications write tuples to the tuple space and read tuples from the tuple space based on a select predicate. Mobius's write/continuous read is built on the same principle, but Mobius has both reader and writer predicates.

Caching/prefetching has been studied extensively in distributed systems. We briefly describe a few relevant mobile computing projects. SEER [38] provides automated, dynamic hoarding decisions for distributed file systems that support disconnected operations. The system observes file access patterns, clusters files with small semantic distances, and colocates them; this pattern mining occurs in each client machine. WhereStore [45] is a device store that uses filtered replication to create partial replicas based on device location histories. Mobius provides a generalization: arbitrary caching/prefetching policies based on intelligent *backend* policy inference from *all* clients. Cedar [50] uses content addressable storage to reduce data transmission by detecting commonality between client and server query results of relational databases. Query containment checking [19] extracts matching queries with ranges of exiting tuples. Both approaches rewrite queries to exploit locally available data. Mobius employs similar rewriting techniques at the device. ICEDB [52] uses global feedback from the server in deciding what tuples to upload. Mobius covers a wide variety of push/pull primitives between device and server based on global and local policies.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we introduced MUD, a unified messaging and data serving abstraction for mobile apps, and presented a system, Mobius, that supports it. Mobius provides several key features: automated caching, writer-/reader-predicate based reads and notifications, server-directed caching policies, and protocols for handling disconnected operation. In addition to simplifying mobile application development, we believe that Mobius will ultimately make more efficient use of hardware resources than is possible with ad-hoc application storage and messaging stacks.

Early experiments with our prototype implementation of Mobius are promising. Our MUD processing engine scales up to approximately one million subscriptions per machine, and is designed to linearly scale out to multiple back-end servers. Similarly, our client-side caching policies significantly reduce network bandwidth use. Finally, the overhead we impose upon client requests is reasonable. In conclusion, we believe that the MUD API, combined with our network and caching technologies, will significantly simplify the lives of mobile application developers.

Acknowledgments

We would like to thank Tyson Condie for his comments and contributions. We also would like to thank the anonymous reviewers for their comments and our shepherd, Doug Terry, for his guidance.

10. REFERENCES

[1] AWS SDK. aws.amazon.com/mobile.
[2] CouchDB. couchdb.apache.org.
[3] Google Cloud SQL. code.google.com/apis/sql.
[4] Google cloud to device messaging. code.google.com/android/c2dm/index.html.
[5] iCloud. www.icloud.com.
[6] JBoss Netty. www.jboss.org/netty.
[7] Junction. openjunction.org.
[8] MVEL. mvel.codehaus.org.
[9] OAuth. oauth.net.
[10] Project Hawaii. http://research.microsoft.com/en-us/um/redmond/projects/hawaii/.
[11] Push notifications for windows phone. http://msdn.microsoft.com/en-us/library/ff402537(v=VS.92).aspx.
[12] SimpleDB. aws.amazon.com/articles/3572?_encoding=UTF8&jiveRedirect=1.
[13] SQL Azure Database. msdn.microsoft.com/en-us/library/windowsazure/ee336279.aspx.
[14] SQLite. www.sqlite.org.
[15] TSpace. www.almaden.ibm.com/cs/tspaces.
[16] Yahoo! Query Language. developer.yahoo.com/yql.
[17] ZooKeeper. zookeeper.apache.org.
[18] A. Adya, G. Cooper, D. Myers, and M. Piatek. Thialfi: A client notification service for internet-scale applications. In *SOSP*, 2011.
[19] K. Amiri, S. Park, R. Tewari, and S. Padmanabhan. Scalable template-based query containment checking for web semantic caches. In *ICDE*, 2003.
[20] Apple push notification service. In *iOS Developer Library*. 2011.
[21] N. Balasubramanian, A. Balasubramanian, and A. Venkataramani. Energy consumption in mobile phones: a measurement study and implications for network applications. IMC, 2009.
[22] N. Belaramani, M. Dahlin, L. Gao, A. Nayate, A. Venkataramani, P. Yalagandula, and J. Zheng. PRACTI replication. In *NSDI*, 2006.
[23] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.
[24] C. Cachin, A. Shelat, and A. Shraer. Efficient fork-linearizable access to untrusted shared memory. In *PODC*, 2007.
[25] B. Calder et al. Windows azure storage: A highly available cloud storage service with strong consistency. In *SOSP*, 2011.
[26] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 1989.
[27] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *TOCS*, 2001.
[28] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.
[29] E. H. Chris, C. Carnes, L. Huang, M. Konyala, L. Noronha, S. Parthasarathy, J. B. Park, and A. Vernon. Scalable trigger processing. In *ICDE*, 1999.

[30] B. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H.-A. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. In *VLDB*, 2008.

[31] C. Curino, Y. Zhang, E. P. C. Jones, and S. Madden. Schism: a workload-driven approach to database replication and partitioning. In *VLDB*, 2010.

[32] P. Eugster, P. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2), 2003.

[33] Y. D. P. Fischer, M. J. Franklin, and R. To. YFilter: Efficient and scalable filtering of XML documents. In *ICDE*, 2002.

[34] R. G. Guy, J. S. Heidemann, W.-K. Mak, T. W. P. Jr., G. J. Popek, and D. Rothmeier. Implementation of the ficus replicated file system. In *USENIX Summer*, 1990.

[35] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *TOPLAS*, 12(3), 1990.

[36] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Madden. CarTel: A distributed mobile sensor computing system. In *SenSys*, 2006.

[37] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *TOCS*, 1992.

[38] G. H. Kuenning and G. J. Popek. Automated hoarding for mobile computers. In *SOSP*, 1997.

[39] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Comput.*, 28, 1979.

[40] A. Oprea and M. K. Reiter. On consistency of encrypted files. In *DISC*, pages 254–268, 2006.

[41] V. Ramasubramanian, T. L. Rodeheffer, D. B. Terry, M. Walraed-Sullivan, T. Wobber, C. C. Marshall, and A. Vahdat. Cimbiosys: A platform for content-based partial replication. In *NSDI*, 2009.

[42] B. Salmon, S. W. Schlosser, L. F. Cranor, and G. R. Ganger. Perspective: Semantic data management for the home. In *USENIX FAST*, 2009.

[43] R. Sears and R. Ramakrishnan. bLSM: A general purpose log structured merge tree. In *SIGMOD*, 2012.

[44] P. Studi, I. Mohomed, M. Balakrishnan, T. Wobber, D. Terry, and M. Mao. Contrail: Enabling decentralized social networks on smartphones. In *Middleware*, 2011.

[45] P. Studi, I. Mohomed, and D. Terry. WhereStore: Location-based data storage for mobile devices interacting with the cloud. In *MCS*, 2010.

[46] A. Tatarowicz, C. Curino, E. P. C. Jones, and S. Madden. Lookup tables: Fine-grained partitioning for distributed databases. In *ICDE*, 2012.

[47] D. B. Terry. *Replication Data Management for Mobile Computing*. Morgan-Claypool, 2008.

[48] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch. Session guarantees for weakly consistent replicated data. In *PDIS*, 1994.

[49] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *SOSP*, 1995.

[50] N. Tolia, M. Satyanarayanan, and A. Wolbach. Improving mobile database access over wide-area networks without degrading consistency. In *MobiSys*, 2007.

[51] H. Yu and A. Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *OSDI*, 2000.

[52] Y. Zhang, B. Hull, H. Balakrishnan, and S. Madden. ICEDB: Intermittently-connected continuous query processing. In *ICDE*, 2007.

APPENDIX

A. FORMAL CONSISTENCY SEMANTICS

We describe the consistency semantics that Mobius provides to clients accessing its tables formally. As operations take time, they are represented by two events occurring at the client, an *invocation* and a *response*. A *history* is the sequence of invocations and responses that occurred in an execution. An operation o *precedes* another operation o' in the history if o completes before o' is invoked. Two operations are *concurrent* if neither one of them precedes the other. Finally, we say that a sequence of events is *sequential* if it does not contain concurrent operations.

Sequential views: We define consistency in terms of the possible *views* clients have of the execution (as previously mentioned, each record is treated as a separate consistency domain). Intuitively, a view is a sequence of invocations and responses which is sequential and equivalent (from the client's point of view) to the execution history. The view must contain the client's own operations in addition to any operation apparent from the client's interaction with the shared storage. The view must be *legal* according to the "sequential specification" of the shared object (i.e., it could have resulted from a sequential interaction with the storage). Finally, the view must respect the operation precedence order of every client's program: for every two operations executed by the same client o and o' in a view, if o completes before o' is invoked then it appears before o' in the view. Note that there are usually multiple views possible at a client.

Sequential Consistency: The first consistency property provided by the system is *sequential consistency* [39]. Here, we formalize the definition using the notion of views:

DEFINITION 1 (SEQUENTIAL CONSISTENCY). *A history σ is* sequentially consistent *if there exists a sequence of events π which is a view of σ for all clients.*

Sequential consistency with write invocation order: We say that a client's view *preserves write invocation order* of a history σ if for every two operations o and o' of the same client in the view, if o is invoked before o' in σ and at least one of them is a write, then o precedes o' in the view. We call the resulting consistency condition *sequential consistency with WIO* (write invocation order):

DEFINITION 2 (SEQUENTIAL CONSISTENCY WITH WIO). *A history σ is* sequentially consistent with WIO *if there exists a sequence of events π which is a view of σ for all clients, and which preserves the invocation order of writes in σ.*

Fork sequential consistency: We formalize the possibility of view divergence using the notion of *fork-sequential consistency* [40].

DEFINITION 3 (FORK-SEQUENTIAL CONSISTENCY). *A history σ is* fork-sequentially consistent *if for each client C_i there exists a sequence of events π_i such that (1) π_i is a view of σ at C_i; and (2) For every client C_j and every operation o in both π_i and π_j, the prefix of π_i up to o is identical to the prefix of π_j up to o (called the* No-join *property).*