CAL: A Framework for Managing Consistency, Availability, and Battery Lifetime in Mobile Applications

ABSTRACT

We propose a framework that allows mobile applications to continously choose between consistency, availability, and battery lifetime as network bandwidth and availability change and battery levels drain. Our system, CAL, provides various mechanisms and interfaces so applications can adjust their performance along these three axes in the tradeoff space as operating conditions change. We analyze a month's worth of mobile and WIFI network-coverage data and characterize the variability of the underlying network and discuss the implications on mobile application design. We describe how CAL eases application design in this context and demonstrate it in an indoor, energy-auditing application – an application for collecting information and querying the physical environment for live energy consumption feeds in buildings. We introduce a new prefetching technique, aimed to optimize fetch size and object set according physical proximity of the objects as inferred by object reference times. We show that CAL helps provide higher consistency and availability at lower energy cost, and degrades gracesfully as resources become exhausted.

1. INTRODUCTION

A new class of mobile applications is emerging on mobile phones. The new fontier for enabling ubiquitous connectivity is moving indoors. Applications are becoming more reliant on ubiquitous, continuous connectivity, however there are fundamental challenges to enable an uninterrupted user experience in an indoor setting.

A variant of participatory sensing, whereby we're not just collecting information about the environment, but we expect the participants to be the authority in the construction of the environment.

This work will have similarities to Informed Mobile Prefetching (IMP, mobisys2012).

2. RELATED WORK

3. INDOOR NETWORK ACCESS PROFILE

3.1 Methodology

3.2 General connectivity and access

3.3 Building 1

3.4 Building 2

3.5 Building 3

3.6 Summary

4. SYSTEM DESIGN

Our system is designed to provide a set of basic services and an API that application designers can use to both reason about and implement strategies for dealing with 1) consistency 2) availability 3) and battery lifetime. We provide an *simple API* that gives the application a way to make local decisions based on the age of application-level objects, a *Prefetcher* that prefetches data at an adjustable rate, and a *cache* that is used to deal with both reduced latency and increased availability. Much of the library consists of a set of interfaces, that encapsulate the notion of an application-level object, the application server, the operations that can be performed on those objects, and an *Expression*, which consists of a set of operations, executed in a given order, atomically.

To deal with disconnected operations, we provide a *OpLog*, which logs the set of operation that were performed locally to application objects, that should eventually be performed on the server. Many of our API calls allow the user to submit a callback object, which is triggered when the certain operation is performed on the object on the server. This two-phase approach is necessary to keep the application responsive in times where access to the net-

work is down. We also allow give the user hooks into the synchronization process, which monitors the consistency between the local object copies and those on the server.

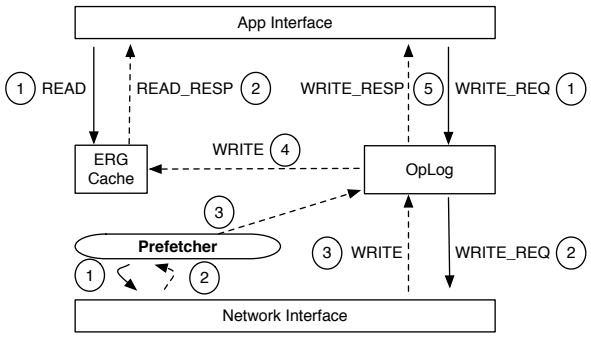

Figure 1: Standard mechanisms for consistency management on the phone. All READ request go to the local cached version of the ERG. All WRITES must go through the OpLog.

Figure 1 shows a high-level overview of the components in the architecture and how they interact. We discuss these components in the following secions. We start with the library that runs on the client and the optional library to run on the server. We also discuss the benefits and implications of using either just the client or both the client and server stubs, particularly with respect to the consistency model.

5. CLIENT STUB

5.1 API design rationale

Different mobile applications have different consistency, energy, and responsiveness requirements. Unlike traditional application, mobile application need to be able to continuous adjust their approach to fall on different points in the trade-off space along each of these axes, throughout the lifetime of the application. Our goal is to design an API where the application programmer can reason about and implement applications that dynamically adjust their position along these three tradeoff axes as the runtime conditions change.

5.1.1 Consistency

Eventually consistency is the consistency model that can be reasonably be implemented in our system. Variable bandwidth and disconnection, as well as varible energy consumption policies constraint the consistency model to best effort, eventually consistent. All clients eventually see exactly the same sequence of changes of the values for each object. We draw an important semantic distinction between the use of the just the client stub and the client-server combination. Using only the client stub gives speculative eventual consistency. Operations are performed, proactively, on local copies of the object. At some point in the future, those updates will be applied to the server. If the server cannot execute the given operations, the server object is copied on the server and the application is notified.

Operations are an important components of the approach we take. We force the application designer to define the set of operations that can be performed by the application server, so that they may also be applied on local copies of objects. For some applications, sets of operations must be performed atomically, or not at all. We define such a collection as an *Expression*. Expressions are only supported if the server stub is implemented, since only the server can execute operational sets atomically and only the server can resolve conflicts between clients.

5.1.2 Availability and responsiveness

Network connection availability and quality change through space and time. Most applications need to make in-time decisions about which data and how much data they are going to fetch for the application. For indoor, interactive application, the network conditions can vary substantially in different parts of the indoor environment. Our api must support a caching layer for responsiveness and proactive preftching for availability. We aim to allow the user to vary the size and frequency of the prefetch mechanism, thereby allowing for dynamic adjustment of application performance and availability as conditions dictate.

5.1.3 Battery lifetime

We can attain high levels of consistency at a higher average cost of energy, by making our all writes go directly to the application server. For each read-/write of an application object, there's an associated, time-varying, cost. Our API should allow applications to make the appropriate choice, given the current opertional conditions. For example, we can achieve high levels of consistency if all reads/writes are write-through to the server. At the other end of the spectrum, we do all reads/writes from locally. In the latter, we run the risk of dealing with stale objects.

Table 1 shows the basic API that is made available to the runtime of the client application. We

API call	Description
read(objectName, save)	Reads the object from the server if a connection is available. Otherwise, reads from the local cache.
read(objectName, freshness,save)	Reads the object from local cache if it's $\leq$ freshness time units old. Otherwise, reads from server.
read(objectName, callback, save)	Reads from the server when the server is available. Callback is triggered after the fetch is complete.
readEOP(objectName, save)	Reads the object from the server if a connection is available and its within the energy budget. Otherwise, reads from the local cache.
readEOP(objectName, freshness, save)	Reads the object from local cache if it's $\leq$ freshness time units old. Otherwise, reads from server.
readEOP(objectName, callback, save)	Reads from the server when the server is available. Callback is triggered after the fetch is complete.
write(objectName, data, op)	Sends a request to the server to apply the operation remotely and copies the object locally after the operation complete. If the server is unavailable, applies the operation locally and logs it.
write(objectName, data, op, freshness)	Writes the operation to local copy if it is $\leq$ freshness time units. Otherwise tries to write to the server.
write(objectName, data, op, callback)	Writes to the server. Callback is triggered when the write is complete. A local copy is cached.
writeEOP(objectName, data, op)	Sends a request to the server to apply the operation remotely and copies the object locally after the operation complete. If the server is unavailable, applies the operation locally and logs it.
writeEOP(objectName, data, op, freshness)	Writes the operation to local copy if it is $\leq$ freshness time units. Otherwise tries to write to the server.
writeEOP(objectName, data, op, callback)	Writes to the server. Callback is triggered when the write is complete. A local copy is cached.

Table 1: Summary of the main API calls of the Context Object Layer. Each call allows the designer to reason about and implement along different points in the tradeoff space between consistency, responsiveness/availability, and energy consumption.

essentially support three types of calls, each with three different sets of parameter and their own semantics.

6. FETCH COST

To determine the energy cost of a fetch, we need to calculate the time it takes to fetch the object.

We have variable connection quality and variable object size.

1. Maintain a weighted average of the size of the object to fetch.
2. Maintain a weighted average of the connection speed.
3. Maintain availability of each network based on the number distribution of object transfers over particular networks.

6.1 Server stub

7. APPLICATION EXAMPLE: ENERGY LENS

7.1 Access pattern and available

7.2 Uptime distribution

8. RELATED WORK

Bayou [3] is a system architecture that is designed to provide high-availability, and variable consistency while supporting application-specifc conflict resolution and application-controlled inconsistency management. The system was specifically designed for mobile clients with variable connection quality and periodic disconnection. In many ways, the design goals of Bayou are very similar to ours. We both aim to provide highly available read and write operations, we both rely on an eventual consistency model, both support application-level detection of conflicts, application-specific conflict resolution, permitting disconnected clients to see their own updates. In contrast, we do not explicitly provide session guarantees. We assume all clients are writing to the same application servers and that updates to the server by any client is consistency across a cluster of application servers.

Our system, however, introduces the notion of variable consistency and availability with respect to energy consumption. Typically applications are willing to spend more energy per I/O operation if connectivity is available and the cost/bit is cheap. Moreover, we include a prefetcher in our framework that is adjusts it prefetch download size based on time-based hierarchical clustering of object reference time, dictated largely by the proximity of physical objects to the user interacting with the application. We focus on mobile applications that are interacting with objects in the physical world, not necessarily a virtual object on a server being edited by a random set of clients. Most reads on the virtual object will come from clients near that object in physical space, thereby limiting the number of concurrent interactions and the prefetch size to the space physical space occuppied by those client's primary users. Bayou is a database system for mobile clients, we are mainly client-side framework with optional server-side support to stronger consistency and ordering guarantees.

PocketWeb [2] is an prefetch framework for mobile phones that prefetching dynamic web content. PocketWeb uses a machine learning approach based on stochastic gradient boosting techniques to model the mobile web browsing patterns of mobile users. The main observation that allows their system to effectively prefetch content is that there are user-specific spatio-temporal access patterns. Their technique builds a model per user and prefetches 80%-90% of the content for 60% of users 2 minutes before they access it. Proactive, periodic prefetching is necessary dynamic web content, in particular. By limiting our framework to the class of indoor, interactive mobile application – where the user is directly interacting with the space around her through her phone – we limit the scope and access pattern to the spatial proximity to the world around them. With the increase in embedded sensing, we will see more applications that fall into this class, and prefetching as many virtual objects near the user as possible, will become more important. We also allow the application designer to control the frequency and scope of the prefetch. Fetching larger sets more frequency when connectivity is good and energy level is high.

Informed Mobile Prefetch (IMP) [1] using an adaptive strategy for prefetcing user-specified application-level objects. IMP notes the objects that are referred to most often and treats the connection and energy-level as a resource to be budgeted depending on the anticipated demand for those object and the projected energy and bandwidth supply. **We use the same prefetch decision criteria, except that we consider the cost benefit of fetching on a collection of items that considers the level in the hierarchical cluster tree, rather than calculate it on a per-item basis.**

9. REFERENCES

[1] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson. Informed mobile prefetching. In *Proceedings of the 10th international conference on Mobile systems, applications, and services*, MobiSys '12, pages 155–168, New York, NY, USA, 2012. ACM.

[2] D. Lymberopoulos, O. Riva, K. Strauss, A. Mittal, and A. Ntoulas. Pocketweb: instant web browsing for mobile devices. In *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, pages 1–12, New York, NY, USA, 2012. ACM.

[3] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser. Managing update conflicts in bayou, a weakly connected replicated storage system. In *In Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, pages 172–183, 1995.