Cloud-Based, User-Centric Mobile Application Optimization

John Kolb, Prashant Chaudhary, Alexander Schillinger, Abhishek Chandra, Jon Weissman
Department of Computer Science & Engineering
University of Minnesota, Minneapolis, Minnesota 55455
Email: {kolb, prashant, schil399, chandra, jon}@cs.umn.edu

***Abstract*—The abundance of compute and storage resources available in the cloud makes it well-suited to addressing the limitations of mobile devices. We explore the use of cloud infrastructure to optimize content-centric mobile applications, which can have high communication and storage requirements, based on the analysis of user activity. We present two specific optimizations, precaching and prefetching, as well as the design and implementation of a middleware framework that allows mobile application developers to easily utilize these techniques. Our framework is fully generalizable to any content-centric mobile application, a large and growing class of Internet applications. A news aggregation application is used as a case study to evaluate our implementation. We make use of a cosine similarity scheme to identify users with similar interests, which in turn is used to determine what content to prefetch. Various cache algorithms, implemented for our framework, are also considered. A workload trace and simulation are used to measure the performance of the application and framework. We observe a dramatic improvement in application performance due to use of our framework with a reasonable amount of overhead. Our system also significantly outperforms a baseline implementation that performs the same optimizations without taking user activity into account.**

Keywords*—*cloud computing; middleware; mobile computing; optimization;

I. Introduction

Mobile devices have become increasingly prevalent in recent years. It is estimated that more than half of all adults in the United States own a smartphone [25]. The convenience and portability that these devices offer has made them a primary means of interacting with the virtual world, whether to communicate with others or to access and produce content. Moreover, the growth in ownership of these devices has been accompanied by an explosion of mobile applications and software specifically designed to run on phones and tablets. Mobile devices are becoming increasingly versatile, and many tasks that once required a desktop or notebook computer can today be carried out entirely on mobile systems.

However, mobile devices are not without drawbacks. First, the processing power of these devices is necessarily less than that of traditional computers. This is not only because of the small size of mobile devices, but also because of the lack of cooling for hardware components. While multi-core mobile CPUs are becoming more common, they still cannot match the processing power of today's notebooks, desktops, and servers. Additionally, mobile devices are inherently limited by their reliance on battery power. This makes energy a precious resource that must be used wisely in order to extend battery life. Finally, mobile phones often must operate over cellular networks, which are slower and less reliable than traditional computer networks. This is further complicated by the fact that network communication is a particularly expensive operation in terms of energy and that many mobile users have carrier-imposed limits on data traffic.

Although mobile systems invest significant effort to mitigate these problems, most solutions require significant familiarity with the platform and hardware upon which the applications will be deployed. The cloud provides great opportunities as it has a large number of resources and can support multiple users through the elasticity and scalability of its resources. Hence, it can be used for mitigating the problems seen above by offloading the processing of content on the cloud. To exploit the capabilities of cloud we have developed a cloud/mobile device-based middleware framework that is able to optimize content-centric mobile applications with minimal effort required from their developers.

Specifically, we focus on mobile applications that access remotely-stored content on behalf of their users. These include a wide variety of popular applications including news readers, video and image viewing applications, music players, as well as other domain-specific applications that fetch content to user devices. Our system is currently designed only for applications in which users are consumers of content and do not produce or share any content of their own. Remote resource retrieval is typically an expensive proposition, as it involves both computation and significant network communication. Resource retrieval can also incur a substantial cost in terms of latency, especially if additional content processing is necessary before it can be delivered. This often results in a noticeable delay that disrupts the user's experience. The middleware employs two principal strategies, based on user profiling, to reduce these costs.

1) **Precaching**: When a resource is retrieved and processed, the system caches the processed version in the cloud. Future requests for this resource can be satisfied by a cache lookup rather than a direct retrieval that may also involve intermediate processing.
2) **Prefetching**: The system speculatively pushes a resource to a user's device, where it is cached in hopes of a future request for that resource. If such a request occurs, it can be satisfied without any network communication.

An important aspect of the system is its mechanism for choosing when to prefetch specific resources for a specific user. Prefetching too few resources means the user will see little to no visible benefit from optimizations, while prefetching too many resources results in the unnecessary consumption of network bandwidth and energy. In order to make intelligent prefetching decisions, we attempt to model and identify a user's *region of interest* – the subset of resources that they are particularly likely to access. This requires the system to record user access patterns and to analyze them through user profiling and data mining techniques.

We have implemented an Android/Amazon EC2-based middleware system and evaluated it using a real-world News Aggregator application. A trace-driven emulation using a workload trace derived from Twitter/Disqus feeds is used to evaluate the performance of the application and to analyze the effects of optimizations as carried out by the middleware. Our results indicate that the middleware is able to significantly reduce access latency with negligible delay introduced by its data processing.

II. Background & Related Work

A. Mobile Offloading

Much work has been done in efforts to offload computationally expensive operations from mobile devices to back-end servers, possibly running in the cloud [22]. This technique can generally be viewed as a form of remote procedure call in which the client is a mobile device. Many systems [8], [21], dynamically choose to run code locally or at a remote location depending on the projected cost of the operation, the potential overhead of offloading, and the current state of the mobile device. Other work has examined the potential for sharing offloading computations among multiple mobile devices [17]. These systems have demonstrated improvements in performance as well as a reduction in energy consumption for a variety of applications, such as image processing and games. The burden imposed on the mobile application developer by each of these systems varies. For example, [21] requires the use of a specific framework, [8] requires the use of code annotations, while [6] works through static analysis and requires no changes in code. The most recent work in mobile offloading [12] deals with offloading computation on cloudlets, though this work does not use any user profiling like we do.

B. Mobile Usage Patterns

As mobile devices have become increasingly ubiquitous, researchers have become interested in identifying the ways in which these devices are used. Multiple studies [5], [7], [10] have shown that smartphone usage exhibits both temporal and spatial patterns. That is, the way in which a person uses a mobile device is directly related to the time of day and that person's location. Moreover, significant work has already been done to investigate the use of data mining techniques to elucidate these kinds of patterns [19], [23], [24], although these techniques are not specifically targeted to mobile device usage data.

In our past work [16], we have demonstrated the use of *region of interest* abstraction to reduce the latency of data access with the use of cloud. In this paper we demonstrate the utilization of past work to build a scalable, configurable and generic framework which can used by developers to build content-centric cloud-based mobile applications. Our framework supports a default configuration which can be used by the developers out-of-the-box.

C. Recommender Systems

Recommender systems seek to identify the resources, such as movies or books, that would be most useful for a particular individual. This can be done through simple content-based approaches or through more sophisticated analysis techniques like collaborative filtering [9] and matrix factorization [11]. At a high level, the goal of a recommender system is to analyze user behavior, whether in the form of explicit item ratings or more implicit usage patterns, in order to identify that user's preferences and to suggest items that conform to those preferences. While both recommender systems and our middleware system aim to determine a user's interests, they have slightly different end goals. A recommender system will attempt to bring items to a user's attention that the user would normally overlook if left to his or her own devices. This objective is known as serendipity or novelty in the recommender systems literature [13]. In our setting, the user's consumption of a novel item is problematic because it precludes any possibility of anticipating and thus optimizing the retrieval of this content.

D. Prefetching

Prefetching is a well-studied practice in the web browsing domain. Most modern browsers support or engage in prefetching to reduce load times. Much research has also been done to improve prefetching techniques in a web context [19], [26]. These efforts typically seek to identify relationships between web pages, which can be in the form of explicit hyperlinks or common access patterns exhibited by users, that may be able to predict which web pages a user is most likely to navigate to next. The use of prefetching to improve the mobile user experience has also been studied extensively. The Informed Mobile Prefetching system [14], for example, makes prefetching decisions based on current network connectivity. Microsoft has also investigated prefetching on its mobile devices, both to predict which applications a user will open in the near future [20] and to prefetch advertisements [18]. Amazon has deployed the Silk browser [1] to offload expensive computations and to perform prefetching for its Kindle devices. Prefetching is also studied for various domains and applications like Spotify [27] and mobile advertisement deliveries [15]. But the prefetching techniques used in these systems are very specific to that particular domain and cannot be generalized to other domains. Here, we have developed a middleware that exploits user interests to drive optimizations in a framework that can be integrated with and customized for specific applications by mobile developers with minimal effort.

III. Overview of Middleware Framework

We have developed a cloud-based middleware for Android applications. Our middleware strives to allow mobile application developers to utilize the *region of interest* [16] abstraction and the optimizations it entails, namely precaching and prefetching. A region of interest is the subset of an application's features or content space that is most commonly utilized by a specific user and therefore of most relevance to that user. Several users may have the same region of interest or may have overlapping regions of interest. This concept can assume a number of different forms, depending on application domain. In an application primarily concerned with physical location, for example, a user's region of interest might correspond to a specific geographic area, while in an information retrieval application the region of interest might be a collection of topics or terms. This abstraction forms the core of our middleware.

We employ data mining techniques in an effort to identify each user's region of interest. Once this is known, we can attempt to predict which content a user is most likely to access in the future, allowing us to anticipate user actions and optimize for them.

Ideally, the system should achieve meaningful improvements in application performance even when a minimum of effort is put forth by the application's developer. Conversely, if the developer wishes to invest the time to customize the middleware's behavior to better match his or her own use case, then this should also be supported. This situation leads to several important goals for the design and implementation of the middleware, both at the user level and at the system level.

User-Centric Goals

- Present a small but flexible interface to the developer
- Allow for clean and concise code
- Require minimal changes when integrating middleware into an existing application
- Provide a sensible default configuration while also supporting customization

System-Centric Goals

- Use a modular design to support user customization
- Make intelligent use of cloud resources
- Incur minimal data processing overhead
- Achieve performance that is no worse than that of an unoptimized version of the application in any situation

Our middleware presents a key-value store as its principal interface to Android applications. Each key-value pair corresponds to an atomic item of content. The key is a unique identifier and handle for the content, such as a hash value or URL, while the value is the content itself. This could be text, an image, or even an application-specific data structure. The middleware supports keys and values of arbitrary types, thus allowing it to generalize to a wide variety of application domains. An application then performs a `get` operation in order to retrieve a resource. Note that the middleware achieves both location and access transparency. That is, an application has no knowledge of whether a resource has been precached or prefetched, nor does it need this knowledge in order to effectively use the middleware and benefit from its optimizations.

A `get` operation is carried out in one of three different ways, each of which appears identical to an Android application. These three scenarios are depicted in Figure 1. First, in the worst case, i.e. operations 1-6 in Figure 1, the application must retrieve the content from its cloud server, which in turn must synchronously fetch the content from the source and process it before responding to the client. This procedure can involve significant latency, as the user application blocks until the necessary data has been retrieved, processed, and delivered. In the second scenario, the application must again retrieve the content from the cloud, but fortunately the item is precached by the cloud-side server and thus can be immediately delivered to the mobile client. We therefore effectively cut out operations 3 and 4. The time required to complete the operation is therefore dominated by network round-trip time. Finally, in the ideal situation, the user requests content that was speculatively pushed to his or her device, in which case the relevant content is obtained without any network communications and without any perceivable delay — simply operations 1 and 6.

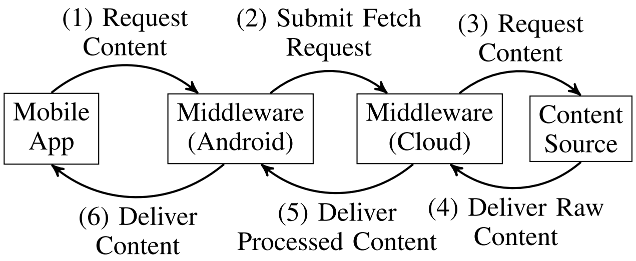


Figure 1: The Three Scenarios for Resource Retrieval

An application developer must specify how resources are retrieved from their sources, as the middleware has no way of inferring this. This is done through the implementation of a simple interface. Similarly, if a developer wishes to deploy a specific data-mining scheme in order to analyze and predict user behavior, they must also implement a particular interface that can then be freely plugged in to the rest of the middleware system. This can be particularly valuable if a developer seeks to analyze the content of user requests in addition to the requests themselves. However, some data-mining techniques may be content-neutral, meaning they can generalize to different domains. In that case, a user can easily configure the middleware to make use of implementations written by others in order to avoid the burden of implementing this functionality for themselves. The developer also has the freedom to implement a cache eviction policies specific to their application domain.

IV. Architecture & Implementation

Our system is made up of several components, each of which has a specific role and specific means of interaction with the other components. The `ResourceManager`, as its name suggests, manages content for an application. This includes requesting and caching content, as well as updating the `ResourceServer` of user activity. When requests cannot be satisfied locally, the `ResourceManager` communicates with the cloud-based `ResourceServer` on behalf of the mobile application. It is within the cloud where the `ResourceServer` satisfies requests that could not be made locally by either getting the requested content from its cache, or having the `ResourceFetcher` fetch it from its origin to be processed, cached, and sent to the user. The `ResourceServer` also works in sync with the `PredictionEngine`, feeding it content and user history, in order to forecast user requests, and speculatively push content to the mobile users. The complete system is diagrammed in Figure 2.

A. *ResourceManager*

The `ResourceManager` is resident on an Android device and is the means by which a developer's application makes use of the middleware system. It exposes a key-value store to

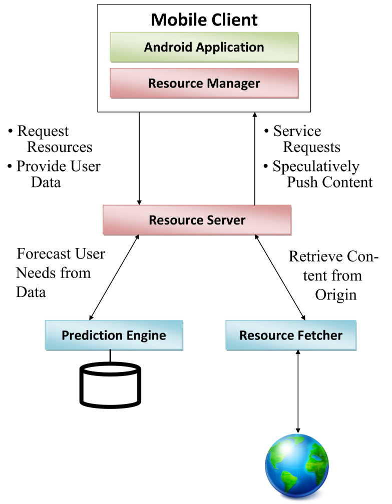


Figure 2: A High-Level View of the Middleware's Structure

the application that is used to transparently retrieve content. As such, the `ResourceManager` features a relatively simple interface. This interface is further simplified by the fact that a read-only view of the key-value store is sufficient, as the Android application will consume content but will not produce any content of its own. Thus, an Android application makes use of two major features of the `ResourceManager`:

1) `get(key)` retrieves the content corresponding to the provided key.
2) `getAvailableResources()` produces a collection of keys, each of which represents an item of content that is currently available for user consumption.

B. *ResourceServer*

The `ResourceServer` is deployed on cloud infrastructure and is responsible for servicing requests for content. A single `ResourceServer` is intended to handle the traffic generated by a large number of `ResourceManager` instances acting on behalf of Android applications. The `ResourceServer` is expected both to retrieve content directly from its source using an implementation of `ResourceFetcher` and to manage this content on behalf of mobile application clients. This component is also responsible for carrying out optimizations based on input from the `PredictionEngine`. Therefore, the `ResourceServer` may initiate retrieval and processing of content either as a direct response to a user request or as a precaching measure. The server will also speculatively distribute content to mobile clients as a prefetching measure.

Client requests processed by the `ResourceServer` can be broken down into four different types, each of which is handled differently:

1) **Fetch**: Retrieve the content item corresponding to a supplied key.
2) **Survey**: Retrieve a list containing keys for all resources currently available.
3) **Prefetch**: Retrieve content to be cached on a mobile device for future use.
4) **Update**: Alert the server about user requests that were satisfied by prefetched content.

The update operation merits further discussion. The operation is necessary in order to maintain consistency between a mobile client and the cloud-based server. The best-case scenario in terms of performance and user experience occurs when the user requests content that has been prefetched, as that request can be satisfied immediately and without any network communication. Thus, when the middleware is doing its job well, there will be user requests that ordinarily never reach its cloud components. This is a problem, as these requests are precisely the information that must be collected and analyzed if intelligent optimizations are to be carried out. To address this problem, the `ResourceManager` will periodically contact the `ResourceServer` in order to synchronize user request history. This allows the mobile client to benefit from prefetching while also keeping the server informed of user events for data mining purposes.

This situation leads to an interesting trade-off between consistency and bandwidth/energy consumption. We see this trade-off in two situations, each mirroring the other. First, we must decide how frequently to synchronize user request history between mobile application instances and the server. Second, we must decide how frequently to ping mobile applications in order to initiate prefetching of content. If either of these operations are performed too frequently, unreasonable amounts of energy and bandwidth may be consumed. If either of these operations aren't performed frequently enough, server-side user data becomes too stale and mobile clients end up retrieving content from the cloud that could have been prefetched. Both time-based variables are easily adjusted in our middleware to achieve a desired balance

C. *ResourceFetcher*

This component is responsible for all interaction with content sources. Its interface consists of two operations, and it is essentially equivalent to the interface exposed to Android applications by the `ResourceManager`. These two operations are:

1) `fetchFromOrigin(key)`: given a key, retrieves the corresponding content to be processed from its source.
2) `getAvailableResources()`: produces a collection of keys, each of which represents an item of content that is currently available for user consumption.

However, there are several key differences between the `ResourceFetcher` and the `ResourceManager`. While the `get` operation of the `ResourceManager` may transparently involve the utilization of precaching or prefetching optimizations, the `ResourceFetcher` always retrieves content directly from the source. Similarly, when a `ResourceManager` requests a list of available resources from a `ResourceServer`, it may receive a cached version in response. A `ResourceFetcher`, however, always computes a fresh view of the available resources.

Note that an application developer is responsible for the implementation of the `ResourceFetcher`. He or she must specify how to retrieve a specific item of content from its source as well as how to create a collection of identifying keys for the resources that are currently available. While this does require some effort, the middleware has no way of inferring how to accomplish these tasks, as it generalizes across various content domains. Furthermore, as its interface consists of just two operations, the burden of implementing the `ResourceFetcher` is as minimal as possible.

D. PredictionEngine

The `PredictionEngine` is responsible for suggesting items of content that are likely to be accessed by a particular user in the near future. It is therefore where all data analysis takes place. In order to accomplish this task, the `PredictionEngine` is informed of all user activity by the `ResourceServer`. The `PredictionEngine` is also provided the requested content itself, although it may or may not make use of this information. For example, in an implementation that is intended to apply to multiple application domains, a prediction engine would most likely process user request metadata but choose to ignore the content of these requests, as the structure of this content is likely to vary across different domains.

The interface of the `PredictionEngine` is primarily concerned with the exchange of information between itself and the `ResourceServer`. This includes receiving user data and content as input and producing prefetch suggestions as output.

Much like the `ResourceFetcher`, the `PredictionEngine` is generally expected to be implemented by the application developer. While this is a non-trivial task, it allows the middleware to be as flexible as possible and developers are free to use ready-made general implementations. Furthermore, we've kept the interface small, requiring the implementation of just four operations. As mentioned above, some implementations may attempt to mine request metadata, while others may examine the actual content that is consumed by users in order to identify patterns and make predictions. Additionally, some implementations may be designed to support prediction pulling, in which predictions are computed upon demand and delivered synchronously to the server, while others may support prediction pushing, in which user data is processed and predictions are produced in the background and asynchronously delivered to the server to be used at a later point in time.

E. Cloud and Mobile Caches

Similarly, the `CloudCache` and `MobileCache` are to be implemented by the application developer. The cloud cache stores preprocessed, precached content accessible by all users. The mobile caches on the other hand store processed, prefetched content specific to the user of the resident device. Both cloud and mobile caches can be implemented with unique policies to satisfy unique usage patterns, making the middleware as general as possible. In this way, the caches can be optimized with contextual knowledge from the mobile device, and auxiliary information from the prediction engine. This particular generalization is achieved without placing a burden on the developer. The cache interfaces, which match that of Java's `Map` interface, allows developers to use a relatively simple policy, such as LRU, simply by extending Java's `HashMap` class. Alternatively, they can tailor a more sophisticated policy to their application's needs, and user patterns.

F. Concurrency and Synchronization

Because the `ResourceServer` runs on cloud infrastructure and is expected to manage content for, and handle traffic from many mobile clients, there is strong motivation to achieve concurrency in its implementation. Not only do we have the necessary processing power available to us, but concurrency is also essential if the server is expected to scale well. To this end, the `ResourceServer` has generally been written to favor asynchrony and concurrency over synchronous and blocking computations. To achieve this, the `ResourceServer` makes use thread pools which generally need not contain more than a handful of threads. The size of the thread pools can be specified by the developers, which can be very useful for scaling the system up or down depending on expected load.

V. Application Case Study

A. News Aggregator Application

We have deployed a News Aggregator application that utilizes our middleware in order to evaluate the effectiveness of the system in a real-world setting. It is inspired by Flipboard [3], a commercially-available mobile application that integrates content from news providers, blogs, and social media into a single readable interface. Our application is a more simple news article aggregator, meaning it collects articles from various news outlets and presents them to the user for consumption. To begin, a user is first presented with a list of general topics. Upon choosing a topic, he or she is presented with a list of headlines pertaining to that topic and then must select an item from the list in order to read the corresponding article. The application uses an external library [4] to parse the source webpage of a news article and discard extraneous material like advertisements, comments, and distracting images. This allows the application to present only the body text of the article to the user.

This application exhibits several characteristics that make it amenable to cloud-based optimization. First, the application involves extensive communication due to the fact that it gathers information from several different news sources. Also, the application has high storage demands due to the large volume of articles that it may process and disseminate to its users. It is natural, therefore, to split the functionality of the application between the cloud and a user's mobile device. The cloud can perform article retrieval, extraction, and storage on behalf of the user. Because of their high resource demands, these tasks would be costly to perform and could disrupt other user operations if performed directly on a mobile device.

What makes the news aggregator even more suitable for cloud-based optimization, however, is its article extraction process. We found that the source webpage for a news article had an average size of about 260 KB, while the body text for that article had an average size of about 4 KB. Thus, if we perform article extraction in the cloud, we find that we can dramatically reduce the amount of data traffic induced by the application, at least from the perspective of a mobile device. However, the downside to this arrangement is that the extraction process is computationally expensive. We've found that it typically requires approximately 4 or 5 seconds of CPU time. This arrangement provides strong motivation to attempt to perform optimizations in order to achieve the data reduction benefits of extraction without suffering from its computational cost. Therefore, the fact that we've introduced the cloud as an extra hop between article source and end user can be justified by its ability to carry out such optimizations.

In order to apply the middleware in this context, we define the *region of interest* to be the topics and news items that a user most consistently wants to read about. For example, we may find that one mobile user enjoys reading about specific sports teams, while another likes to keep up with the latest political developments. Thus, if we can develop a reliable means of identifying these interests, we may be able to predict which articles a user is likely to read in the near future and perform the precaching and prefetching operations discussed earlier.

B. Prediction Engine

In order to test the middleware, a prediction engine that is suitable for the news aggregator is required. We use a prediction scheme that relies on the cosine similarity metric to identify similar pairs of users. We maintain a bit vector for each user, with one entry for each article known to the system. If the user has read that article, the corresponding entry in their bit vector is set to 1, otherwise it is 0. Then, given two user history vectors u and v, we compute their similarity as follows. This score must be between 0 and 1, and a higher score indicates a stronger similarity between two users.

$$s(u,v) = \frac{u \cdot v}{\|u\|\,\|v\|} \tag{1}$$

We can then construct the region of interest from all articles that have been read by individuals who are sufficiently similar to a given user. This assumes that users naturally fall into clusters based on their interests. We can define a similarity threshold T and assume that all individuals whose similarity to a given user does not exceed T are irrelevant to that user. Let $H(x)$ be the user history vector of user x and let $U(x)$ be the set of users who are sufficiently similar to x to be deemed relevant. Thus:

$$U(x) = \{y \mid s(H(x), H(y)) \geq T\} \tag{2}$$

Let $R(x)$ be the set of articles read by user x. We can then compute a user's region of interest as follows:

$$RoI(x) = \bigcup_{y \,\in\, U(x)} R(y) \tag{3}$$

This logic comes into play each time a user reads an article, as we then prefetch that article for all users who are considered sufficiently similar to the original user.

Our current implementation does not allow the `PredictionEngine` to suggest that an article be precached rather than prefetched, although enabling this behavior would only require a minor change to the `PredictionEngine` interface. Instead, the server will retain a copy of an article in its cache after prefetching or retrieving that article, which means that precaching currently occurs only as a side-effect of prefetching and unoptimized retrieval.

VI. Experimental Evaluation

A. Experimental Setup

We used a trace-driven emulation of mobile client requests in order to test the performance of the news aggregator combined with our middleware system. We used data from Twitter and Disqus to construct a workload trace for the news aggregator. Specifically, we collected all Tweets that were issued by CNN between early February and early April of 2014. CNN will emit Tweets to announce the posting of a new article to its website. We interpret this as a publish event, i.e. the article that was announced by the Tweet is now available for user consumption. Additionally, we used an API provided by Disqus [2] to collect all user comments on these articles. We interpret these as consumption events. That is, when a user comments on an article, we assume that they have read this article and treat this event as such in our workload trace.

In order to evaluate the effectiveness of the middleware, we emulated the interactions of 40 users, corresponding to the most prolific commenters from our dataset. More specifically, we deployed the `ResourceServer`, `PredictionEngine`, and `ResourceFetcher` components of the middleware on a c1.xlarge instance in Amazon's EC2 (Elastic Cloud Compute) infrastructure and then emulated the activities of 40 mobile clients corresponding to each of these users by emitting a sequence of requests that directly correspond to the sequence of comments in the dataset, thus replaying the original sequence of events. Due to the need to emulate such a large number of clients, we emulate client requests using a desktop PC. As we are primarily interested in latency and cloud-side processing, this does not significantly affect our results. However, to accurately capture the application performance on mobile devices, we submitted requests to the cloud-based server from an HTC One X phone running Android 4.1.1 and measured the average latency for prefetching. This value was used for the prefetching cost in our emulated experiments. In these experiments, we accelerated the rate at which requests were submitted to the server in order to reduce experimental run time.

B. Implementation Comparison

We ran the workload trace for several different server implementations in order to judge the effectiveness of the optimizations we propose as well as the user similarity analysis scheme. We tested the following implementations.

1) **No Precaching** is a version of the middleware's server component with neither precaching nor prefetching, i.e., all requests are served by the source.
2) **No Prefetching** is a server that precaches article content but does no prefetching. In this case, the server will serve requests from the cloud cache if articles are stored there, otherwise they'll be served from the source.
3) **Random** is a server that does both precaching and prefetching, however it makes no use of user profiling. It chooses whether or not to prefetch an article for a particular user with a 50/50 probability.
4) **User Similarity** is a server that also does both precaching and prefetching, but it incorporates our user similarity scheme to identify articles for prefetching.

It uses a default user similarity threshold of $T = 0.40$. Thus, when a user reads an article, that article will be prefetched for all other users with a similarity score above 0.40.

Unless noted otherwise, all experiments were run with a cloud cache size of 200 and mobile cache size of 10, both using an LRU eviction policy. For each of these implementations, we computed the following metrics:

1) **Latency**: We measure the time that elapses between a client issuing a request and the delivery of a response back to the user. This time can include network latency of going to the cloud/source as well as the computational cost of extracting article text, unless the article is prefetched, in which case it will correspond to the local access time.
2) **Data Transfer**: This is the average amount of data exchanged between the client and server for each user. Prefetching can increase the amount of network traffic between a mobile device and the server due to content that is prefetched but never consumed.
3) **Recall**: This is the average fraction of requests for each user that were satisfied using prefetched content when possible. The higher the recall, the more articles would be accessed locally from the mobile device, thus reducing the latency of access.
4) **Precision**: This is the average fraction of prefetched articles for each user that were subsequently read by that user. Higher precision values correspond to fewer false positives in terms of prefetched articles, thus leading to smaller wasted network bandwidth.

The comparison results between the different server implementations are given in Figure 3.

1) Latency: Figure 3a gives a CDF of request latency for each server implementation (Note: x-axis is log-scale). When neither precaching nor prefetching is used, the request latency for all requests is on the order of several seconds. This improves dramatically once precaching is enabled, leading to a latency of about 165 milliseconds for approximately 85% of the requests. The effectiveness of precaching justifies our use of the cloud as an intermediary between content sources and mobile devices. The latency improves even further when prefetching is enabled. Looking at the left extreme of the figure, we observe that even a random prefetching scheme is able to reduce the latency of about 20% of requests to a few milliseconds, while our more intelligent strategy based on user profiling achieves this reduced latency for approximately 40% of requests, thus significantly outperforming the random baseline algorithm. Focusing our attention to user similarity strategy unoptimized requests (no precaching/no prefetching) require an average of 5 seconds, which consists of both a network delay and data processing time. Requests for precached content require an average of 164 milliseconds, mainly due to network round trip time. Finally, prefetched requests require an average of 5 milliseconds, justifying any overhead the middleware may incur.

2) Data Transfer: Neither of the first two implementations engage in any prefetching, so they transfer only as much data as necessary to mobile clients. Prefetching increases the amount of network traffic between the server and the clients, since some of the prefetched articles may not actually be accessed by the users. Interestingly, the random and user similarity implementations incur roughly the same amount of network traffic, but we see that the user similarity scheme makes much more effective use of this data. From the first two graphs of Figure 3, we see that there is a direct trade-off between latency and data traffic. There may be some situations where high data traffic is tolerated for the sake of minimizing latency, while in other situations higher latency is tolerated to limit data traffic. As we show later, we can achieve the desired trade-off by changing the user similarity threshold T for our user similarity-based server.

3) Recall: When prefetching does not occur, the recall is necessarily 0 because no user requests can be satisfied with prefetched content. As our random prefetching scheme prefetches each article with probability 0.5, we would expect the theoretical upper limit of its recall to be around this number. In reality, the recall for the random implementation is much lower because it prefetches without any notion of user preferences. The user-similarity scheme, on the other hand, achieves a recall that is more than double that of the random scheme, clearly demonstrating the benefits of intelligent prefetching through user profiling.

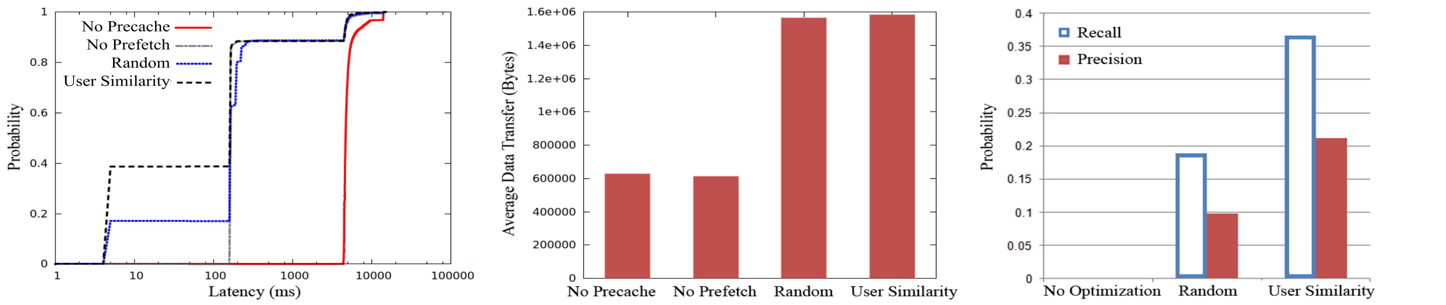
4) Precision: This metric is only applicable to implementations that actually perform prefetching. We see that approximately one in ten articles prefetched in the random implementation end up being consumed by the user, while more than one in five articles prefetched by the user similarity scheme are consumed. Thus, our user similarity scheme not only achieves much better recall than a random implementation, but it also achieves much better precision. Furthermore, it accomplishes this with roughly the same amount of network overhead.

C. Overhead

We carried out some basic microbenchmarking in order to evaluate the overhead incurred by the middleware's data processing efforts. That is, we sought to evaluate the extent to which the additional bookkeeping and data collection carried out by the middleware hinders the processing of a user request. In particular, we are interested in breaking down request latency into two pieces:

1) **Unavoidable Latency** occurs due to the article retrieval and extraction process.
2) **Avoidable Latency** is incurred by additional operations that are not strictly necessary to process a user's request but are required for the functionality of the middleware.

Because of the middleware's need to maintain user history vectors and to perform similarity computations, user requests will take longer to process than they normally would. We analyzed all unoptimized requests in the workload trace and found that average server response time was 4,864 milliseconds, while the average amount of time spent retrieving and extracting an article was 4,860 milliseconds. This means that, on average, only approximately 4 milliseconds of latency was caused by bookkeeping and other data processing operations. Hence, we see that, despite some of the extra work done by the



(a) Latency CDF (x-axis is log scale)

(b) Data Transfer

(c) Precision/Recall

Figure 3: Server and Prediction Engine Comparison

middleware, latency is strongly dominated by the unavoidable cost of retrieving and extracting article content.

We also explored the possibility that load placed on the server by background threads performing prefetching and data processing hindered the ability of the remaining threads to process user requests in a timely fashion. If a server's CPU is occupied by prefetching efforts, then incoming user requests may take longer to be serviced. To evaluate this, we compared the average response time of the non-prefetching server featured in Section VI-B to that of the user similarity-based implementation. We found that, when the two servers are deployed on multi-core virtual machines with identical specifications, there was no significant difference in response time. Background prefetching does degrade response time when deployed on a single-core machine, but we assume that the middleware will be deployed in a multi-core environment, as multi-core virtual machines are widely available from cloud providers and relatively affordable.

D. Parameter Analysis

Our middleware, as well as the prediction engine described in Section V-B, feature several parameters that can affect system behavior and performance. We examine three parameters here: the similarity threshold, server/mobile cache sizes, and cache policies.

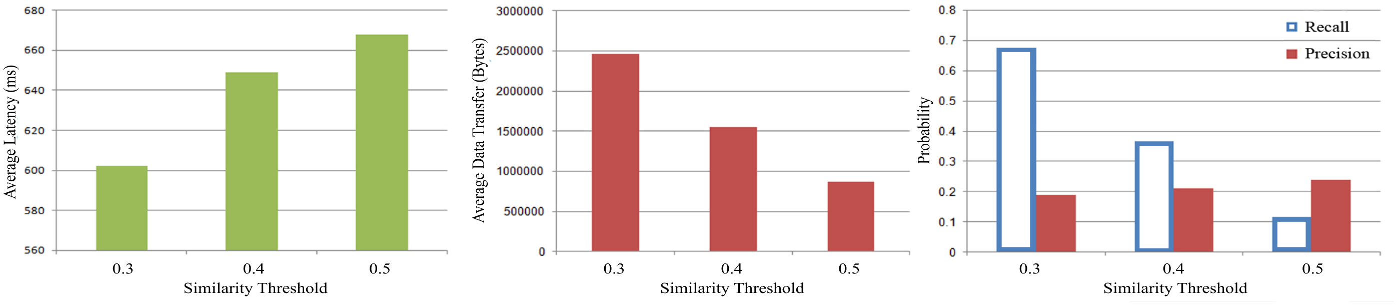
1) Impact of Similarity Threshold: The user similarity-based prediction engine uses a similarity threshold value T that determines which users are deemed relevant when constructing a region of interest. We evaluated the performance of three prediction engines – with T values of 0.30, 0.40, and 0.50 – using the same experimental procedure as in Section VI-B. The full results are given in Figure 4. As T increases, the system becomes more selective in its prefetching. Therefore, higher T values yield a smaller reduction in latency because fewer prefetches occur, but they also reduce the number of false positives, leading to lower values of data transfer and higher precision values. Lower T values produce a better average latency but also lead to more false positives. In short, the prediction engine's T value can be tuned to influence its precision and recall. In situations where low latency is important, the T value can be decreased to induce more aggressive prefetching, whereas in situations where reducing network traffic is critical, the T value can be increased to force more conservative prefetching.

2) Impact of Cache Size: We next examine the impact of the cloud-side server cache size (used for precaching) on system performance. The server may prefer to maintain a cache size that could fit in memory to provide fast accesses; therefore, smaller cache sizes may be desirable to reduce cost in the cloud. We varied the maximum number of resources that could be stored in the cloud-side cache and used an LRU replacement algorithm when this maximum was exceeded.

Figure 5a plots the number of unoptimized requests, i.e. those that had to be sent to the source, for each cache size (mobile cache size is fixed at 5 items). As expected, a smaller cache size leads to more unoptimized requests, so that there is a clear trade-off between server-side memory consumption and latency. This is because a smaller cache forces the server to discard resources that will be requested by users at a later time, causing it to miss opportunities for precaching and retrieve content directly from the source. Surprisingly, we see that a relatively small cache size still yields significant benefits. A server with a maximum cache size of roughly 50 entries performs comparably to servers with maximum sizes of 100 and 200 entries. This implies that it is possible to achieve good system performance even with a modestly-sized cache, which likely could be small enough to be stored in server memory to minimize access time. We also see an interesting trend in which decreasing the cache size below 50 very quickly degrades system performance, as evidenced by the results for a server with a maximum cache sizes of 25, 15 and 0 entries. These results suggest that a "optimal" cache sizes could be empirically determined to achieve a desirable trade-off between cost and latency.

We also ran a trace using data collected on all users, rather than the top 40 most prolific, with the same configuration as the preceding bar, i.e. cloud cache size of 200, mobile cache size of 5, and our user similarity prediction engine. The results of this test, seen as the last bars in Figure 5a demonstrate how even users who use the app infrequently are still able to benefit from its optimizations. This is a result of the content domain's strong temporal locality and high reuse of content among users.

3) Impact of Cache Policy: Different content domains require different cache eviction policies. Figures 5b and 5c

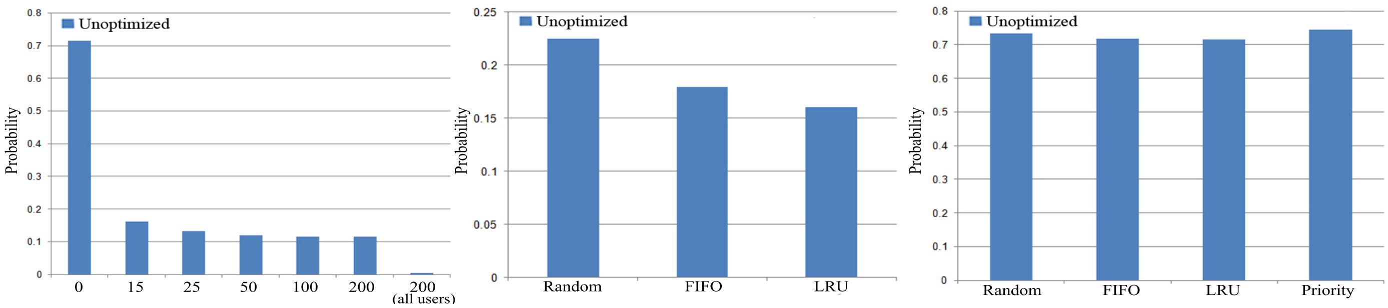


(a) Average Latency (ms)

(b) Data Transfer (Bytes)

(c) Precision/Recall

Figure 4: Effects of T Value on System Performance



(a) Cloud Cache Size

(b) Cloud Cache Policy

(c) Mobile Cache Policy

Figure 5: Cache Size and Eviction Policy Comparison

show the efficacy of various policies with respect to our news aggregator. Figure 5b compares policies used by the cloud cache restricted to a size of 15. We use a small cache size in order to demonstrate the efficacy of different policies, as larger cache sizes become insensitive to the cache eviction policy. This insensitivity indicates a small working set that fits within the cache. For this reason, the cloud's policy for this domain is arguably insignificant. Nevertheless, the figure reveals that an LRU policy for the cloud cache results in the fewest unoptimized requests, albeit narrowly. FIFO performs nearly as well as LRU as, in general, this domain exhibits read-once behavior. Eventually, content grows stale and falls out of use. The LRU policy is responsive to staleness and trending content, i.e. a popular article will stay popular and therefore avoid eviction.

Figure 5c is concerned with the policy used by the mobile cache with a fixed size of 5. The Priority policy evicts the article with the lowest score, set by the prediction engine upon prefetching, from the mobile cache. Additionally, the policy decays the score of an article while it is stored in the mobile cache and whenever it is accessed. This more sophisticated policy does not outperform others, as highly scored content can stick in memory. As evidenced by our results, the policy used for the mobile cache, much like that for the cloud cache, has little weight on the performance of our middleware. Given the accuracy of our prediction engine, we should expect that each element has a high probability of being accessed by the user. Therefore, the element we choose to evict tends to be irrelevant, so long as old content is eventually evicted. For this reason, a random eviction policy is comparable to LRU and Priority. This quality is advantageous for the developer as it allows he or she to implement a relatively simple policy. Even so, we leave the implementation to the developers who have greater insights into the requirements of the domain for which they are implementing our middleware framework.

VII. Conclusion

We have explored the application of user profiling techniques to the optimization of mobile software using cloud-based resources and infrastructure. Specifically, we examine content-centric applications and the use of two optimization techniques, precaching and prefetching, as a means of improving performance. We implemented a middleware system to address the system-level challenges of precaching and prefetching. Such challenges include issues of data collection, management, analysis, and propagation. Our system makes extensive use of asynchrony and concurrency to allow data analysis to occur in parallel with the servicing of user requests. This leaves just two primary tasks to a developer who wishes to integrate his or her application with this framework: implementing a means of interaction with content sources and implementing a mechanism to generate predictions of future user behavior from past history.

To assess the middleware, we implemented a news aggregation application and a corresponding prediction engine that performs user similarity analysis in an attempt to construct regions of interest using clusters of users with common interests. We used Twitter and Disqus data to construct a workload trace and used this as the basis of an emulation intended to evaluate the potential benefits of prefetching and precaching as enabled by the middleware. We found that many user requests in the emulation benefited from either precaching or prefetching, causing a reduction in the latency normally induced by intermediate content processing performed at the cloud. We also saw very limited overhead due to the additional tasks performed by the middleware during the processing of a user request. Moreover, our similarity-based implementation outperforms a baseline random algorithm in terms of both precision and recall.

There are several areas of potential future interest raised by this work. First, we intend to demonstrate and evaluate the middleware's ability to generalize to various content domains. Another interesting possibility is the idea of prioritizing some predictions for user behavior over others. This would most likely be based on prediction confidence, i.e. the estimated likelihood that a prediction will turn out to be true. In our system, this could be tied to the similarity score between two users. This may allow the middleware to dynamically adapt the aggressiveness of its prefetching to different conditions. Finally, user location could serve as an additional source of data that may allow the middleware to make more intelligent optimization decisions.

Acknowledgment

This work was supported by NSF Grant CSR-1162405.

References

[1] Amazon Silk. http://amazonsilk.wordpress.com. Accessed: 3-11-2014.

[2] API – Disqus. https://disqus.com/api/docs. Accessed 5-22-2014.

[3] Flipboard. https://flipboard.com/. Accessed 5-1-2014.

[4] GravityLabs/goose – GitHub. https://github.com/GravityLabs/goose. Accessed 4-21-2014.

[5] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer. Falling Asleep with Angry Birds, Facebook and Kindle: A Large Scale Study on Mobile Application Usage. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 47–56, New York, NY, USA, 2011. ACM.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. CloneCloud: Elastic Execution Between Mobile Device and Cloud. In *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, pages 301–314, New York, NY, USA, 2011. ACM.

[7] K. Church and N. Oliver. Understanding Mobile Web and Mobile Search Use in Today's Dynamic Mobile Landscape. In *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, pages 67–76, New York, NY, USA, 2011. ACM.

[8] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: Making Smartphones Last Longer with Code Offload. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 49–62, New York, NY, USA, 2010. ACM.

[9] M. D. Ekstrand, J. T. Riedl, and J. A. Konstan. Collaborative Filtering Recommender Systems. *Foundations and Trends in Human–Computer Interaction*, 4(2):81–173, 2011.

[10] H. Falaki, R. Mahajan, S. Kandula, D. Lymberopoulos, R. Govindan, and D. Estrin. Diversity in Smartphone Usage. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 179–194, New York, NY, USA, 2010. ACM.

[11] S. Funk. Netflix Update: Try This at Home. http://sifter.org/~simon/journal/20061211.html, December 2006. Accessed: 3-10-2014.

[12] K. Ha, Z. Chen, W. Hu, W. Richter, P. Pillai, and M. Satyanarayanan. Towards wearable cognitive assistance. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, pages 68–81, New York, NY, USA, 2014. ACM.

[13] J. L. Herlocker, J. A. Konstan, L. G. Terveen, and J. T. Riedl. Evaluating collaborative filtering recommender systems. *ACM Transactions on Information Systems*, 22:5–53, 2004.

[14] 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.

[15] A. J. Khan, K. Jayarajah, D. Han, A. Misra, R. Balan, and S. Seshan. Cameo: A middleware for mobile advertisement delivery. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 125–138, New York, NY, USA, 2013. ACM.

[16] J. Kolb, W. Myott, T. Nguyen, A. Chandra, and J. Weissman. Exploiting User Interest in Data-Driven, Cloud-Based Mobile Optimization. In *Proceedings of the Second IEEE International Conference on Mobile Cloud Computing, Services, and Engineeering*, 2014.

[17] C. Mei, D. Taylor, C. Wang, A. Chandra, and J. Weissman. Sharing-Aware Cloud-Based Mobile Outsourcing. In *5th IEEE International Conference on Cloud Computing*, IEEE Cloud 2012, pages 408–415, June 2012.

[18] P. Mohan, S. Nath, and O. Riva. Prefetching Mobile Ads: Can Advertising Systems Afford It? In *Proceedings of the 8th ACM European Conference on Computer Systems*, EuroSys '13, pages 267–280, New York, NY, USA, 2013. ACM.

[19] A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized Web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1155–1169, Sept 2003.

[20] A. Parate, M. Böhmer, D. Chu, D. Ganesan, and B. M. Marlin. Practical Prediction and Prefetch for Faster Access to Applications on Mobile Phones. In *Proceedings of the 2013 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, UbiComp '13, pages 275–284, New York, NY, USA, 2013. ACM.

[21] M.-R. Ra, A. Sheth, L. Mummert, P. Pillai, D. Wetherall, and R. Govindan. Odessa: Enabling Interactive Perception Applications on Mobile Devices. In *Proceedings of the 9th International Conference on Mobile Systems, Applications, and Services*, MobiSys '11, pages 43–56, New York, NY, USA, 2011. ACM.

[22] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-Based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, Oct 2009.

[23] M. Seno and G. Karypis. LPMiner: an algorithm for finding frequent itemsets using length-decreasing support constraint. In *Proceedings of the IEEE International Conference on Data Mining*, pages 505–512, 2001.

[24] M. Seno and G. Karypis. SLPMiner: an algorithm for finding frequent sequential patterns using length-decreasing support constraint. In *Proceedings of the IEEE International Conference on Data Mining*, pages 418–425, 2002.

[25] A. Smith. Smartphone Ownership – 2013 Update. http://www.pewinternet.org/files/oldmedia/Files/Reports/2013/PIP_Smartphone_adoption_2013_PDF.pdf, June 2013. Accessed: 2-27-2014.

[26] W.-G. Teng, C.-Y. Chang, and M.-S. Chen. Integrating Web caching and Web prefetching in client-side proxies. *IEEE Transactions on Parallel and Distributed Systems*, 16(5):444–455, May 2005.

[27] B. Zhang, G. Kreitz, M. Isaksson, J. Ubillos, G. Urdaneta, J. A. Pouwelse, and D. H. J. Epema. Understanding user behavior in spotify. In *INFOCOM*, pages 220–224. IEEE, 2013.