EdgeReduce: Eliminating Mobile Network Traffic Using Application-Specific Edge Proxies

Andreas Pamboris and Peter Pietzuch
Department of Computing, Imperial College London
Email: {ap5309, prp}@doc.ic.ac.uk

***Abstract*—Mobile carriers are struggling to cope with the surge in smartphone traffic, which reflects badly on end users who often experience poor connectivity in densely populated urban environments. Data transfers between mobile client applications and their Internet backend services contribute significantly to the contention in radio access networks (RANs). Client applications, however, typically transfer unnecessary data because (i) backend service APIs do not support a fine-grained specification of the data actually required by clients and (ii) clients aggressively prefetch data that is never used.**

We describe EdgeReduce, an automated approach for reducing the data transmitted from backend services to a mobile device. Based on source-level program analysis, EdgeReduce generates *application-specific proxies* for mobile client applications that execute part of the application logic at the network edge to filter data returned by backend API calls and only send used data to the client. EdgeReduce also permits the tuning of aggressive prefetching strategies: proxies replace large prefetched objects such as images by *futures*, whose access by the client triggers the retrieval of the object on-demand. We show that EdgeReduce reduces the RAN traffic for real-world iOS client applications by up to 8×, with only a modest increase in response time.

I. Introduction

Mobile network operators are projected to carry the bulk of "last mile" Internet traffic in the future. According to Cisco, global mobile data traffic will grow 10-fold from 2014–2019 [1]. Yet, in current mobile networks, operators struggle to keep up with the volume of data traffic. In particular, radio access networks (RANs) become a bottleneck due to the limits on the density of mobile base stations in urban environments and on the frequency spectrum that they can utilise [2]. Even the next generation of 4G/LTE networks is unlikely to meet the exponentially growing demand for mobile data capacity [3].

While about half of mobile data traffic constitutes video streaming, the other half is non-multimedia traffic, with a substantial fraction caused by the large number of client/server applications on today's smartphones [1]. Mobile client applications interact with Internet backend services, which host the application's content, through service APIs. For example, mobile clients for social networks, such as *Facebook* and *Twitter*, retrieve updates on user activity; photo sharing clients, such as *Picasa* and *Flickr*, host users' photo collections remotely; and e-commerce clients, such as *eBay*, *Groupon* and *Amazon*, provide the means for online purchasing of different goods. Clients typically access content through restful HTTP APIs, such as the Twitter REST API [4] and the Amazon Marketplace Web Service API [5].

We make the observation, supported by evidence in Section II, that *mobile client applications retrieve more data from backend service APIs than is strictly necessary, thus increasing the utilisation of RANs*. This has two main causes:

(1) Service APIs are designed with generality in mind, and not tailored to the needs of specific client applications. As a result, not all data returned by an API call is used by the client, with some discarded after transmission. For example, the Twitter API response to a request for the list of recent messages includes detailed user account information, which is typically ignored by clients.

(2) Client applications often aggressively prefetch binary content such as images from backend services. While this improves application response time when the user accesses prefetched content, it increases the amount of data transmitted over the RAN. Client applications often employ simple strategies such as prefetching all objects [6], which are wasteful. For example, Twitter clients typically prefetch all user profile images associated with a list of messages, even if only a few images will be viewed by the user.

Various techniques for reducing mobile data traffic were proposed in the past. Compression (e.g. *gzip* for HTTP traffic) is widely used to reduce the overhead of verbose application layer protocols such as XML used by backend services; new application layer protocols such as SPDY [7] and QUIC [8] are designed to decrease data transmission times by eliminating unnecessary communication. Client- or network-side caches [9] and redundancy elimination (RE) proxies [10] avoid the repeated transmission of the same data. All of these approaches, however, cannot prevent the transmission of *unused application data* across the RAN.

In contrast, our idea is to generate *application-specific proxies (ASPs)* located at the network edge, i.e. as part of the mobile network, with the goal of reducing data traffic from backend services to client applications. ASPs host the client application logic that parses response data from the backend service and stores the results as application data objects, which are in turn transmitted to the client application. Since data parsing is performed by the ASP, any data that is retrieved but not further used by the client application after parsing will not be transmitted to the mobile device.

We describe EdgeReduce, an approach for reducing RAN traffic for mobile client applications written in Objective-C on the iOS platform. EdgeReduce generates a stateless ASP automatically from the source code of a client application through program analysis. We assume that client applications are designed according to a *model-view-controller (MVC)* design pattern [11], which separates data presentation from data representation in an object-oriented design. EDGEREDUCE identifies classes from the application model that parse data from a backend service, which are placed at the ASP. Their access to data objects is transformed to *remote calls* over the RAN. Data returned by a backend service but never used as part of the model will therefore not be sent to the client.

To further reduce communication between the ASP and the client application, EDGEREDUCE employs two optimisations:

Creation of transient data objects. If data objects are updated by the ASP as part of the parsing process multiple times, it becomes more efficient to materialise them at the ASP and only transmit the final versions to the client. EDGEREDUCE identifies opportunities to create such transient data objects at the ASP when it reduces the number of remote calls.

Replacement of prefetched objects with futures. Client applications prefetch objects such as images from backend services to improve performance. Since prefetched objects are used by UI objects, they are transferred unnecessarily to the client before being displayed. The ASP therefore replaces references to large binary objects with *futures*, which are sent to the client instead. Upon accessing a future, the client retrieves the corresponding binary object from the ASP.

We evaluate EDGEREDUCE using three real-world mobile client applications for Twitter, Groupon and Yahoo! Finance on the iOS platform. We show that EDGEREDUCE can reduce RAN traffic by a factor of up to 8×, while only increasing application response time by at most 11%. We also show that EDGEREDUCE has the potential to speed up execution when large amounts of data are sent to client applications.

The remainder of this paper is organised as follows: Section II discusses existing approaches for mitigating mobile network contention and the potential to reduce network traffic by eliminating unused data sent to mobile client applications. Section III introduces the design of EDGEREDUCE, and Section IV explains how ASPs are generated from the source code of client applications written in Objective-C. In Section V, we evaluate the effectiveness and overhead of EDGEREDUCE, and conclude the paper with related work (Section VI) and conclusions (Section VII).

II. BACKGROUND

Next we discuss background on mobile networks and previous approaches for reducing network traffic. We also analyse the behaviour of client applications, particularly with regards to data transfers between clients and backend service APIs.

A. Mobile Networks

Mobile networks typically follow a hierarchical structure, which conceptually comprises three abstract layers: the edge, backhaul and core networks [9], [12]. The *edge network* is the *radio access network (RAN)* and consists of a set of *base stations* and mobile devices, which communicate directly over wireless communication channels, e.g. according to the 3G/4G standards. The *core network* interconnects devices on edge networks with the public Internet via gateway routers to IP networks, while the *backhaul network* is primarily responsible for controlling connected base stations and relaying data to edge or core networks.

Typically network capacity can be limited in edge and backhaul networks [13]. While in backhaul networks, mobile network operators have invested in fibre or high-frequency point-to-point wireless links to handle increasing data demands, in edge networks, increasing capacity is bounded by radio resource constraints of 3G/4G networks such as their limited frequency spectrum [2]. In addition, adding more base stations in a geographical region entails a high operational cost and is often constrained by frequency interference and placement problems. Deployments of smaller range base stations such as *femtocells* and *picocells* [14] can add more capacity to RANs, but require significant investment by mobile operators.

B. Reducing Mobile Network Contention

To address contention in mobile networks, operators have deployed or investigated a variety of solutions.

1) Caching: Many solutions for *caching* popular content to reduce network traffic were proposed in the past [9]. *Forward caches* use dedicated middleboxes for intercepting HTTP requests in backhaul networks [15], [16]. A request is relayed to the backend service only if it cannot be satisfied from the cache or if the requested content is not cacheable. Caching thus suppresses redundant data transfers from backend services.

Client-side caches [17] maintain copies of previously retrieved content on mobile devices in case it is requested again or required during offline operation. Maintaining large client-side caches, however, is infeasible due to the limited memory and storage resources of mobile devices.

Caches inherently require repeated requests for the same content to reduce traffic. They cannot reduce traffic from backend services that is unique and application-specific. Depending on the deployment locations, they mainly aim at reducing contention in backhaul networks, ignoring bottlenecks in RANs.

2) Compression: Compression is another widely-used approach for reducing network traffic. HTTP provides inherent support for the *gzip* and *deflate* compression methods. A client application announces its supported methods when issuing HTTP requests, and a backend service responds with compressed data in a supported format. For image transmission, transparent lossy compression is sometimes used [18]. This can substantially reduce the size of large images, albeit with a loss in image quality.

Compression cannot eliminate unused, application-specific data traffic sent by backend services. It is, however, an orthogonal approach in that it removes redundancy from any application data transmitted across the network.

3) Redundancy Elimination (RE): RE schemes are protocol-independent techniques for eliminating redundant network traffic [10], [19], [20]. For example, they may identify identical web content named by different URLs or delivered using different protocols by processing the payload of packets. Using synchronised caches at both ends of a bandwidth-constrained channel, they then exchange small fingerprints of cached data.

While RE can reduce redundant data traffic considerably, it suffers from the same problems as caching: it requires substantial resources at the client side, and it cannot reduce data traffic without any repetition.

4) Efficient Protocols: Another approach to reduce network contention is to deploy more efficient protocols. The *SPDY* protocol [7] is added between the application and transport layers to allow concurrent interleaved streams over a single TCP connection. To overcome bandwidth constraints, it implements request priorities, thus reducing contention caused by non-critical data. Though designed specifically for minimising transmission times, SPDY also compresses and in some cases eliminates request and response HTTP headers. However, given that headers range in size from 200 bytes to 2 KB, its ability to reduce overall data traffic is limited [7].

QUIC [8] is an experimental transport-level protocol that uses the connectionless nature of UDP to provide a multiplexed, congestion-aware transport with low latency. It is designed to avoid head-of-line blocking at the receiver end. However, it requires client- and server-side support and lacks the maturity of established transport-layer protocols.

New network protocols are typically application-agnostic, which means that they cannot exploit opportunities for reduction of backend data traffic that are application-specific.

5) Mobile Network Traffic Offloading: Switching from typical mobile base stations to special-purpose, short-range *femtocells* installed in residential environments also helps reduce traffic in mobile networks [14]. Similarly, with WiFi ubiquitously available in commercial hotspots or deployed for residential use, seamless WiFi offloading allows users to move from 3G/4G network connectivity to WiFi transparently [21]. Various studies [22]–[24] have shown merit in opportunistically migrating network traffic from mobile networks to WiFi access points in a delay-tolerant manner.

Mobile network traffic offloading presents itself as an ideal remedy to the problem of mobile data explosion. Nevertheless, it presupposes specialised access points available on demand, especially for non-delay-tolerant applications.

Overall, the approaches discussed above exploit opportunities for network traffic reduction that are independent of the business logic of applications. They regard applications as black boxes and therefore ignore inefficiencies of specific applications and backend service APIs, making them orthogonal to the techniques employed by EDGEREDUCE.

C. Mobile Client Communication

Modern mobile client applications communicate with backend services through web APIs. These APIs are implemented using the *HTTP* protocol and follow the *representational state transfer (REST)* architectural style [25], providing access to resources referenced by global URL identifiers.

Backend APIs expose a programmatic interface using well-defined request-response interaction. Typically backend APIs support (i) the retrieval of lists of data items, such as products in *eBay*, messages and friend lists in *Twitter*, photos in *Flickr* and financial data in *Yahoo! Finance*; (ii) the search for specific data items given user-defined criteria, such as products by category, friends by name or images based on metadata; and (iii) the updating of content such as adding new products or social status updates. API calls provide general operations on the data maintained by the backend service without taking specifics into account of how a given client application presents the data to the users.

Applications that interact with backend APIs need to understand the format of the data returned. A common practice is to use human-readable text-based encoding formats, which facilitates interoperability and cross-platform support. The most commonly used formats are the *extensible markup language (XML)* and the *JavaScript object notation (JSON)*. Their benefits in terms of simplicity and interoperability, however, come at the cost of substantially higher encoding overheads: e.g. XML includes metadata about the data schema as part of each data message. This leads to an increase in the transmitted data compared to application-specific binary encoding formats.

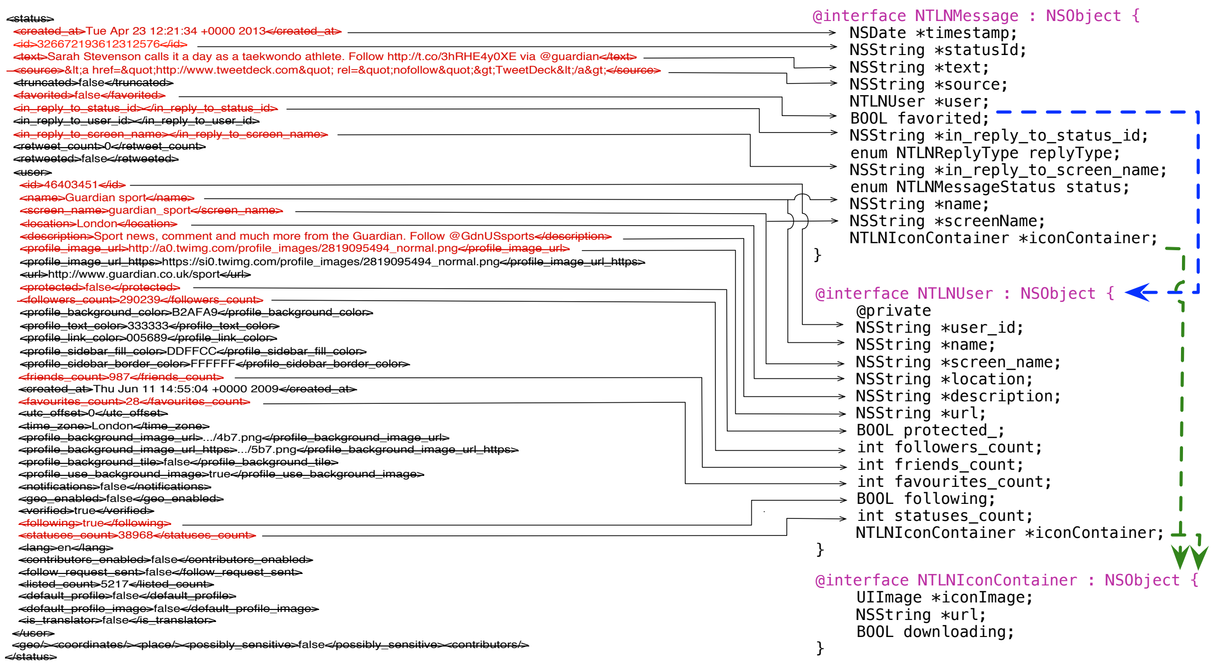
In Figure 1, we show part of the data returned by the Twitter REST API in response to a user request for the most recent Tweet messages. On the left-hand side, we show the representation of a single Tweet message in XML, as returned by the backend service. The right-hand side shows the corresponding fields of three data objects, `NTLNMessage`, `NTLNUser` and `NTLNIconContainer`, that a Twitter client application uses to model this information. Arrows indicate the pieces of data used to initialise object fields and items that are crossed out correspond to the encoding overhead of XML.

The `NTLNMessage` object stores information about a Twitter message such as its timestamp, its content and the posting user. It also contains pointer references to the `NTLNUser` object, which stores information about the user responsible for the message, and the `NTLNIconContainer` object with a user's profile picture.

As shown in the example, less than half of the data returned by the backend service is actually used by the client application: the original XML message has 2429 bytes, while the created data objects only occupy 842 bytes in memory. In general, there are three main sources of inefficiencies, I1–I3, when a client application interacts with a backend service:

I1. Inefficient data representation. Data items returned by the backend service API are expressed using an inefficient encoding format. In the example in Figure 1, a large part of the overhead is due to the repetitive nature of start and end tags, which capture the XML data schema. In addition, numerical values and binary data are encoded as text values, which also contributes to the increased size of messages sent, compared to the size of the (binary) data objects in memory.

This inherent redundancy in the response data yields high compression ratios. Therefore, many backend service APIs that use HTTP with XML or JSON also use transparent gzip compression, as supported by web servers and clients.



```
<status>
 <created_at>Tue Apr 23 12:21:34 +0000 2013</created_at>
 <id>326672193612312576</id>
 <text>Sarah Stevenson calls it a day as a taekwondo athlete. Follow http://t.co/3hRHE4y0XE via @guardian</text>
 <source>&lt;a href=&quot;http://www.tweetdeck.com&quot; rel=&quot;nofollow&quot;&gt;TweetDeck&lt;/a&gt;</source>
 <truncated>false</truncated>
 <favorited>false</favorited>
 <in_reply_to_status_id></in_reply_to_status_id>
 <in_reply_to_user_id></in_reply_to_user_id>
 <in_reply_to_screen_name></in_reply_to_screen_name>
 <retweet_count>0</retweet_count>
 <retweeted>false</retweeted>
 <user>
  <id>46403451</id>
  <name>Guardian sport</name>
  <screen_name>guardian_sport</screen_name>
  <location>London</location>
  <description>Sport news, comment and much more from the Guardian. Follow @GdnUSsports</description>
  <profile_image_url>http://a0.twimg.com/profile_images/2819095494_normal.png</profile_image_url>
  <profile_image_url_https>https://si0.twimg.com/profile_images/2819095494_normal.png</profile_image_url_https>
  <url>http://www.guardian.co.uk/sport</url>
  <protected>false</protected>
  <followers_count>290239</followers_count>
  <profile_background_color>B2AFA9</profile_background_color>
  <profile_text_color>333333</profile_text_color>
  <profile_link_color>005689</profile_link_color>
  <profile_sidebar_fill_color>DDFFCC</profile_sidebar_fill_color>
  <profile_sidebar_border_color>FFFFFF</profile_sidebar_border_color>
  <friends_count>987</friends_count>
  <created_at>Thu Jun 11 14:55:04 +0000 2009</created_at>
  <favourites_count>28</favourites_count>
  <utc_offset>0</utc_offset>
  <time_zone>London</time_zone>
  <profile_background_image_url>.../4b7.png</profile_background_image_url>
  <profile_background_image_url_https>.../5b7.png</profile_background_image_url_https>
  <profile_background_tile>false</profile_background_tile>
  <profile_use_background_image>true</profile_use_background_image>
  <notifications>false</notifications>
  <geo_enabled>false</geo_enabled>
  <verified>true</verified>
  <following>true</following>
  <statuses_count>38968</statuses_count>
  <lang>en</lang>
  <contributors_enabled>false</contributors_enabled>
  <follow_request_sent>false</follow_request_sent>
  <listed_count>5217</listed_count>
  <default_profile>false</default_profile>
  <default_profile_image>false</default_profile_image>
  <is_translator>false</is_translator>
 </user>
 <geo/><coordinates/><place/><possibly_sensitive>false</possibly_sensitive><contributors/>
</status>
```

```
@interface NTLNMessage : NSObject {
    NSDate *timestamp;
    NSString *statusId;
    NSString *text;
    NSString *source;
    NTLNUser *user;
    BOOL favorited;
    NSString *in_reply_to_status_id;
    enum NTLNReplyType replyType;
    NSString *in_reply_to_screen_name;
    enum NTLNMessageStatus status;
    NSString *name;
    NSString *screenName;
    NTLNIconContainer *iconContainer;
}

@interface NTLNUser : NSObject {
     @private
     NSString *user_id;
     NSString *name;
     NSString *screen_name;
     NSString *location;
     NSString *description;
     NSString *url;
     BOOL protected_;
     int followers_count;
     int friends_count;
     int favourites_count;
     BOOL following;
     int statuses_count;
     NTLNIconContainer *iconContainer;
}

@interface NTLNIconContainer : NSObject {
     UIImage *iconImage;
     NSString *url;
     BOOL downloading;
}
```

Fig. 1: Data returned by the Twitter backend service API and its subsequent use in a Twitter client application

I2. Unnecessarily returned data. A backend service API may return more data items or fields than necessary when compared to what is used by the client application. In the Twitter example above, the coarse granularity of the Twitter REST API does not support a fine specification of the data of interest: the Twitter client cannot express that it does not require statistics about the past user behaviour, the user's display settings or other account information, such as its creation date, timezone or language preferences. All of this metadata is included in the response data regardless.

To address this problem, it is usually necessary to change the backend service API. For example, Facebook offers a new API, the *Facebook Query Language (FQL)* [26], which allows client applications to query for user data using an SQL-style interface. This permits clients to retrieve precisely the required data by specifying filters on data items. However, such more expressive backend service APIs require the re-engineering of existing client applications and add more complexity to the development of clients.

In practice, there is typically little incentive for service providers to change an already established backend service API due to the re-engineering effort required for existing client applications. In contrast, EDGEREDUCE reduces RAN utilisation independently of the efficiency of backend APIs. While it relies on the availability of application source code, it does not burden developers by transforming mobile client applications automatically.

I3. Unnecessarily prefetched data. A client application may prefetch many data items, which will not be all used by the application. For example, the Twitter client application prefetches all images associated with retrieved Tweet messages. In Figure 1, when a user requests the most recent Tweet messages, the data items received by the client application contain the URLs of images associated with a given message. When processing this data, the client retrieves all images by default, which are stored in the `iconImage` field of the `NTLNIconContainer` objects. However, each image is stored locally and only displayed when a user views the message associated with the image. Depending on user behaviour, only a fraction of the images returned by the backend service are ultimately displayed on the mobile device.

Tuning the prefetching behaviour of a client application requires changes to its business logic. In general, prefetching requires the choice of a policy that strikes a balance between the amount of prefetched data and the probability that a requested object was prefetched. A wide range of prior work exists on effective prefetching strategies, e.g. in the context of web applications exploiting spatial locality, pattern mining and contextual information [6], [27]. For simplicity, however, mobile client applications typically do not use sophisticated prefetching policies but prefetch all objects instead.

D. Mobile Client Application Architecture

Mobile applications on the iOS platform are typically structured around a variation of the *model-view-controller (MVC)* design pattern [11]. An MVC design separates the representation of information from the user's interaction with it in an object-oriented application. As a result, applications are more easily extensible since objects become reusable and their interfaces are clearly defined.

As shown in Figure 2, the MVC design pattern has three types of objects, *Model*, *View* and *Controller* objects, which are separated by interfaces over which they communicate with each other: View objects represent the user interface (UI)

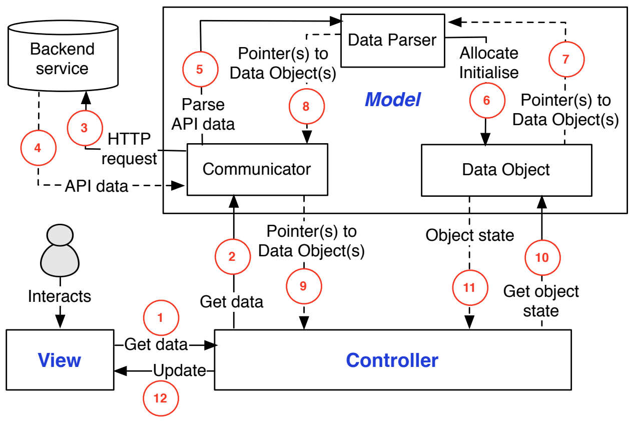


Fig. 2: MVC design pattern used in mobile client applications

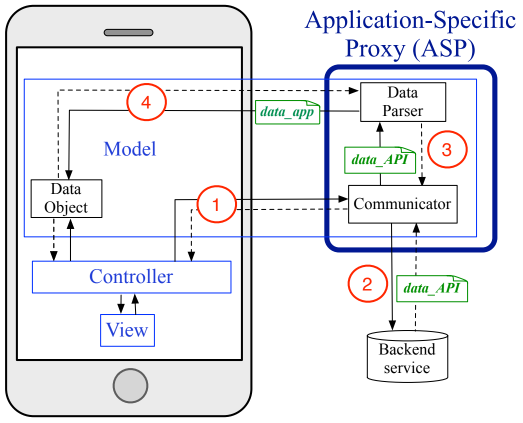


Fig. 3: Filtering of backend API data

of applications; Model objects encapsulate application data and corresponding operations on that data (i.e. the business logic of the application); and Controller objects mediate input between the two by converting user actions into commands, thus keeping them separate.

The MVC pattern is promoted heavily by Apple for iOS development. The Cocoa Touch frameworks are designed around MVC, while XCode, Apple's integrated development environment, creates class stubs for View and Controller objects automatically upon creation of a new iOS project.

1) Classification of Model Objects: Based on an examination of mobile client applications on the iOS platform (see Section V), we classify Model objects according to their role, as shown in Figure 2. *Communicator* objects interface with a backend service API; *Data Parser* objects process the network data returned by backend API calls and convert them to separate data fields with semantic meaning to the application; and these are stored as part of *Data Objects*, which encapsulate the data used by the application and their associated operations.

This classification is further evidenced by the fact that the iOS Developer Library encourages developers to follow the `MVCNetworking` sample [28], which displays a photo gallery obtained from a web server. `MVCNetworking` conforms to the architecture from Figure 2: Data Objects (`PhotoGallery` and `Photo`) represent a gallery of photos on the network; Communicator objects (`NetworkManager` and `QHTTPOperation`) manage the core network interactions and execute HTTP requests; and Data Parser Objects (`GalleryParserOperation`) parse the XML photo gallery data.

2) Application Workflow: Based on the above classification, Figure 2 shows the sequence of steps leading to the display of information retrieved from a backend service by a user.

When users decide to view, for example, a list of their latest Tweet messages, they interact directly with the appropriate View object. The user request is handed over to the corresponding Controller object (step 1), which in turn relays the request to a Communicator object (step 2). The latter issues an HTTP request to the Twitter backend service (step 3), which responds with the data that encapsulates the requested information (step 4). This is passed to a Data Parser object for deserialisation (step 5), after which the corresponding Data Objects, which model the information according to the application semantics, are initialised accordingly (step 6). Pointers to the Data Objects are then returned to the Controller object (steps 7–9) and are used to update the corresponding View objects (steps 10–12) on the mobile device.

3) Opportunities and Challenges: We observe that Communicator and Data Parser objects only create Data Objects for content that is used subsequently by Controller objects. Any unused data that is sent from the backend service will not be output by Data Parser objects. Therefore, less data should be output by the Data Parser object compared to what was retrieved by the Communicator object, addressing inefficiencies I1 and I2 from Section II-C.

The above observation, however, does not address the inefficiency I3 due to unnecessarily prefetched data. For prefetched data such as images, Data Objects are created by the Data Parser and subsequently accessed by the Controller. The distinguishing feature to identify unused prefetched Data Objects is that they are never used by View objects.

III. EdgeReduce Design

To address the inefficiencies I1–I3 from Section II-C, EdgeReduce extracts *application-specific proxies (ASPs)* from client applications to filter the data returned by backend services. ASPs can be deployed at the network edge—either directly on mobile base stations [29] or within the mobile backhaul network, e.g. at radio network controllers or gateway equipment (see Section II-A).

The operation of EdgeReduce involves two steps: (i) a *static analyser* identifies the application logic that needs to be included in the ASP given the source code of the client; and (ii) a *source-level compiler* generates the ASP implementation and transforms the client application so that all communication with the backend service occurs via the ASP.

A. Filtering of Backend API Data

In Figure 3, we give an overview of a client application after it was transformed by the EdgeReduce approach. The

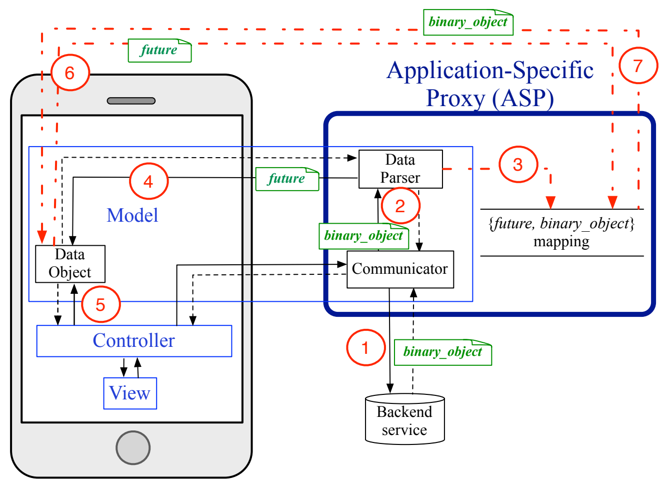


Fig. 4: Replacing large binary Data Objects with futures

generated ASP contains the *Communicator* and *Data Parser* classes of the original client application, which are responsible for retrieving (steps 1 and 2) and processing (step 3) the data returned from the backend service API calls, $data\_API$. All other application classes remain on the client side and are only exposed to $data\_app$, which is the portion of $data\_API$ that is used by the client application to construct its application data models (step 4). Any additional unused data included in $data\_API$ is thus not delivered to the client application, relieving the RAN from unnecessary data transfers.

B. Replacing Data Objects with Futures

Client applications often prefetch large binary Data Objects such as images, which are not necessarily used by the client's View objects (see Section II-C). EDGEREDUCE addresses this problem as follows: the ASP can replace large binary Data Objects with *futures*, which are significantly smaller in size. It therefore avoids the transmission of these objects over the RAN until they are actually used by the client application. In this case, the client application retrieves them from the ASP, using the corresponding futures as a reference.

The optimisation for replacing Data Objects with futures is shown in Figure 4. When an API call returns a binary Data Object $binary\_object$ (step 1), the response data is passed to the Data Parser object (step 2). Before the Data Parser object instantiates the corresponding Data Object, it creates a new association between the $binary\_object$ and a fresh *future* (step 3). The $binary\_object$ is stored at the ASP and its future is given to the Data Object (step 4). When the client attempts to access the $binary\_object$ (step 5), it triggers a request to the ASP using the future (step 6). The ASP then returns the original object to the client (step 7). As a result, a prefetched $binary\_object$ that is never accessed by the client will remain at the ASP and not transferred over the RAN.

IV. Proxy Generation

Next we describe the process of generating the ASP from the source code of a client application. This includes: (i) identifying the Communicator and Data Parser classes through source-level program analysis (Section IV-A); and (ii) transforming the client's source code to place the Communicator and Data Parser objects as part of the ASP, which involves converting the corresponding local method calls between Controller and Communicator objects, as well as Data Parser and Data Objects, to remote calls (Section IV-B).

A. Source-Level Program Analysis

EDGEREDUCE statically analyses the source code of client applications to distinguish between the different types of application classes according to the classification discussed in Section II-D. The goal is to identify Communicator and Data Parser classes to be placed at the ASP.

Communicator classes are selected based on the fact that they include methods that interact with objects of the `NSURLConnection` class. This class is defined in the iOS Foundation framework, which is a base layer for all primitive Objective-C classes. `NSURLConnection` objects retrieve data from a URL in a synchronous or an asynchronous fashion. Objects of this type are used to interface with the API of backend services.

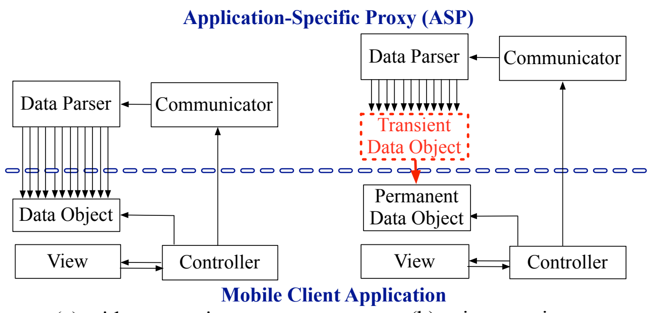
Data Parser classes are defined as the classes that perform serialisation and deserialisation of data transmitted over the network. Usually this kind of functionality is realised by built-in iOS classes such as the `NSJSONSerialization` class or third-party libraries such as the `SBJSON` library. For EDGEREDUCE, we manually compiled a list of such classes from libraries for the two most commonly used serialisation formats, namely the XML and JSON. This list is given as input to EDGEREDUCE's static analyser. For example, for XML, Data Parser classes include the `NSXMLParser` class and the `libxml2` C library; for JSON, they include the `NSJSONSerialization` class and the `JSONKit`, `TouchJSON` and `SBJSON` libraries. EDGEREDUCE can be extended to support new encoding formats, provided that classes handling data according to the format can be identified.

B. Source Code Transformation

EDGEREDUCE's *source-level compiler* generates ASPs for client applications written in Objective-C. We describe how the Communicator and Data Parser classes are placed at the ASP using source-level compilation techniques, which transform local into remote method calls.

We also explain two techniques for reducing the communication overhead between the ASP and the client application: (i) the use of transient Data Objects at the ASP, which are sent to the client in a single remote call, thus avoiding multiple calls that initialise individual object fields (Section IV-B2); and (ii) the replacement of large binary Data Objects by *futures*, which are retrieved from the ASP on demand.

1) Implementation: Communicator and Data Parser classes need to become part of the ASP. EDGEREDUCE replaces these classes in the source code with representative *delegate classes*, which relay method invocations from the client to the ASP using remote calls. Similarly, for the ASP, delegate classes are used to represent all other application classes, which are implemented by the client but are also accessed by the ASP.



(a) without transient Data Object (b) using transient Data Object

Fig. 5: Use of transient Data Object to reduce remote calls

Delegate classes are generated automatically using the *Internet Communications Engine* (ICE) [30], an object-oriented RPC framework with support for Objective-C. They have the same method signatures as the underlying classes and handle the serialisation and deserialisation of method parameters and return values. The RPC framework also automatically compresses data exchanged as part of remote calls.

To manage pointers to objects that are passed as arguments or returned by remote calls, EDGEREDUCE assigns a unique *object identifier* to each object upon its creation. This is used to replace pointers to objects in remote calls. Using the object identifier, the client application and the ASP can convert pointer arguments and return values to the appropriate objects in their address spaces, i.e. either to an application object instance or its delegate object. A global *identifier dictionary* maps identifiers to application object instances or delegate objects. When handling incoming remote calls, the referred objects are retrieved based on their identifiers.

2) Transient Data Objects: As illustrated in Figure 5a, Data Parser objects may interact frequently with Data Objects, which results in a high number of remote calls between the ASP and the client. These method calls typically correspond to calls to the Data Objects' setter methods, which initialise each of their fields separately rather than using a single call.

Figure 5b shows how EDGEREDUCE overcomes this problem by enabling the ASP to execute frequent calls to Data Object methods locally. When allocating and initialising these objects, the ASP uses *transient Data Objects*, which are Data Objects that are created at the ASP—instead of the client. Transient Data Objects remain valid until the ASP returns them to the client application via a remote call made by the Controller object. This requires serialising transient Data Objects and returning their actual data to the client, instead of using an object identifier. The client then initialises its permanent Data Objects accordingly.

3) Data Objects With Futures: With aggressive prefetching strategies, large binary objects may be transferred unnecessarily to the client application. To address this limitation, the ASP uses smaller *futures* to replace binary objects, which are retrieved on demand only when about to be used by the client.

Our prototype implementation focuses on images as a representative example of large binary Data Objects, since images constitute the largest payload of an average web page. Especially with high-resolution images becoming common [31], the amount of image data sent to a client application often contributes significantly to the RAN traffic.

EDGEREDUCE's static analyser automatically identifies Data Object fields that represent images, i.e. pointers to the `UIImage` class. It also identifies locations in the source code that initialise these fields using the image data received from a backend service. At the ASP, each image is assigned a unique *image identifier* to serve as a *future* for the image. Before an image is sent to the client application, the identifier is added to an *images* dictionary, indexed by its corresponding identifier.

At the ASP, before an image is used to initialise a Data Object field, it is replaced by the corresponding future. The client intercepts methods that are used to display an image on the screen, e.g. the `drawRect` method of the `UIImageView` class—a view-based Objective-C container for displaying and animating images—to first retrieve the image from the ASP using the future as a reference.

An implementation challenge is how to modify methods of classes for which there is no source code, such as the built-in Objective-C methods for the `UIImageView` class. As a solution, we use *method swizzling* [32]: when the Objective-C runtime loads a binary, all objects have their fields and method implementations defined in memory. These object templates also include a map associating method names with implementations. The Objective-C runtime allows for modifying these mappings at runtime by either replacing the original implementation with a user-defined function, thus modifying the default method behaviour, or changing the method name for a built-in Objective-C method.

EDGEREDUCE's source-level compiler is thus able to patch existing methods with the replacement methods: the original implementation of, e.g., `drawRect` is changed to `originalDrawRect`; and the `drawRect` method is mapped to a new user-defined method that first retrieves the image given its future from the ASP and then calls the `originalDrawRect` method.

For an EDGEREDUCE implementation on the Android platform, the above approach is infeasible due to a lack of support for method swizzling. An alternative approach would be to wrap an unmodifiable class with a custom wrapper class that has the same interface, which would thus allow the interception of methods that display images on the device. This would require, however, substituting all occurrences of the original class in the source code with the wrapper class.

C. Discussion

For our prototype implementation of EDGEREDUCE, we assume that the Controller and Data Parser objects contain no device-specific functionality such as accesses to hardware sensors or the file system. Our experience with mobile client applications suggests that these objects are independent and have no side-effects other than that of retrieving and parsing data sent from backend services. If dependencies exist, static analysis can be used to identify them.

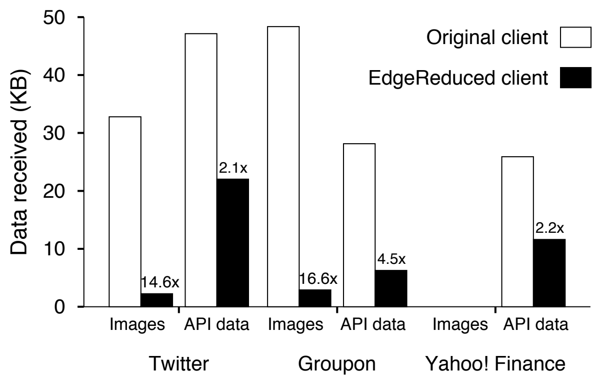


Fig. 6: Effect of EDGEREDUCE on RAN traffic

In addition, we assume that Data Parser objects make use of a small set of well-known libraries for parsing XML- and JSON- encoded data. Since most mobile client applications use a handful of network protocols and formats, this is the case for the majority of applications. For this, EDGEREDUCE's static analyser is based on observations drawn from an existing set of open-source client applications for iOS. This may not exhaustively identify all occurrences of Communicator and Data Parser objects, but it can be easily extended to include a wider set of attributes for identifying the role of different application classes.

The use of ASPs raises security challenges because client data becomes exposed at edge nodes. However, we assume that edge nodes are under the control of the mobile network operator and therefore, similar to network links, must be considered trusted. Multiple ASPs can be hosted efficiently by virtualised base stations in an isolated fashion [29].

V. EVALUATION

In this section, we evaluate experimentally the ability of EDGEREDUCE to reduce RAN usage for a realistic set of mobile client applications. We also quantify the impact of EDGEREDUCE on the response time of client applications.

A. Experimental Set-up

For our experiments, we use an Apple iPhone 4s to host the client application and a 2.26 Ghz Intel Core 2 Duo machine with 8 GB of RAM to host the ASP. We conduct experiments with two types of network connectivity between the nodes: (i) a 802.11g WiFi network with an average round trip time (RTT) of 23 ms and an average bandwidth of 8 Mbps; and (ii) a 3G mobile network with an average RTT of 425 ms and bandwidth of 0.4 Mbps.

We apply EDGEREDUCE to three iOS client applications for Twitter, Groupon and Yahoo! Finance:

The **Twitter client [33]** (Twitter) displays and supports the sharing and posting of Tweet messages. It interacts with the Twitter platform via the Twitter REST API [4], which provides interfaces for accessing and manipulating Twitter data such as timelines, followers and messages. We consider a workload in which the user retrieves the most recent Tweet messages.

The **Groupon client [34]** (Groupon) displays popular Groupon deals. It interacts with the Groupon platform via the Groupon API [35], which provides interfaces for categorised access to deals, such as location-aware deals and travel deals. Our workload constitutes of retrieving the latest deals in a given geographic location.

The **Yahoo! Finance client [36]** (Yahoo) produces plots of stock quotes using financial data from the Yahoo! Finance platform via their API [37]. The workload for this application involves retrieving and plotting the stock quote data for a set of stock symbols.

All HTTP traffic in the experiments is compressed using gzip. The results reported correspond to averaged values over 10 experimental runs, with a low variance between runs.

B. Network Bandwidth Usage

First, we compare the network usage of the *original* and EDGEREDUCE versions of all three applications. We show the relevant reduction in RAN data traffic with respect to the workloads described above. Figure 6 plots the breakdown of network usage, split according to image and non-image API data, for the Twitter, Groupon and Yahoo clients, before and after transformation with EDGEREDUCE.

With respect to non-image API data, EDGEREDUCE manages to reduce traffic by $2.1\times$, $4.5\times$ and $2.2\times$ for Twitter, Groupon and Yahoo, respectively. The reductions are due to the elimination of encoding overheads and returning only the portion of the data that is used by the client applications. In addition, EDGEREDUCE reduces the amount of image data received by the Twitter and Groupon clients by a factor of $14.6\times$ and $16.6\times$, respectively. (The Yahoo client does not perform image transfers.) This is accomplished by transmitting only approximately 7% of the total images returned by the Twitter and Groupon backend services when a user requests the most recent Twitter messages or popular Groupon deals. For the remaining images, futures in the form of image identifiers are returned, which are significantly smaller in size (4 bytes per future versus, on average, 2.2 KBytes per image). Of course, depending on user activity, savings in image data may diminish in accordance to the number of images that the user decides to browse in the client application.

Overall, a significant reduction in RAN traffic is obtained for all three client applications using EDGEREDUCE. Both Twitter and Groupon exhibit similar reductions in image data transfers by avoiding the transmission of approximately the same number of images, which are returned by the corresponding backend API calls. However, the response to the Groupon API call for the most popular deals contains approximately twice as much unused data compared to the response to the Twitter API call for recent Tweet messages. This can be observed in the relevant savings for non-image data that EDGEREDUCE achieves in both cases.

C. Application Performance

Next we explore the impact of EDGEREDUCE on application response times. We focus on the increase in response time

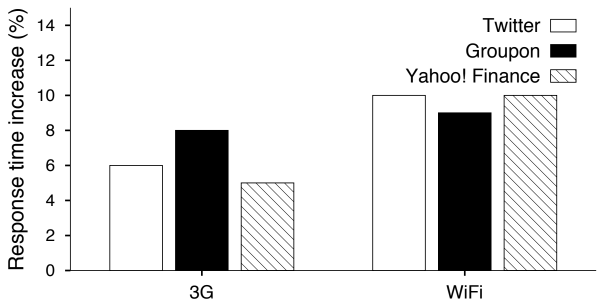


Fig. 7: EDGEREDUCE performance overhead

TABLE I: Transient Data Object optimisation savings

Client application	Number of RPCs		Reduction in response time over 3G (in secs)
	before	after	
Twitter	268	36	101
Groupon	106	18	39
Yahoo	11	11	0

due to the communication overhead of the additional remote calls introduced by EDGEREDUCE, as well as the overhead of the on-demand retrieval of binary objects. To account for the interoperability of EDGEREDUCE with seamless WiFi offloading approaches (see Section II-B), we also include results that show the impact of EDGEREDUCE on application performance over WiFi networks.

1) Application Response Time: In Figure 7, we compare the response times of the original and EDGEREDUCE versions for the three applications. We measure response time as the time between when a user action is initiated and when it results in a UI update. Using EDGEREDUCE, all three client applications are marginally outperformed by their original versions.

We observe an increase in response time that ranges from 5% to 11%, which is due to the additional remote calls between the client application and the ASP. The majority of remote calls are used to copy transient Data Objects to permanent Data Objects on the client side, as well as to return the futures of binary Data Objects to the client application.

In all cases, EDGEREDUCE performs slightly better when the mobile device and the ASP are connected over a 3G network. This is due to the relative difference between the data transmission rates over WiFi and 3G networks. Gains in response time are obtained by avoiding the transmission of unnecessary data across the substantially slower 3G network. For WiFi networks, however, the absolute saving is negligible—the contribution of the transmission time of unused data over WiFi to the overall response time is low.

Due to the optimisation of transient Data Objects (see Section IV) the overhead of the additional remote calls remains low. Table I shows the impact of the optimisation on the performance of the three applications. For Twitter and Groupon, it achieves significant reductions in the number of RPCs—by a factor of 7.4× and 5.9×, respectively. This reduces the response time by 101 s and 39 s, respectively. For Yahoo, however, the optimisation does not yield any benefit. Data Parser objects already output an intermediate representation

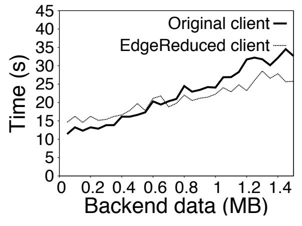


Fig. 8: RAN transmission times for varying data sizes

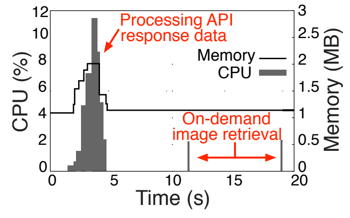


Fig. 9: CPU and memory requirements of ASPs

of the required API data, e.g. an array of used values, which is sent to the mobile client from the ASP in a single call.

2) On-Demand Object Retrieval: To evaluate the impact of EDGEREDUCE's future mechanism on application performance, we also observe the user-perceived latency when retrieving an image based on its future. We measure the time between a user request for displaying an image until the image was rendered on the screen. This includes the processing time at the ASP, i.e. the lookup time for the image object using its future, and the delay for transmitting the image to the client.

For the WiFi network, requesting and transmitting a single image from the ASP to the mobile device takes 102 ms, out of which only 33 ms are due to the processing delay. Over the 3G network, the delay increases to 509 ms due to the higher network latency—the processing delay remains the same. As expected, there is a trade-off between reducing RAN usage and providing the lowest application response times. Retrieving binary Data Objects on demand significantly reduces network usage, however, at the cost of degraded application performance.

3) Backend Data Size: We also explore the effect of larger amounts of data sent from the backend service on the application response time over a 3G network. We conduct an experiment that uses the Groupon client's behaviour as reference to transfer variable amounts of backend data, ranging from 50 KB–1.5 MB—we maintain the same number of remote calls but vary their data sizes. We further assume the same ratio of used to unused backend data as reported in Section V-B.

Figure 8 plots the responsiveness of the original and the EDGEREDUCE versions of Groupon for different amounts of backend data returned, assuming minimum data reduction by disabling EDGEREDUCE's mechanism for tuning the prefetching of images. When the amount of backend data increases, the application response time for the original version increases more sharply than the EDGEREDUCE version. The overhead of additional remote calls between the ASP and the client is gradually masked by the savings in data transmission time over the slow network. Eventually, these savings are enough to achieve a speedup in execution.

We conclude that, for client applications that retrieve large amounts of backend data over a 3G network and exhibit a similar ratio of used to unused backend data, EDGEREDUCE can speed up execution. This is only the case when the gains of avoiding the transmission of data over the limited RAN outweigh the cost of the additional remote calls between the mobile device and the ASP.

D. Resource Overhead

To explore the resource requirements of an ASP, we monitor the CPU and memory utilisation of the ASP created for the Groupon client. Figure 9 shows the percentage of CPU usage when a user requests the list of most popular Groupon deals and views the top two results. While processing this request, CPU usage at the ASP increases to 11%: the ASP issues the HTTP request, parses the response data and returns the results to the client. The two spikes in CPU utilisation after 13 s and 19 s correspond to two subsequent requests for images.

The figure also shows the memory consumption at the ASP. Approximately 1 MB of memory is consumed initially by the baseline objects and data structures created at the ASP. During the processing of the initial request, an additional 1 MB of memory is allocated while processing the API response data. Overall, the CPU and memory utilisation at the ASP remain low, suggesting that multiple ASPs can be hosted as part of a virtualised base station.

VI. Related Work

This section compares EdgeReduce with prior work in the areas of automatic application partitioning, code offloading and distributed programming support.

A. Automatic Application Partitioning

Automatic application partitioning techniques aim to facilitate the development of distributed applications with good application performance. *J-Orchestra* [38] automatically partitions Java applications at the byte-code level. Similar to EdgeReduce, it distributes objects across machines using compiler techniques that substitute local with remote calls and data objects with indirect references. J-Orchestra offers a partitioning mechanism but no policy, relying on the user to specify the location of application classes. In contrast, EdgeReduce generates application-specific proxies specifically to reduce the network usage of mobile applications.

Coign [39] is a system for automatic distribution of applications built from COM software components. It constructs a graph of inter-component communication by profiling application execution and partitions applications across nodes in order to minimise communication. To realise a partitioning, Coign exploits the fact that COM components are location-transparent. In contrast, EdgeReduce only decides which application classes to place at a network proxy and cannot rely on a component model to realise a partitioning.

Wishbone [40] facilitates the deployment of sensor network applications by partitioning a graph of stream operators. It profiles the execution of operators against sample data and decides on a partitioning that minimises network bandwidth or CPU consumption. While Wishbone assumes arbitrary dataflow programs, EdgeReduce explicitly targets the dataflow in mobile client applications due to the MVC design pattern.

B. Code Offloading Systems

Code offloading systems partition mobile applications to offload resource-intensive functionality, primarily to reduce application response times. *Cloudlets* [41] and *Virtual Smartphone over IP* [42] treat smartphones as thin clients that are served by virtual device images on powerful remote servers. They mediate user input and screen updates from the device to the virtual device image and back, effectively offloading the entire application logic to a remote server. In doing so, however, they rely on a high-bandwidth network connectivity.

Systems such as *MAUI* [43], *Thinkair* [44], *COMET* [45] and *CloneCloud* [46] apply a more fine-grained partitioning with richer optimisation goals such as improving response time or reducing energy consumption. Mobile applications are partitioned at the function-level by converting local method calls into remote calls, or by migrating entire VM images from the device to a remote server.

In contrast with EdgeReduce, such approaches do not focus on reducing the network usage of mobile client applications. They rather assume that offloaded applications are side-effect free and do not interact with backend services.

C. Distributed Programming Support

The use of *futures* in distributed programming is an old concept that was first proposed for achieving concurrency in distributed applications [47], [48]. A future, or *promise*, is a reference to the result of a remote procedure call, which is still unevaluated. By deferring the acquisition of the result of a remote call to sometime in the future, a thread does not have to block execution until it attempts to evaluate the corresponding future, in which case it explicitly asks for the result.

EdgeReduce uses the idea of futures in a similar fashion but for a different reason. It allows for large binary Data Objects to remain at the ASP unless used by the client application. This helps reduce RAN traffic due to aggressive prefetching strategies that tend to retrieve large amounts of unused objects such as images.

VII. Conclusions

Mobile networks need to consider innovative techniques for handling the avalanche of data traffic. This paper makes the observation that mobile client applications receive unnecessary data from Internet backend services due to a semantic mismatch of the granularity of API calls and the unnecessary prefetching of data.

We propose EdgeReduce, an approach for generating application-specific proxies (ASPs) for mobile client applications that blurs the boundary between mobile end-systems and networks. ASPs host the application logic that receives response data from an Internet backend service and converts it into application objects, which are then sent to the mobile device. As a result, discarded data from the backend service is never transmitted over the RAN. To reduce network usage, we describe optimisations that allow ASPs to minimise the number of remote calls to the client and to retrieve large binary objects on-demand. Our evaluation shows the potential of EdgeReduce as a practical approach to reduce data traffic in mobile networks for today's mobile applications.

References

[1] Cisco, *Global Mobile Data Traffic Forecast Update*, 2014, http://www.cisco.com/c/en/us/solutions/collateral/service-provider/visual-networking-index-vni/white_paper_c11-520862.html.

[2] R. Agency, *Cellular and 3G Telephony*, http://ofcom.org.uk/static/archive/ra/topics/pmc/document/licencetypes/cellularinfo.htm.

[3] Communications Network Research Institute (CNRI), *Mobile Data Offload*, 2010, http://cnri.dit.ie/research.data_offload.html.

[4] Twitter, *Twitter REST API*, 2013, https://dev.twitter.com/overview/documentation.

[5] Amazon, *Amazon MWS*, 2013, https://developer.amazonservices.com.

[6] B. D. Higgins, J. Flinn, T. J. Giuli, B. Noble, C. Peplin, and D. Watson, "Informed Mobile Prefetching," in *MobiSys*, 2012.

[7] T. C. Projects, *SPDY: An Experimental Protocol for a Faster Web*, 2013, http://www.chromium.org/spdy/spdy-whitepaper.

[8] J. Roskind, *QUIC: Design Document and Specification Rational*, 2013, https://docs.google.com/document/d/1RNHkx_VvKWyWg6Lr8SZ-saqsQx7rFV-ev2jRFUoVD34/edit?pli=1.

[9] J. Erman, A. Gerber, M. Hajiaghayi, D. Pei, S. Sen, and O. Spatscheck, "To Cache or Not to Cache: The 3G Case," in *IEEE Internet Computing*, 2011.

[10] N. T. Spring and D. Wetherall, "A Protocol-Independent Technique for Eliminating Redundant Network Traffic," in *SIGCOMM*, 2000.

[11] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns*, 1996.

[12] S. Woo, E. Jeong, S. Park, J. Lee, S. Ihm, and K. Park, "Comparison of Caching Strategies in Modern Cellular Backhaul Networks," in *MobiSys*, 2013.

[13] G. Fleishman, *The State of 4G: It's all about Congestion, not Speed*, 2010, http://arstechnica.com/tech-policy/2010/03/faster-mobile-broadband-driven-by-congestion-not-speed/.

[14] V. Chandrasekhar, J. G. Andrews, and A. Gatherer, "Femtocell Networks: A Survey," in *IEEE Communications Magazine*, 2008.

[15] M. R. Ebling, L. B. Mummert, and D. C. Steere, "Overcoming the Network Bottleneck in Mobile Computing," in *Workshop on Mobile Computing Systems and Applications (WMCSA)*, 1994.

[16] J. Erman, A. Gerber, M. T. Hajiaghayi, D. Pei, and O. Spatscheck, "Network-aware Forward Caching," in *World Wide Web (WWW)*, 2009.

[17] F. Sailhan and V. Issarny, "Energy-Aware Web Caching for Mobile Terminals," in *ICDCS*, 2002.

[18] T. R. Fischer and Q. Chen, "Subband Image Coding for Packet Erasure Channels." in *ICIP*, 1996.

[19] A. Anand, A. Gupta, A. Akella, S. Seshan, and S. Shenker, "Packet Caches on Routers: The Implications of Universal Redundant Traffic Elimination," in *SIGCOMM*, 2008.

[20] A. Anand, C. Muthukrishnan, A. Akella, and R. Ramjee, "Redundancy in Network Traffic: Findings and Implications," in *SIGMETRICS*, 2009.

[21] C. Hetting, "Seamless Wi-Fi Oflload: From Vision to Reality," Aptilo Networks, Tech. Rep., 2013.

[22] S. Dimatteo, P. Hui, B. Han, and V. O. K. Li, "Cellular Traffic Offloading through WiFi Networks," in *Mobile Adhoc and Sensor Systems (MASS)*, 2011.

[23] A. Balasubramanian, R. Mahajan, and A. Venkataramani, "Augmenting Mobile 3G Using WiFi," in *MobiSys*, 2010.

[24] K. Lee, J. Lee, Y. Yi, I. Rhee, and S. Chong, "Mobile Data Offloading: How Much Can WiFi Deliver?" in *Co-NEXT*, 2010.

[25] R. T. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, 2000, Doctoral dissertation.

[26] Facebook, *Facebook Query Language (FQL) Reference*, 2013, https://developers.facebook.com/docs/reference/fql/.

[27] H. Shen, M. Kumar, S. K. Das, and Z. Wang, "Energy-Efficient Data Caching and Prefetching for Mobile Devices Based on Utility," in *Mobile Network Applications*, 2005.

[28] Apple, *MVCNetworking*, 2010, https://developer.apple.com/library/ios/samplecode/MVCNetworking/Introduction/Intro.html.

[29] IBM, *IBM ASPN*, 2013, http://www-03.ibm.com/press/us/en/pressrelease/40490.wss.

[30] ZeroC, *Internet Communications Engine (ICE)*, 2005, http://zeroc.com.

[31] A. Bradley, *Automating Image Compression And Optimization*, 2013, http://www.resrc.it.

[32] CocoaDev, *MethodSwizzling*, 2013, http://cocoadev.com/MethodSwizzling.

[33] T. Mori, *NatsuLion Twitter client*, 2009, https://github.com/takuma104/ntlniph.

[34] V. Aranha, *Groupon client*, 2011, https://github.com/vivianaranha/Groupon-API---iOS.

[35] Groupon, *Groupon API*, 2013, http://www.groupon.com/pages/api.

[36] W. Lyon, *Yahoo! Finance client*, 2013, https://github.com/johnymontana/WillzPlotz_iOS.

[37] Y. Finance, *Yahoo! Finance API*, 2013, https://code.google.com/p/yahoo-finance-managed/wiki/YahooFinanceAPIs.

[38] E. Tilevich and Y. Smaragdakis, "J-Orchestra: Automatic Java Application Partitioning," in *ECOOP*, 2002.

[39] G. C. Hunt and M. L. Scott, "The Coign Automatic Distributed Partitioning System," in *OSDI*, 1999.

[40] R. Newton, S. Toledo, L. Girod, H. Balakrishnan, and S. Madden, "Wishbone: Profile-based Partitioning for Sensornet Applications," in *NSDI*, 2009.

[41] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies, "The Case for VM-Based Cloudlets in Mobile Computing," in *IEEE Pervasive Computing*, 2009.

[42] E. Y. Chen and M. Itoh, "Virtual Smartphone Over IP," in *WoWMoM*, 2010.

[43] 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 *MobiSys*, 2010.

[44] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "ThinkAir: Dynamic Resource Allocation and Parallel Execution in the Cloud for Mobile Code Offloading," in *INFOCOM*, 2012.

[45] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen, "COMET: Code Offload by Migrating Execution Transparently," in *OSDI*, 2012.

[46] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution between Mobile device and Cloud," in *EuroSys*, 2011.

[47] A. Chatterjee, "Futures: A Mechanism for Concurrency Among Objects," in *Supercomputing*, 1989.

[48] B. Liskov and L. Shrira, "Promises: Linguistic Support for Efficient Asynchronous Procedure Calls in Distributed Systems," in *SIGPLAN*, 1988.