NetMob: A Mobile Application Development Framework with Enhanced Large Objects Access for Mobile Cloud Storage Service

Yunus Parvej Faniband[1], Iskandar Ishak[2], Fatimah Sidi[3], Marzanah A. Jabar[4]
Faculty of Computer Science and Information Technology
Universiti Putra Malaysia
43400 UPM, Serdang, Selangor Darul Ehsan, Malaysia

***Abstract*—Mobile enterprise applications are primarily developed for existing backend enterprise systems and some usage scenarios require storing of large files on the mobile devices.These files range from large PDFs to media files in various formats (MPEG videos).These files needs to be used offline , sometimes updated and shared among users. Present work studied different Mobile Backend as a service (M)BaaS platforms to understand techniques use to handle large file and found that many are either missing the feature or does not handle performance issues for large files. In this paper, we are proposing, NetMob, a mobile synchronization platform that allows resource-limited mobile devices to access large objects from the cloud. This framework is mainly focused on large file handling and has support for both table and objects data models that can be tuned for three consistency semantics, resembling strong, causal and eventual consistency. Experimental results conducted using representative workloads showed that NetMob can handle large files access with the size ranging from 100MB upto 1GB and is able to reduce sync time with object chunking in our experiment settings.**

***Keywords*—Mobile cloud computing; data consistency; mobile back-end as a service; mobile apps; distributed systems**

I. Introduction

In general, mobile cloud computing architecture has two unique set of entities namely Fixed Hosts (FHs) and Mobile hosts (MHs) [1]. FHs are machines (Works stations and Servers) with efficient computation power and reliable storage of data and run large databases. FHs that are connected through fixed network. MHs with limited processing and storage power (cellular phone, palmtops, laptops, notebooks) are not continually communicating with the fixed network. They may be disconnected for various reasons.

Additional dedicated fixed hosts called mobile support stations (MSSs) acts as the channel between the FH and MH through wireless LAN (local area network) connections, cells or connections to the network with standard modems.

When the network connectivity becomes unavailable or unacceptable, the MH enters the disconnected state. Disconnected operation (see Fig. 1) is a three-stage changeover between the following states [2].

1) Data hoarding : This is the process of preloading or prefetching the data in anticipation of a foreseeable disconnection. Before going to offline mode (disconnection), the data structures necessary for operation during disconnection are either replicated (catched) or moved (partitioned) at the MH.
2) Disconnected operation: When the MH is offline (disconnected from the network), data might be changed, added or even removed at either the MH or the FH.
3) Synchronization or Reintegration : When the connection is reestablished, each operation executed at the MH should be synchronized (reintegrated) with appropriate updates executed at other sites in order to attain seamless consistency.

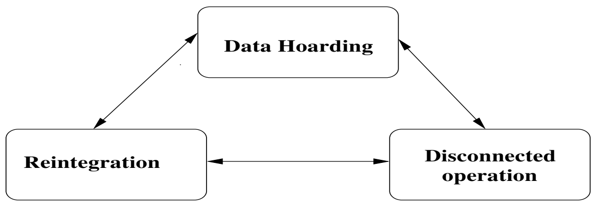

Fig. 1. States of Disconnected Operation.

For a given distributed system, the complexity of operations in each of above the three states is determined by the interdependence of data operated on. The execution of distributed applications in local-area networks is significantly different than in wireless, mobile systems. Wireless applications must use different communication pattern in order to address the high latency, low bandwidth, intermittent connections and communication charges based on time and content.An application operating on a LAN can manage good user interactions in case queries to a non-local database, but the same application operating on a wireless network may become unresponsive due to the delay in response. Hence wireless applications chose data replication, explicit or implicit (caching or data hoarding), as the primary technique to address the Disconnected operation.

The introduction of multi-user and collaborative features for wireless application increase the complexity, as multiple users have to share data objects and thus communicate and collaborate with each other [3]. In such cases there must be a sophisticated coordination mechanisms other than the conventional mechanism of locks. Thus addressing the wireless mobile systems constraints in the application development becomes challenging for developers, since they have to retain favorable user interaction and performance along with tackling the data coordination issues.

Mobile services can be developed and deployed in various cloud computing scenarios. The main service models of cloud computing are [4] Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). With the advent of new service model, Backend as a Service (BaaS), sometimes also referred as Mobile Backend as a Service (MBaaS), the native mobile applications can be easily integrated with the cloud. MBaaS frameworks should:

- Facilitate non blocking, responsive (ensure high availability) and reliable mobile applications during disconnection.
- Support Cloud-connected multi-user, shared-data mobile apps that require to handle the inter-dependent data locally,and also across multiple devices with cloud storage.
- Provide a synchronization model with tunable consistency guarantees so that developers have the flexibility to configure how data is synchronized and data conflict are handled.
- Provide a synchronization-aware high-level APIs that support applications for on demand and background synchronization tasks.
- Enable support for large files (i.e. a couple of megabytes or gigabytes) synchronization.
- Require to be efficient in power consumption and bandwidth usage for mobile clients and carry out efficient periodic/configured sync operations.

Each MBaaS system offers a distinctive set of functionalities through APIs (REST or wrapper libraries of the APIs) and allows programs to be written specially to execute in the cloud. Amazon Mobile SDKs provide the means to interact with cloud services through REST APIs. Multi-platform SDKs (iOS, Android, Fire OS, and Unity) are offered to interact with the AWS services, including S3 (storage), DynamoDB (database), Simple Notification Service (SNS) and Mobile Analytics [5] . Apple provide iCloud service (CloudKit SDK) to store and access data in iCloud [6]. Mobile applications are broadly classified into two types such as offline applications and online applications [7]. Unlike online apps,in offline (native) application, the mobile device and back-end system are not connected always. In order to support continuous mobile services, offline applications will process the presentation and business logic with the available local data on the device itself. Periodically data is updated by synchronizing with back-end systems.

Recently a large number of research efforts have been conducted on enterprise cloud storage services and personal cloud storage services. The investigations from [8] attempted to find out mobile user access behavior in a large-scale mobile cloud storage with a a dataset of 350 million HTTP request logs. The study observed the trend of using the cloud storage for large file sharing, with the average volume as large as about 70 MB, in multiple sessions for retrieving one file.

Another study from a cloud storage service provider (Filestack [9]) analyzed a dataset of 100,000 applications.They provided the services of handling file uploads, transformations, storage, and delivery.Their observation targeted the statistics of uploaded trend of files ranging from different sizes and formats from the year 2015 to 2016 as shown in Fig. 2.Their analysis concluded that all file increased 50% year over year, but files sized 100MB and above increased over 170% year over year. This leads to the conclusion that file sizes are getting larger and mobile users access or share large size of files (above 100MB).

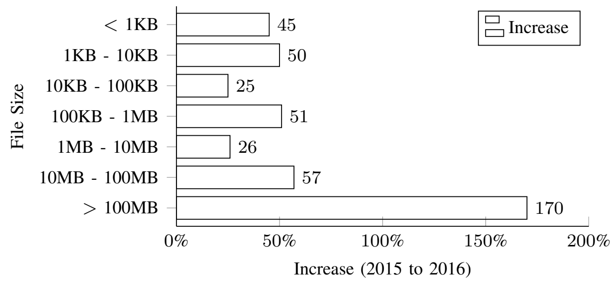

Fig. 2. Increase in files uploaded by File size

Handling the task of uploading and retrieving large files from and to a mobile app is a cumbersome process for developers due to issues of latency, speed, timeouts and interruptions. With the growing prevalence of sharing file of larger sizes among mobile users, providing reliable and efficient synchronization service for large files has become an important feature.

The rest of this paper is organized as follows: Section II discuss about the related work with details of support for large files upload and retrieval in mobile data synchronization frameworks with cloud storage services. Section III describes in detail the proposed framework, NetMob, a cloud based framework to support End-to-end data consistency for large data object access. Section III deal with architecture and design in detail along with details of NetMob data model and supported APIs. Section IV deals with the technique of handling the large objects with Segmentation and Object Chunking for object storage in Open Stack Swift [10]. While the Section V describes the NetMob handling of large object support with Segmentation and Object Chunking both at the client and server side, Section VI illustrate NetMob support for the consistency schemes with Cassandra [11] and Open Stack Swift [10]. Section VII discuss about the NetMob implementation followed by Evaluation (Section VIII & Section IX) with comparison of NetMob with Dropbox, and Conclusion and future work (Section X).

II. Related Work

Supporting large file upload and retrieval is crucial for the mobile cloud storage services, as file sizes are trending larger and mobile users access or share files of large size [8], [9]. Practical large object services are, however, only available for PC clients and not for mobile apps. To understand the support for large file objects, we analyzed both the commercial and open source cloud storage services for mobile. Table IV summarizes the large objects support and limitations in the different reference implementations. Even though the commercial cloud frameworks provide support for large objects, many frameworks do not handle large files.

The key observation from the study is that many of the systems do not support large objects and some have limitations.

Simba's [12] sync protocol does not support streaming APIs to handle big size objects (e.g. Media file like Videos). SwiftCloud [13] is a middleware system that implements a Key-CRDT on top of Riak [14]. The Riak designers do not recommend storing objects over 50MB for performance reasons. Izzy [15] is an initial version of Simba and do not support large objects.

Mobius [16] does not handle large files but addresses the messaging and data management challenges of mobile application development. Special CRDT cloud types data in TouchDevelop [17], [18] do not address large size.

Open Data Kit 2.0 [19] which is an Android based service have a 1 MB size limit on remote-procedure calls.To address 1MB limit, ODK Kit implements a primitive transport-level chunking interface using a client-side proxy to bring together the chunks and only reveal a higher-level abstraction to the tools.

QuickSync [20] framework is built using Dropbox and Seafile APIs that supports large data size up to 180MB. The *chunked_upload* API supports uploading of larger files in multiple chunks. It supports the interrupted uploaded to be resumed later with chunk of any size up to 150 MB with a default size of 4 MB.

The Parse Server [21] only supports files up to to 10MB. The ParseFile data type allows the app to store application files in the cloud in addition to a smaller data structure of ParseObject. PareObject allows upto to 10MB data in bytes array or in the Stream form and SaveAsync call saves the file to Parse.

BaasBox [22] which is an open source MBaaS framework does not support large files. It is based on Play framework which is a lightweight, stateless, web-friendly architecture. In order to support large files, the REST APIs in Play framework can be configured for the maximum payload size in POST operations. Body parsers in Play framework is a HTTP request (at least for those using the POST and PUT operations) that contains a body. The default size of POST request is 100KB and can be configured according to the server configuration.

Dropbox [23] REST APIs supports large files up to 150MB. The *files_put* API has a maximum file size limit of 150 MB and does not support uploads with chunked encoding. The *chunked_upload* API support uploading of larger files in multiple chunks. Chunks can be of any size up to 150 MB with a default size of 4 MB. Dropbox supports resuming uploads if interrupted due to network disconnections.

Google Drive [24] APIs supports resumable uploads for files more than 5MB, with a single request or in multiple chunks.The PUT request allows chunks in multiples of 256 KB (256 x 1024 bytes) in size, except for the final chunk that completes the upload. Chunks size has to be kept as large as possible so that the upload is efficient.

Amazon Dynamo [25] provides high availability allowing updates even during the network partitions or server failures and targets applications that require only key/value access. Amazon DynamoDB enforce a maximum item size of 400KB in a table, including both attribute name binary length and attribute value lengths. If the application needs to store more data in an item than the DynamoDB size limit permits, the app can try compressing one or more large attributes, or it can store them as an object in Amazon Simple Storage Service (Amazon S3) and store the object identifier of S3 in the DynamoDB item.

The documentation of iCloud supported by CloudKit [6] neither specify a Document file size limit, nor a Core Data (iOS local) storage limit, other than a user account icloud storage allowance. But the uploads are dependent on the storage limit of device/user iCloud account. When the app adopt iCloud document management lifecycle, the operating system (iOS) initiates and manages uploading and downloading of data for the devices attached to an iCloud account. The app does not directly communicate with iCloud servers and, in most cases, does not invoke upload or download of data.

The commercial framework Kinvey [26] supports to store and retrieve binary files of size up to 5TB with the help of third-party service. Kinvey currently use Google Cloud Storage, as a third-party service to provide short-lived links, that can be used to upload or download files.

Kony [27] is another platform that supports Large Binary Objects API to retrieve and delete large binary objects, schedule a download, and get the location of the objects. While the Sync Chunking Mechanism applies to all of sync, the Large Binary Objects API supports the download of binary data stored in a particular object in multiple chunks. The download occurs in the background, allowing the user to perform tasks simultaneously.Kony applies Byte Range Serving technique, where a client can request a specific portion of Large Binary file that is present on the backend. This technique efficiently uses the network bandwidth by allowing user to download the binary in chunks ranging over multiple requests to the server.

Prior work from the authors of this paper [28] presented a review of data consistency and synchronization frameworks in Mobile Cloud Computing for Mobile Apps. This work was focused on client-centric data consistency and the offline data synchronization feature of various frameworks. While previous work from the same authors covers results from the selected studies in areas such as data consistency, handling offline data, data replication, synchronization strategy, this paper deals with only large file handling support.

III. NetMob

To meet mobile application development requirements and with main purpose of supporting large files (i.e. hundreds of megabytes or gigabytes or more), we have developed NetMob. NetMob is an applications framework, implemented in C# and centered around the main aspect of providing the support for large files (from hundreds of MBs up to 5GBs) in mobile cloud services and enable the programmers to create arbitrarily complex, synchronized replicated large data objects.

A. NetMob Architecture and Design

NetMob architecture consists of mainly two modules. One Client software executes on the mobile device and the other

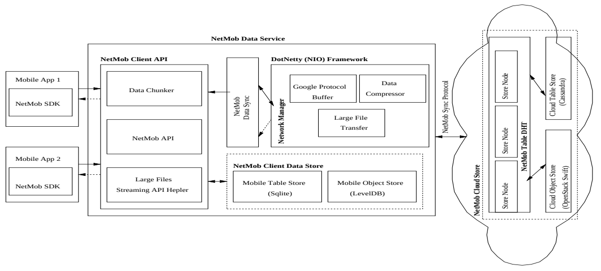

Fig. 3. Architecture of NetMob framework.

server for data storage in the cloud. The combination of these two software assists the development of mobile application on the device according to the NetMob SDK. Fig. 3 shows the basic architecture of NetMob.

The NetMob System Data Control Service (Nm-SDCS) is the client software which acts as a interface of device-to-cloud communication, for the mobile apps and responsible for exchanging data and messages with the with NetMob Cloud Data Server through a custom sync protocol. Each NetMob app communicates with the system-wide service of NetMob System Data Control Service, through the NetMob Data API (streaming and CRUD) provided by the NetMob SDK.Nm-SDCS service also consists of NetMob Local database store (Nm-LDBS) to save all the application data and metadata inside tables or object store.The local database store (Nm-LDBS) is managed by Nm-SDCS and is not directly accessible to NetMob apps.

The cloud server consists of NetMob Cloud Data Server (Nm-CDS) which store the data and interact with the data control service via custom sync protocol.

To reduce network footprint,the Network Data Manager (NMD) helps to transmit network data (from multiple rows, across multiple objects/tables) and messages from multiple apps with data compression and support for large file transfer.

The key design goals supported by NetMob architecture are as follows:

1) NetMOB SDK provide programming model familiar to mobile app developers with Data API consisting of CRUD and sync systematic along with unique support for streaming access to large objects.The complexities of disconnected operations, data hoarding, conflict detection and push notifications are hidden behind this interface.
2) NetMob provide a high level abstraction for building a fault-tolerant apps and assist apps to programatically handle delay-tolerant data transfer between the mobile device and the cloud.A non-zero period value of delay tolerance (DT) can be set which determines the frequent of change collection.
3) NetMob SDK provides apps with data granularity of both tables and object data with a feature to specify the distributed consistency for the data.
4) NetMob utilize efficient data reduction and bandwidth reduction techniques to minimize both the number of messages and bytes transferred over the network and hence support efficient device battery usage.

B. Data Model

NetMob's data model simplifies data storage for apps to store all of their data and hiding the details of how data is stored and synced.NetMob offers a data model called NetMob Table (Table$_{NM}$ for short) supporting both tables and objects. An individual row of an Table$_{NM}$, called a Row$_{NM}$. Each Row$_{NM}$ can accommodate associated tabular and object data with the tunable distributed consistency for the table as a whole from available consistency options. Hence NetMob permit consistency specification per table and treat a row as the unit of atomicity preservation. Same consistency is applicable to all tabular and object object data in Table$_{NM}$. NetMob ensure that all app and user data stored in the Table$_{NM}$ and provide synchronization service with the cloud and on to other mobile devices.

NetMob can also support apps that need either a tabular-only or object-only schema.NetMob also frees the developer from writing complicated transaction management and recovery code by utilizing the Row$_{NM}$, which offers a programmable higher-level interface,for a unit of app data with consistency guarantees under all scenarios.

C. API Interface

The design of NetMob API is similar to the well-known CRUD interface and enable the apps to set the Table/Object properties, access their data and push new data and perform conflict resolution.NetMob a stream abstraction that allows the objects to be written to, or read from, which is very suitable to handle large object. NetMob also support local reading or writing only a part of the large object , which is not supported by typical BLOBs (binary large objects) in relational databases [38]. Any app written adhering to this API interface is considered as a NetMob-app.

API, described in Table I, provide following features:

1) CRUD (Create, Read, Update, and Delete) functionality on both tables and objects.
2) Methods to subscribe tables for synchronization.
3) Notifications for new data and conflicts.
4) Support for conflict detection and facility for resolution.

D. NetMob Design

1) NetMob Client (Client$_{nm}$): Client$_{nm}$ allow the networked NetMob-apps to have I/O data model even during the disconnected operations and enable partition tolerance. Client$_{nm}$ allows seamless data access for the apps by hiding the them from server and network disconnections. It is designed to run as static instance of device-wide service with following functionalities:

1) Provides seamless access to table and object data for all NetMob-apps through a well defined lightweight Interface (sClientLib).
2) Support large objects with stream abstraction to read and write objects.
3) Maintain appropriate local replication of data on the mobile device to enable disconnected operations.

TABLE I. **NetMob APIs.**

Type	API	Purpose
CRUD Operations on tables and objects	PutObject[]	Creating Objects
	GetObject[]	Retrieving Objects
	PutObject[]	Updating Objects
	DeleteObject[]	Deleting Objects
CRUD Operations with Chunking	PutObjectChunk[]	Creating chunk
	GetObjectRange[]	Retrieving chunk
	DeleteObjectChunk[]	Deleting chunk
	DeleteChunks(table, selection)	Deleting multiple chunks
Large Object Handling	PutManifest[]	Creating large objects using Chunk
	GetManifest[]	Retrieving large objects using Chunk
	DeleteObjects[]	Deleting large objects
Object Synchronization	writeSyncSubscribe(table, period, delayTolerance, syncprefs)	Register for sync notifications
	writeSyncUnSubscribe(table)	Unregister for sync notifications
	instantWriteSync(table)	Invoke instant Write Sync
	readSyncSubscribe(table, period, delayTolerance, syncprefs)	unregisterReadSync(table)
	instantReadSync(table)	Invoke instant Read Sync
Notification APIs	dataAvailableFresh(table, numRows)	Notification for new data availability
	conflictData(table, numConflictRows)	Notification for conflict in data
Conflict Resolution	beginCROperation(table)	Start Conflict resolution
	getConflictedDetails(table)	Get conflicted Details
	resolveConflict(table, row, choice)	Resolve Conflict
	endCROperation(table)	End Conflict resolution

4) Guarantee fault-tolerance, high availability, data consistency, and atomicity at row-level.
5) Execute all synchronization tasks over the network.
6) Provide notifications to the apps for events like new data, conflict.
7) Monitors liveness of apps, and memory management in case of app crashes.

2) NetMob Local Data Store ($LDBS_{NM}$) : The NetMob Local Data Store ($LDBS_{NM}$) act as a local persistent storage module to save both tabular data and objects of app in the mobile device's memory (typically the internal flash memory or the external SD card). The primary responsibility of ($LDBS_{NM}$) is to support atomic updates over the local data and allow efficient CRUD operations support (on $Rows_{NM}$). It also support atomic sync of variable sized and possibly large objects. It must respond quickly to the change detection queries and inform about the sub object changes in the stored local data. The data layout of $LDBS_{NM}$ is shown in Fig. 4. The logical structure of the table and object storage is also depicted in Fig. 4.

The primary goal design of $LDBS_{NM}$ is to enable storage of large objects by dividing local data into fixed-size chunks and store in a key–value store (KVS) that supports range queries. LevelDB [29], a KVS based on a log-structured merge (LSM) tree [30] is chosen, that has a good throughput for both appends and overwrites.

3) NetMob Cloud Server ($Cloud_{NM}$) : The primary responsibility of NetMob Cloud Server (called $Cloud_{NM}$) is to manage data across multiple $Client_{NM}$, $Table_{NM}$, and NetMob-apps. $Cloud_{NM}$ facilitate tunable consistency storage mechanism with three different consistency plans and synchronization for $Table_{NM}$.

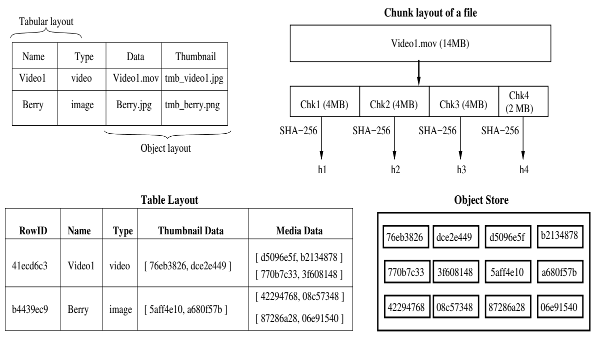

Fig. 4. NetMob Local Data Store.

$Cloud_{NM}$ is divided into two modules,client-facing Gateway and a data store, NetMob Cloud Store (for short, $Store_{NM}$), based on independently scalable client management and data storage, respectively. For data scalability,store is organized into store nodes. At-most one Store node is assigned to each $Table_{NM}$ to manage both its tabular and object data.In order to ensure read-my-writes consistency [31] with scalability, the data of $Table_{NM}$ is saved in two separate stores, each for tabular and object data. The store nodes server the client requests by serializing the synchronization operations and support three different consistency plans on each table at the server.

The Gateway acts as an interface for the communication between $Client_{NM}$ and the $Cloud_{NM}$. The load balancer assign a Gateway for the requested clients and handles authentication of client through an authenticator. The Gateway handle the table subscriptions, sending notifications, communication of the clients and transfer sync data between $Clients_{NM}$ and $Store_{NM}$. Since the Gateway is subscribed to change notifications for all $Store_{NM}$ nodes and eventually gets notified on changes to a subscribed $Table_{NM}$.

E. Sync Protocol

The design goal of $Cloud_{NM}$ is to communicate with the clients both for storage as well as data synchronization.Hence it interacts with the $Client_{NM}$ in the terms of change-sets.NetMob synchronization protocol is built on Netty [32] framework that support better throughput and lower latency. Netty has protocol support for transferring large files using zlib/gzip compression and data transfer through Google Protobuf [33].

Any client that require to communicate with each table of interest , needs to register with the server by subscribing to a write and/or read subscription. The design of sync protocol handle multiple independent writers using versioning to provide multi-version concurrency control through $Cloud_{NM}$. $Cloud_{NM}$ use the technique of compact version numbers, instead of full version vector [34] in order to support StrongS and CausalS consistency schemes.

For efficient change identification, NetMob keeps a version number per row and assign a unique row identifier. NetMob follows the scheme is used in gossip protocols [35] to identify that rows that needs to be synchronized.With each update

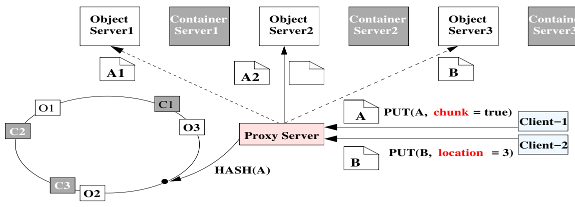

Fig. 5. NetMob cloud server Object Chunking in OpenStack Swift.

of the row,Row versions are incremented at the server. NetMob set the largest incremented row version maintained in a table, as the table version. Based on this table versioning scheme NetMob immediately determine which rows need to be synchronized and also select the list of Rows_{NM} which are modified, or the change-set.

For efficient network transfer, NetMob use the chunking methods and objects are stored and synced as a collection of fixed-size chunks.Client APIs expose chunking and apps continue to locally read/write objects as streams.

NetMob design is derived from previous work of unified table and object interface in the context of local systems [36] as well as in a networked app ([15], [12], [37] and [38]).

IV. Large Object Support with Segmentation and Object Chunking

The Chunking mechanism allows NetMob clients to transparently split objects into smaller parts, when uploading data to the object storage. In order to support upload of large objects, in Object Storage, the Open Stack Swift [10] use segmentation process. The process of Segmentation involves dividing the object and accordingly generating a file that delivers the segments together as a single object. Segmentation allows a virtually unlimited size of single object upload with faster and opportunity of parallel uploads of the segments.

OpenStack Object Storage allows single item of up to 5 GB in size for uploading. Openstack Swift follow segmentation process by fragmenting the object, and automatically creating a special manifest file that sends the segments concatenated as a single object. The technique of splitting objects into smaller chunks not only increase the efficiency and facilitate parallel uploads and also transparently support small, equal-sized chunks.

Fig. 5 illustrates the process of segmentation. During the upload, a client can request for object chunking and the proxy server split the arriving data into several blocks. These different blocks will be internally named based on the placement in the cluster. A special manifest file is generated with the ordered list of the names of all object blocks. During the GET request, the proxy sends the parts in order to the client, by reading the manifest file.

V. Large File Handling

The network manager implemented using the .Net DoTNetty uses zip (zlib/gzip) data compression with Google protobuf support and also provide support for large file transfer.

A. Large Object Support with Segmentation

In order to support upload of large objects, in Object Storage, the Open Stack Swift [10] use segmentation process. The process of Segmentation involves dividing the object and accordingly generating a file that delivers the segments together as a single object. Segmentation allows a virtually unlimited size of single object upload with faster and opportunity of parallel uploads of the segments.

Each object is stored as a single file on disk unless its size exceeds the maximum file size configured for the Swift cluster. This maximum file size defaults to a rather small value, 5 GB, in order to prevent a single object from filling up a disk while most of the cluster is empty. If an object to be stored is very large, it is divided into several segments and stored with a manifest to allow reassembly later.

B. Client Side Large Objects Handling in LevelDB with LSM

At the client side, NetMob integrate SQLite for table data and LevelDB for objects.NetMob embeds LevelDB at the mobile side for handling the large files.LevelDB use the concepts of Sorted String Table, Log Structured Merge (LSM) to support processing large workload where the input is in Gigabytes in size.

Sorted String Table (SSTable) [39] is popular data structure for efficiently storing large numbers of key-value pairs and support high throughput with sequential read/write workloads. SSTable can process and exchanging datasets where the input is in Gigabytes in size.

This Log Structured Merge (LSM) [30] architecture provides a number of interesting behaviors in combination with the SSTable. Since the LSM allow all writes go directly to the MemTable index, write operations are always fast regardless of the size of dataset (append-only). Also the random reads are either quickly retrieved from memory or served from disk (search MemTable initially and then the SSTable indexes. The MemTable is regularly flushed into disk as SSTable.

LevelDB architecture combines a set of processing conventions applied to SSTable and a MemTable to create a powerful embedded database engine. The other products that follow similar architecture are Google's Cassandra, BigTable, and Hadoop's HBase.

LevelDB supports fast write operations regardless of the size of dataset, as the all writes go directly to the log and a MemTable. The log is periodically flushed into disk as sorted string table files (SST) of size upto 2MB. Each SST file is internally split into single readable block of size 4K. These blocks are structured such that end block is an index that points to the beginning of each data block and it is the key of the entry at the beginning of the block. A Bloom filter accelerate the lookup process and facilitate fast search of an index to determine the block that may have the desired entry. LevelDB in addition minimize the read costs by partitioning SST into sets, or levels. Levels are numbered starting from 0, and each levels beyond Level 1 is 10 times the size of the previous level.

VI. Consistency levels in Cassandra and Swift

NetMob supports three most commonly used consistency schemes Strong, Causal and Eventual, identical to the ones in

Pileus [40] as illustrated in Table II.

TABLE II. CONSISTENCY SCHEMES IN NETMOB

Operation	Strong	Causal	Eventual
Offline write permission	No	Yes	Yes
Offline read permission	Yes	Yes	Yes
Need Conflict resolution	No	Yes	No

NetMob support a tunable Cloud Table store through Cassandra [11], [41] that allows consistency levels to be controlled from the client while performing an operation. Cassandra supports both the eventual and strong consistency models. In order to ensure high-availability, Cassandra is configured to make use of three-way replication. For supporting strong consistency during reading, Casandra will be configured for ReadConsistency=ONE, that indicates the immediate response from the closest replica. Similarly for supporting strong consistency during writing, Casandra will be configured for Write-Consistency=ALL. This configuration make sure that for all replica nodes in the cluster for that partition, a write operation is written not only to commit log and also to memtable.

Since OpenStack Swift supports Eventual consistency by default, in order to enforce the strong consistency, Store$_{NM}$ initially creates a new object and consequently deletes the old one after the updated Row$_{NM}$ is committed. Store$_{NM}$ attach a read/write lock to each Table$_{NM}$, in order to guarantee exclusive write access for update with additional concurrent access to multiple threads for reading.

VII. IMPLEMENTATION

A. Client

The prototype of Client$_{NM}$ is implemented as system wide service to provide data services to multiple apps with reduces network usage with compression techniques. Client$_{NM}$ is developed as a system wide service, which is connected by NetMob-apps through local remote procedure calls (RPC). Client$_{NM}$ integrate SQLite for table data and LevelDB for objects. Both data and push notifications between Client$_{NM}$ and Cloud$_{NM}$ are served with a single persistent TCP connection.This single TCP connection will avoid insignificant connection establishment and teardown [[42]] from NetMob apps. Client$_{NM}$ also implements the client APIs for OpenStack Swift that covers most of the Swift APIs, and handles authentication and large object streaming.

B. Server

Store$_{NM}$ prototype integrate Cassandra [41] to provide the tabular data support and utilize OpenStack Swift [10] APIs for Object storage.

Cassandra supports both the eventual and strong consistency models. In order to ensure high-availability, Cassandra is configured to make use of three-way replication. For supporting strong consistency during reading, Casandra will be configured for ReadConsistency=ONE, that indicates the immediate response from the closest replica. Similarly for supporting strong consistency during writing, Casandra will be configured for WriteConsistency=ALL. This configuration make sure that for all replica nodes in the cluster for that partition, a write operation is written not only to commit log and also to memtable.

Since OpenStack Swift supports Eventual consistency by default, in order to enforce the strong consistency, Store$_{NM}$ initially creates a new object and consequently deletes the old one after the updated Row$_{NM}$ is committed. Store$_{NM}$ attach a read/write lock to each Table$_{NM}$, in order to guarantee exclusive write access for update with additional concurrent access to multiple threads for reading.

VIII. EVALUATION

For application frameworks the latency is a benchmarking factor to study the effect of high-level abstraction for efficient sync of mobile application data [43], [44], [45], [46]. Based on the literature [47] consistency models suitable for mobile environments are classified based on three deviation metrics. This classification is termed as 3D Design Framework [48], [49]. Three metric parameters form the three axes of the 3D Design Framework:

1) Numerical deviation: difference in number of updates applied to replicas.
2) Order deviation: difference in order of operations between replicas.
3) Staleness: delay until replicas see an update.

The actual deviations may differ depending on the level of inconsistency tolerated by the system. The 3D Design Framework classification resulted after applying the metrics to various consistency models [48]. A suitable consistency model can be devised based on the acceptable amount of inconsistencies. The axes do not contain concrete values, as these depend on the system. The nature of the system also determines the acceptable amount of inconsistencies.

The empirical evaluation and comparison of the NetMob framework is performed in three main categories:

- The efficiency of NetMob Sync Protocol.
- The performance of Cloud$_{NM}$ Data API consisting of CRUD and chunking operations.
- Effect of consistency and latency.

A. Experimental Setup

The evaluation environment of NetMob included a set of virtual machines and mobile device client. A virtual machine (VM) setup with OpenStack Swift (Version 1.12.0.37) deployment with one proxy node and 4 storage nodes. The proxy node with Ubuntu 14.04 was equipped with Intel Xeon CPU and 4 GB RAM and storage nodes (Ubuntu 14.04) had Intel Xeon E5-2403 processors and 1 GB RAM. Another VM with Cassandra (Version 1.2.5) setup with Ubuntu 14.04 was equipped with Intel Xeon CPU and 2 GB RAM.The system was tested with Xiaomi RedMi 5 Plus Device, 4GB RAM running Android Oreo. Evaluation was done using a WPA-secured WiFi network, instead of 4G network.

TABLE III. NetMob Sync protocol overhead

# No of Rows	Size of object	Size of payload	Size of Message (% Overhead)	Size of Network Transfer (% Overhead)
1	None	1 B	101 B (99.009%)	132 B (99.242%)
	200 KiB	200 KiB	210.098 KiB (99.905%)	235 KiB (99.149%)
	100 MB	106.34 MB	106.59 MB (0.235%)	106.89 MB (0.515%)
10	None	100 B	2.41 KiB (96%)	694 B (85.6%)
	200 KiB	2048 MB	2210 MB (99%)	2510 MB (99%)
	100 MB	1034.34 MB	1035.85 MB (0.15%)	1037.65 MB (0.318%)

B. Sync Protocol Overhead

NetMob is designed with a primary requirement to support programmers with an interface that can assist in efficient sync of mobile application data, with special focus on large file objects.It must be ensured that NetMob must not add significant overhead during the sync process. NetMob promise an efficient sync protocol with limited overhead and is lightweight.

To test the efficiency of NetMob Sync protocol we measured the message size and network transfer size for varied payload sizes with combinations of tabular and object data. Table III depicts the cumulative sync protocol overhead with varied payload sizes. The empirical evaluation calculate overhead for a single synchronization request comprising of a single row and a group of 10 rows with different sizes of payload.

The test results revealed NetMob produces an overhead of 100 bytes for a baseline message. This request is composed of no object data, but a single row consisting of tabular data of 1 byte. However it is observed that due to data compression in NetMob, the overhead in the case of per-row baseline request is reduced by 76%, for batch operations of 10 rows into a single sync request. Moreover, the data transfer overhead eventually becomes negligible when the payload (tabular or object) size increases. This shows that NetMob sync protocol is lightweight and efficient especially with group/batched row operations.

C. Performance of NetMob Data Retrieval and Chunking APIs

The NetMob API interface was tested by the mobile client application to issue requests of writing (uploading), reading (downloading) and deleting large files of various size. The ability of NetMob to support large files through fixed-size object chunking was also evaluated through test cases that configure the chunk size while calling the NetMob APIs.

1) Create-Read-Delete API performance:
NetMob framework is tested for both combination and individual tests of create-update-delete queries for data. For the write benchmark, with files of varying sizes from 1MB to 1GB. File read tests are conducted to analyze the ability of NetMob to handle large data from 1MB to 1GB. Tests for deleting individual files are also carried out. The create-read tests are initially carried out with a default chunk size of 8MB. Fig. 6 (A) depicts the results of create-read-delete APIs of NetMob.

The graph shows a gradual variation of the latency with respect to increase in the size of the file. Also the upload APIs took more time than the download APIs since the time spent is acknowledgment and processing data during the upload operation. The upload time variation was less for the upload

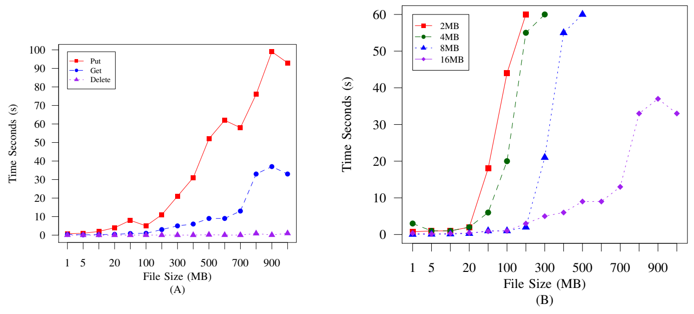

Fig. 6. (A)- Latency for Put, Get and Delete queries and (B) Put query latency for different chunk sizes (2, 4, 8 and 16 MB) in NetMob

file operations of size greater then 500MB, since the NetMob sync protocol effectively archive the data during network data transfer.

Delete operation for NetMob Clients is relatively fast, since lazy deletion marks objects as deleted. Overall, experimental evaluation of NetMob Data APIs showed that NetMob is efficient in handling the writing (uploading), reading (downloading) and deleting of large files.

2) Effectiveness of Object Chunking:
Next, we evaluated the performance of object chunking. To modify the object-to-node ratio, we use files of different sizes and change the object chunk size. A large file size produces a low number of objects with large chunk size and hence, a low object-to- node ratio or read-writes are faster. Fig. 6 (B) depicts NetMob performance of upload (Put query) operation with varying Object Chunk size.

Large chunk size proved to be more efficient as per the graph, since the large chunks produce fewer number of partitions of files hence can transmit faster. The authors recommend a chunk size of 16MB for NetMob for efficient large file handling, depending on the processing memory available in the mobile device.

D. Consistency vs. Performance

Cassandra supports both the eventual and strong consistency models. In order to ensure high-availability, Cassandra is configured to make use of three-way replication. For supporting strong consistency during reading, Casandra will be configured for ReadConsistency=ONE, that indicates the immediate response from the closest replica. Similarly for supporting strong consistency during writing, Casandra will be configured for WriteConsistency=ALL. This configuration make sure that all the writes must be written to both commit log and memtable on all replica nodes in the cluster, for that partition under consideration.

Since OpenStack Swift supports Eventual consistency by default, in order to enforce the strong consistency, Store$_{NM}$ initially creates a new object and consequently deletes the old one after the updated Row$_{NM}$ is committed. Store$_{NM}$ attach a read/write lock to each Table$_{NM}$, in order to guarantee

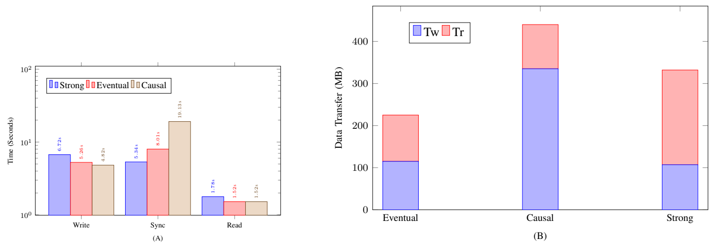

Fig. 7. (A) End-to-end latency for 100MB object and (B) Data transfer in different consistency schemes in NetMob

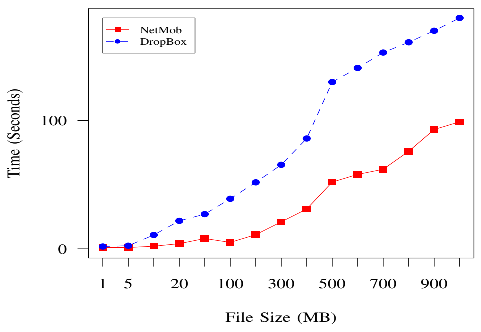

Fig. 8. **NetMob comparison with Dropbox framework for Upload data.**

exclusive write access for update with additional concurrent access to multiple threads for reading.

To measure the consistency parameters, the NetMob is tested with three types of mobile test clients Reader (Tr), Writer (Tw) and CausalTester (Tc). Prior to Tw's write operation, the client Tc have to make sure that it writes to a row which has the same row-key as Tw. This setup is tested with two Xiaomi RedMi 5 Plus Device, 4GB RAM running Android Oreo.The write payload is a 100MB file to measure the latency for reads, writes that is perceived by the NetMob client applications and latency of data sync with Cloud_{NM}. For evaluating the eventual and Causal consistency we used the notification time of one second. It must be ensured that all updates must happen prior to this time window.

Wi-Fi latency and associated data transfer are depicted in Fig. 7(A) and Fig. 7(B) respectively.

The "Write" latency is the app perceived latency of update at Writer (Tw). The sync-update latency from Writer (Tw) to Reader (Tr) is referred as "Sync". The "Read" latency is referred to the time taken by the app for reading updated data at Reader (Tr). Fig. 7(B) shows the total data transferred by Writer (Tw) and Reader (Tr) clients and for each consistency scheme.

Since $\text{Eventual}_{\text{S}}$ consistency requires to read only the latest version, it turns out into a single sync process resulting in less data transfer according to the rule of last write. Strong_{S} consistency require all updates to propagate immediately, there is a higher data transfer, but has lowest sync latency as data is synced immediately.In case of conflicts,Sync latency for Causal_{S} is greater than $\text{Eventual}_{\text{S}}$, since the former requires more RTTs to resolve conflicts. Conflict resolution With CausalS increase the data transfer amount, since for the first sync attempt by Writer (Tw) fails, so Writer (Tw) must read CausalTester's (Tc) conflicting data, and retry its update. Sync latency and data transfer for Causal_{S} and $\text{Eventual}_{\text{S}}$ are similar in case of absence of conflicts (not depicted).

IX. Performance Comparison with Other Sync Framework

The NetMob local performance is compared with the popular Data Syncing product Dropbox with chunk size of 16MB chunking. The test application invoke the writes, reads, and deletes for data containing one 100MB object for Dropbox (Core API) and NetMobClient. Fig. 8 shows standard deviation and average times and over three trials for the upload data (Put query). The Dropbox performance is affected by the location of the Dropbox server and the upload bandwidth at the time of testing. NetMob performs 15% faster than the Dropbox for the writes. The reason for the low performance of Dropbox may be due to server location (US) and the upload bandwidth at the time of testing. Delete performance for NetMob Client and Dropbox is almost same, since lazy deletion marks objects as deleted and but physically deleted only after sync operation is completed.

X. Conclusion and Future Work

Handling the task of uploading/retrieving large files from/to a mobile app is a cumbersome process for developers due to issues of latency, speed, timeouts and interruptions. In this paper, we investigated the large objects (from hundreds of MBs up-to 5GBs) support and limitations in the different reference implementations for cloud storage services for mobile. To address the large files, we propose NetMob, a framework with tunable chunking support and for large objects both at the mobile and the cloud storage. This work further contributed by implementing NetMob framework, that allows the large objects to be written to, or read from the cloud storage and also support local reading or writing only for a part of the large object, with tunable consistency option. The extensive evaluations demonstrate that NetMob can effectively store and sync large files with the reduced synchronization time and minimize significant traffic overhead for representative large file workloads. As a future work we would like to support additional consistency schemes (like Sequential consistency and others) in NetMob framework.

Acknowledgment

Authors thank the Faculty of Computer Science and Information Technology, Universiti Putra Malaysia for its financial support.

References

[1] E. Pitoura and G. Samaras, *Data management for mobile computing*. Springer Science & Business Media, 2012, vol. 10.

[2] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the coda file system," *ACM Transactions on Computer Systems (TOCS)*, vol. 10, no. 1, pp. 3–25, 1992, doi:https://doi.org/10.1145/146941.146942.

[3] J. P. Munson and P. Dewan, “Sync: a java framework for mobile collaborative applications,” *Computer*, vol. 30, no. 6, pp. 59–66, 1997.

[4] P. Mell and T. Grance, “The nist definition of cloud computing recommendations of the national institute of standards and technology,” *Nist Special Publication*, vol. 145, p. 7, 2011.

[5] A. W. S. Mobile, “Aws sdk,” 2016, https://aws.amazon.com/mobile/.

[6] A. Inc, “icloud for developers,” 2016, ”http://developer.apple.com/icloud”.

[7] H. Wu, L. Hamdi, and N. Mahe, “TANGO: A flexible mobility-enabled architecture for online and offline mobile enterprise applications,” *Proceedings - IEEE International Conference on Mobile Data Management*, pp. 230–238, 2010.

[8] Z. Li, X. Wang, N. Huang, M. A. Kaafar, Z. Li, J. Zhou, G. Xie, and P. Steenkiste, “An empirical analysis of a large-scale mobile cloud storage service,” in *Proceedings of the 2016 Internet Measurement Conference*. ACM, 2016, pp. 287–301.

[9] F. Shanon Montelongo, “How to upload large files,” https://blog.filestack.com/thoughts-and-knowledge/how-to-upload-large-files/.

[10] “Openstack swift object storage service,” 2018, http.//swift.openstack.org.

[11] A. Lakshman and P. Malik, “Cassandra: structured storage system on a p2p network,” in *Proceedings of the 28th ACM symposium on Principles of distributed computing*. ACM, 2009, pp. 5–5, doi:https://doi.org/10.1145/1582716.1582722.

[12] D. Perkins, N. Agrawal, A. Aranya, C. Yu, Y. Go, H. V. Madhyastha, and C. Ungureanu, “Simba: Tunable end-to-end data consistency for mobile apps,” in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 7, doi:https://doi.org/10.1145/2741948.2741974. [Online]. Available: https://github.com/SimbaService/Simba

[13] N. Preguiça, M. Zawirski, A. Bieniusa, S. Duarte, V. Balegas, C. Baquero, and M. Shapiro, “Swiftcloud: Fault-tolerant geo-replication integrated all the way to the client machine,” in *2014 IEEE 33rd International Symposium on Reliable Distributed Systems Workshops (SRDSW)*. IEEE, 2014, pp. 30–33.

[14] R. Klophaus, “Riak core: Building distributed applications without shared state,” in *ACM SIGPLAN Commercial Users of Functional Programming*. ACM, 2010, p. 14, doi:https://doi.org/10.1145/1900160.1900176.

[15] S. Hao, N. Agrawal, A. Aranya, and C. Ungureanu, “Building a delay-tolerant cloud for mobile data,” in *2013 IEEE 14th International Conference on Mobile Data Management*, vol. 1. IEEE, 2013, pp. 293–300, doi:https://doi.org/10.1109/MDM.2013.43.

[16] B.-G. Chun, C. Curino, R. Sears, A. Shraer, S. Madden, and R. Ramakrishnan, “Mobius: unified messaging and data serving for mobile apps,” in *Proceedings of the 10th international conference on Mobile systems, applications, and services*. ACM, 2012, pp. 141–154, doi:https://doi.org/10.1145/2307636.2307650.

[17] S. Burckhardt, “Bringing touchdevelop to the cloud,” 2013, https://www.microsoft.com/en-us/research/blog/bringing-touchdevelop-to-the-cloud/.

[18] S. Burckhardt, M. Fahndrich, D. Leijen, and B. P. Wood, “Cloud types for eventual consistency,” in *European Conference on Object-Oriented Programming*. Springer, 2012, pp. 283–307, doi:https://doi.org/10.1007/978-3-642-31057-7_14.

[19] W. Brunette, S. Sudar, M. Sundt, C. Larson, J. Beorse, and R. Anderson, “Open data kit 2.0: A services-based application framework for disconnected data management,” in *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. ACM, 2017, pp. 440–452, doi:https://doi.org/10.1145/3081333.3081365.

[20] Y. Cui, Z. Lai, X. Wang, and N. Dai, “Quicksync: Improving synchronization efficiency for mobile cloud storage services,” *IEEE Transactions on Mobile Computing*, vol. 16, no. 12, pp. 3513–3526, 2017, doi:https://doi.org/10.1109/TMC.2017.2693370.

[21] P. Platform, “Parse platform,” 2016, https://parseplatform.github.io/.

[22] “The baasbox server,” https://github.com/baasbox/baasbox/, accessed: 2019-01-26.

[23] Dropbox, “Build your app on the dropbox platform,” 2016, https://www.dropbox.com/developers.

[24] G. Drive, “Google drive,” 2016, https://developers.google.com/drive/.

[25] “Amazon dynamodb - best practices for storing large items and attributes,” https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/bp-use-s3-too.html, accessed: 2019-01-26.

[26] Kinvey, “Kinvey baas,” 2016, https://www.kinvey.com/.

[27] “Kony mobilefabric,” http://docs.kony.com/7_0_PDFs/sync/kony_sync_orm_api_guide.pdf, accessed: 2019-01-26.

[28] Y. P. Faniband, I. Ishak, F. Sidi, and M. A. Jabar, “A review of data synchronization and consistency frameworks for mobile cloud applications,” *INTERNATIONAL JOURNAL OF ADVANCED COMPUTER SCIENCE AND APPLICATIONS*, vol. 9, no. 12, pp. 601–611, 2018.

[29] LevelDB, “A fast and lightweight key/value database library,” code.google.com/p/leveldb.

[30] P. O’Neil, E. Cheng, D. Gawlick, and E. O’Neil, “The log-structured merge-tree (lsm-tree),” *Acta Informatica*, vol. 33, no. 4, pp. 351–385, 1996.

[31] D. B. Terry, M. M. Theimer, K. Petersen, A. J. Demers, M. J. Spreitzer, and C. H. Hauser, “Managing update conflicts in bayou, a weakly connected replicated storage system,” in *ACM SIGOPS Operating Systems Review*, vol. 29. ACM, 1995, pp. 172–182, doi:https://doi.org/10.1145/224056.224070.

[32] “Netty - nio client server framework,” https://netty.io/, accessed: 2019-01-26.

[33] “Protocol buffers,” https://developers.google.com/protocol-buffers/, accessed: 2019-01-26.

[34] D. S. Parker, G. J. Popek, G. Rudisin, A. Stoughton, B. J. Walker, E. Walton, J. M. Chow, D. Edwards, S. Kiser, and C. Kline, “Detection of mutual inconsistency in distributed systems,” *IEEE transactions on Software Engineering*, no. 3, pp. 240–247, 1983.

[35] R. Van Renesse, D. Dumitriu, V. Gough, and C. Thomas, “Efficient reconciliation and flow control for anti-entropy protocols,” in *proceedings of the 2nd Workshop on Large-Scale Distributed Systems and Middleware*. Citeseer, 2008, p. 6.

[36] K. Ren and G. Gibson, “Tablefs: Embedding a nosql database inside the local file system,” in *APMRC, 2012 Digest*. IEEE, 2012, pp. 1–6.

[37] A. Shraer, A. Aybes, B. Davis, C. Chrysafis, D. Browning, E. Krugler, E. Stone, H. Chandler, J. Farkas, J. Quinn *et al.*, “Cloudkit: structured storage for mobile applications,” *Proceedings of the VLDB Endowment*, vol. 11, no. 5, pp. 540–552, 2018.

[38] T. T. Chekam, E. Zhai, Z. Li, Y. Cui, and K. Ren, “On the synchronization bottleneck of openstack swift-like cloud storage systems,” in *IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on Computer Communications*. IEEE, 2016, pp. 1–9.

[39] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, “Bigtable: A distributed storage system for structured data,” *ACM Transactions on Computer Systems (TOCS)*, vol. 26, no. 2, p. 4, 2008.

[40] D. B. Terry, V. Prabhakaran, R. Kotla, M. Balakrishnan, M. K. Aguilera, and H. Abu-Libdeh, “Consistency-based service level agreements for cloud storage,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM, 2013, pp. 309–324, doi:https://doi.org/10.1145/2517349.2522731.

[41] Apache, “Apache cassandra database,” http://cassandra.apache.org.

[42] J. C. Mogul, *The case for persistent-connection HTTP*. ACM, 1995, vol. 25, no. 4.

[43] M. Klems, D. Bermbach, and R. Weinert, “A runtime quality measurement framework for cloud database service systems,” in *Quality of Information and Communications Technology (QUATIC), 2012 Eighth International Conference on the*. IEEE, 2012, pp. 38–46.

[44] D. Bermbach, J. Kuhlenkamp, B. Derre, M. Klems, and S. Tai, “A middleware guaranteeing client-centric consistency on top of eventually consistent datastores.” in *IC2E*, 2013, pp. 114–123, doi:https://doi.org/10.1109/IC2E.2013.32.

[45] D. Bermbach and S. Tai, “Eventual consistency: How soon is eventual? an evaluation of amazon s3’s consistency behavior,” in *Proceedings of the 6th Workshop on Middleware for Service Oriented Computing*. ACM, 2011, p. 1.

[46] H. Wada, A. Fekete, L. Zhao, K. Lee, and A. Liu, "Data consistency properties and the trade-offs in commercial cloud storage: the consumers' perspective." in *CIDR*, vol. 11, 2011, pp. 134–143.

[47] A. S. Tanenbaum and M. Van Steen, *Distributed systems: principles and paradigms*. Prentice-Hall, 2007.

[48] J. Cao, Y. Zhang, G. Cao, and L. Xie, "Data consistency for cooperative caching in mobile environments," *Computer*, 2007.

[49] Y. Huang, J. Cao, B. Jin, X. Tao, J. Lu, and Y. Feng, "Flexible cache consistency maintenance over wireless ad hoc networks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 8, pp. 1150–1161, 2010.

[50] V. Balegas, S. Duarte, C. Ferreira, R. Rodrigues, N. Preguiça, M. Najafzadeh, and M. Shapiro, "Putting consistency back into eventual consistency," in *Proceedings of the Tenth European Conference on Computer Systems*. ACM, 2015, p. 6, doi:https://doi.org/10.1145/2741948.2741972.

[51] Y. Bai and Y. Zhang, "Stoarranger: Enabling efficient usage of cloud storage services on mobile devices," in *Distributed Computing Systems (ICDCS), 2017 IEEE 37th International Conference on*. IEEE, 2017, pp. 1476–1487.

[52] "Evernote system limits," https://help.evernote.com/hc/en-us/articles/209005247, accessed: 2019-01-26.

[53] A. Gheith, R. Rajamony, P. Bohrer, K. Agarwal, M. Kistler, B. W. Eagle, C. Hambridge, J. Carter, and T. Kaplinger, "Ibm bluemix mobile cloud services," *IBM Journal of Research and Development*, vol. 60, no. 2-3, pp. 7–1, 2016, doi:https://doi.org/10.1147/JRD.2016.2515422.

APPENDIX

TABLE IV. SUMMARY OF LARGE OBJECT SUPPORT IN REFERENCE DESIGNS

Framework	LO Support	Max file upload size	Chunking	LO Handling Technique
Simba [12] ∗	✗	-	✗	Does not support large objects.
SwiftCloud [13]	✗	-	✗	✗ SwiftCloud is a middleware system that implements a Key-CRDT on top of Riak.The Riak designers do not recommend storing objects over 50MB for performance reasons
Indigo [50]	✗	-	✗	✗ Indigo is based on SwiftCloud..The Riak designers do not recommend storing objects over 50MB for performance reasons.
Izzy [15]	✗	-	✗	Izzy is an initial version of Simba and do not support large objects
Mobius [16]	✗	-	✗.	Does not handle large files but addresses the messaging and data management challenges of mobile application development.
TouchDevelop [17] ∗ [18]	✗	-	✗	Special cloud types data do not address large size.
Open Data Kit 2.0 [19] *	✗	-	✓	Android Services have a 1 MB size limit on remote-procedure calls.To address 1MB limit, a primitive transport-level chunking interface using a client-side proxy to re-assemble chunks and only expose a higher-level abstraction to the tools.
StoArranger [51]	✗	-	-	Does not address large objects
QuickSync [20] ∗	179MB	✓	✓	QuickSync system is built on Dropbox and Seafile, cloud storage systems.Performance test included data size up to 179MB. Dropbox *chunked_upload* API support uploading of larger files of size 150 MB or more, in multiple chunks, with the feature of resuming if upload is interrupted. The chunk size is limited to 150MB and by default 4MB.
Parse Server [21] ∗	✗	10MB	✗	The ParseFile data type allows the app to store application files in the cloud that would otherwise be too large or cumbersome to fit into a regular ParseObject. PareObject allows data in bytes array or in the Stream form and SaveAsync call saves the file to Parse.
BaasBox [22] ∗	✗	100 KB	✗	BaasBox is based on java Play framework.BaasBox uses Play framework to manage REST. The Playframework Body parsers can be configured for the maximum payload size in POST operations. Body parsers is a HTTP request (for the POST and PUT operations) that contains a body. This body can be composed with any format specified in the Content-Type header.
Dropbox [23]	✓	150MB (REST API)	✓	Dropbox *chunked_upload* API support uploading of larger files of size 150 MB or more, in multiple chunks, with the feature of resuming if upload is interrupted. The chunk size is limited to 150MB and by default 4MB.
Evernote [52]	✓	5 MB-200 MB	-	Allowed upload size varies depending on the user type. In addition to text, notes can also contain attachments (called “Resources” in the Evernote API). These files can be of any file type.The attachment file is crated as a Resource object and added to the note. The documentation does not mention about chunking or upload of large objects.
Google Drive [24]	✓	Resumable upload for files more than 5 MB	✓	For larger files (more than 5 MB) or less reliable network connections, Google Drive support resumable upload with a single request or in multiple chunks.The PUT request allows chunks in multiples of 256 KB (256 x 1024 bytes) in size, except for the final chunk that completes the upload. Chunks size has to be kept as large as possible so that the upload is efficient.
iCloud with CloudKit [6]	✓	Documentation doesn’t specify a file size	✓	When the app adopt iCloud document management lifecycle, the operating system (iOS) initiates and manages uploading and downloading of data for the devices attached to an iCloud account. The app does not directly communicate with iCloud servers and, in most cases, does not invoke upload or download of data.
Amazon DynamoDB [25]	✓	400 KB	✗	Amazon DynamoDB enforce a maximum item size of 400KB in a table, including both attribute name binary length and attribute value lengths. If the application needs to store more data in an item than the DynamoDB size limit permits, the app can try compressing one or more large attributes, or it can store them as an object in Amazon Simple Storage Service (Amazon S3) and store the Amazon S3 object identifier in the DynamoDB item.
Bluemix Mobile Cloud Service [53]	✓	-	✓	When handling transfers of large files content is streamed and sent in chunks.WLResourceRequest API for iOS has a helper WLResourceRequest.sendWithDelegate API that allows downloading of large files.The upload endpoint reads the client app uploaded large file in chunks and sequentially writes to a local file in the filesystem.The download endpoint connects to a backend server and downloads a large file in chunks. Each chunk is written to the endpoint output stream to be read sequentially by the client app.
Kinvey [26]	✓	Upto 5TB using Google Cloud Storage	-	Kinvey support to store and retrieve binary files of size up to 5TB with the help of third-party service. Kinvey does not directly serve or accept files. Instead, the Kinvey Files API works by providing a short-lived URL to a third-party cloud storage service from which file(s) can be uploaded or downloaded.Currently, the third-party service used is Google Cloud Storage.
Kony [27]	✓	20MB	✓	The Large Binary Objects API allows you to retrieve and delete large binary objects, schedule a download, and get the location of the objects. While the Sync Chunking Mechanism applies to all of sync, the Large Binary Objects API supports the download of binary data stored in a particular object in multiple chunks. The download occurs in the background, allowing the user to perform tasks simultaneously.Default chunk size is 4MB and configurable to set transfer buffer size.

Note:
*: open-source