EDB: A Multi-Master Database for Liquid Multi-Device Software

Oskari Koskimies
Johan Wikman
Nokia Technologies
Otaniementie 19, FI-02150 Espoo, Finland
{firstname.lastname}@nokia.com

Tapani Mikola
Antero Taivalsaari
Nokia Technologies
Visiokatu 3, FI-33720 Tampere, Finland
{firstname.lastname}@nokia.com

***Abstract*— Device shipment trends indicate that the number of web-enabled devices will grow very rapidly. The rapid growth of different types of devices in our daily lives will fundamentally change the expectations on device synchronization. In this paper, we introduce EDB – a database architecture that has been built specifically to support automatic multi-master synchronization between multiple mobile devices with potentially intermittent network connectivity. EDB supports the broader vision of multiple device ownership and liquid software in which applications and services are expected to seamlessly roam from one device or computer to another.**

***Index Terms*—Liquid software; multiple device ownership; multi-device computing; multi-screen computing; multi-master synchronization; distributed databases; mobile databases; eventual consistency.**

I. Introduction

Device shipment trends indicate that the number of web-enabled devices is growing very rapidly. Every day, over 3.5 million new mobile devices and tablets are activated worldwide – over five times more than the number of babies born each day. We will quickly move from a world in which each person has only two or three devices – a PC, smartphone and tablet – to a world in which people will use dozens of devices in their daily lives: laptops, phones, tablets and "phablets" of various sizes, game consoles, TVs, car displays, digital photo frames, home appliances, and so on – all of them connected to the Internet. These internet-connected consumer devices will come in various shapes and sizes, targeting different usage situations, use cases and environments.

In general, we are at a tipping point with connected devices, entering a new era of multiple device ownership. This trend will only strengthen as other new types of gadgets and wearables such as smartwatches and intelligent eyewear become available more widely. This new era will dramatically raise the expectations for device interoperability, implying significant changes for software architecture as well.

In this paper, we will introduce *EDB* (also known as Elastic DataBase) that was designed specifically support multiple device ownership and the broader vision of *liquid software* discussed in Section II. EDB is a NoSQL document store that supports multi-master synchronization, specifically in the context of multiple mobile devices with potentially intermittent network connectivity.

The rest of this paper is structured as follows. In Section II, we will describe the liquid software vision and provide general motivation for a multi-master data architecture. In Section III, we will provide an overview of the EDB system, summarizing its overall architecture, requirements, key technical concepts and characteristics. In Section IV, we summarize the replication and conflict handling mechanisms that are at the heart of a true multi-master database. In Section V, we will discuss the current EDB implementation, followed by a summary of related work in Section VI. Finally, Section VII provides some concluding remarks.

II. Liquid Multi-Device Software

By *liquid software*, we refer to an approach in which applications and data can flow from one device or screen to another seamlessly, allowing the users to roam freely from one device to another without worrying about device management or having to remember complex steps. A central aspect of a true, casual multi-device computing experience is the ability to move fluidly from one device to another. This kind of behavior is also sometimes referred to as *experience roaming* or *experience continuity*.

Three simple liquid software usage scenarios are depicted in Fig. 1. In the top left image, the user is transferring images "on the fly" from a smart phone to a tablet. In the top right image, the user is transferring a live application from one tablet to another. In the bottom image, the user is transferring application state from her tablet to her car's navigation and entertainment system. In each case, the assumption is that the users can continue doing what they were doing on one device on the other devices, with seamless and fluid transition between the devices, and applications automatically adjusting to the form factor and specific capabilities of each target device.

This kind seamless, continuous usage of software across multiple devices is not generally supported yet, although such user interface concepts have been presented in a number of forums before [1, 2, 3]. Furthermore, some years ago cloud-based data and file synchronization services such as Apple Computer's *iCloud* (http://www.icloud.com) and Google's

Google Sync (http://www.google.com/sync) started raising the expectations and paving the way for automatically synchronized devices. Although these systems are limited to devices supporting the same native ecosystem, and generally for performing file and data synchronization only, they already point out the direction in which the industry is headed in the longer run.

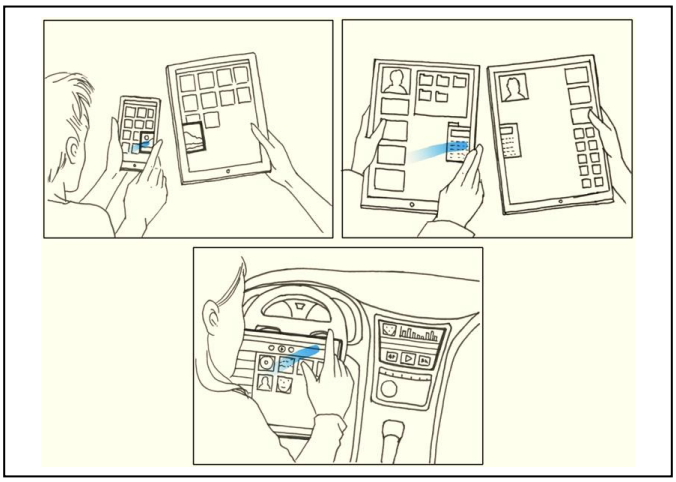

Fig. 1. Liquid software illustrated.

We recently published a *Liquid Software Manifesto* [4] in which we laid out the principles for seamless multi-device software. Apple's recent *Handoff* announcement is the first commercial manifestation of the broader vision of liquid software, allowing an Apple device user, e.g., to start editing e-mail messages or text messages on one Apple device and seamlessly continue editing them on another Apple device or computer (http://www.apple.com/ios/ios8/continuity/).

The term liquid software was originally coined by Hartman, Manber, Peterson and Proebsting in a technical report back in 1996 [5]. Their focus was primarily on enabling the flexible use of network-transported code on top of the Java platform [6]. However, the necessary underlying technical mechanisms are much the same regardless of whether the transportation state is related to user interface state or other areas. For more background and historical references on liquid software, refer to Section VI and our earlier paper [4].

III. EDB IN A NUTSHELL

EDB is a database architecture and implementation that has been created specifically to enable a liquid software environment. EDB is built for a heterogonous environment consisting of multiple battery powered personal mobile devices with limited CPU power, memory and network bandwidth, and possibly running different operating systems. The envisioned use case for EDB is that of a *device cloud* or *edge cloud* in which the user's devices (smartphones, tablets, laptops, smartwatches, etc.) can communicate with each other both locally and via the cloud. In a device cloud environment, devices will attempt to synchronize themselves primarily locally and directly (peer-to-peer) instead of performing synchronization via centralized clouds.

A. Architecture Overview

At the implementation level, liquid software implies that the applications and their data are kept in sync across devices. Furthermore, to support the illusion of truly liquid user interface behavior, it must be possible to carry user interface state information from one device to another as efficiently as possible. Ideally, state synchronization should occur automatically with minimum intervention required from the application or system software developer.

The requirement for data to be seamlessly available on all the user's devices leads system design towards a platform database architecture that transparently synchronizes data between devices, supporting multi-master replication with automatic conflict handling. The basic EDB architecture is depicted in Fig. 2.

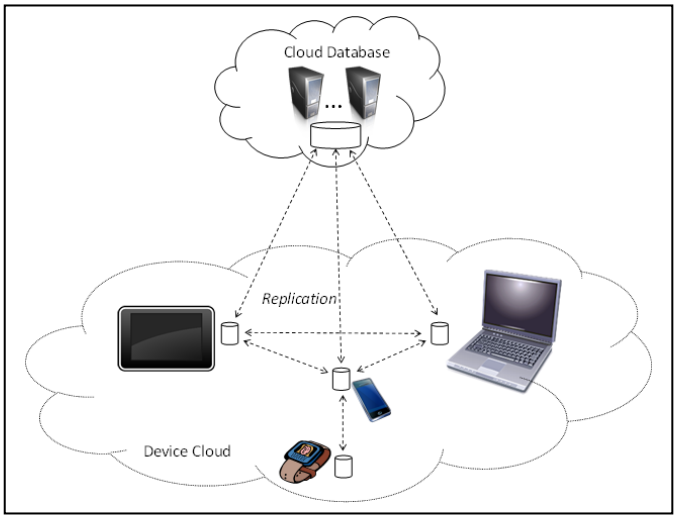

Fig. 2. Overview of the EDB architecture.

As seen in Fig. 2, each EDB-enabled device hosts a database instance, with multi-master replication occurring between the devices automatically when data on any single device changes. Local connectivity mechanisms such as Bluetooth LE or WiFi Direct can be utilized to synchronize the devices directly. Effectively, the user's devices form a *device cloud* that can – if necessary – operate independently of any conventional cloud backend. In addition, we assume that there is a *cloud database* – hosted on a conventional cloud-based server – that maintains a copy of the user's data, allowing the user to power off or even lose any or all of his devices, and yet rest assured that data is still available.

Note that EDB is used primarily for storing application metadata. Replication of large binary files (e.g., photos, videos, e-mail attachments) can be highly inefficient, and some of the target devices might be unable to store a large number of such files in the first place. Therefore, large binary content is kept outside EDB and handled separately by accessing and caching such content via URLs.

B. Requirements

In this subsection we explain the most essential requirements and design constraints behind EDB.

Full offline support. A mobile device may be offline or with limited connectivity for prolonged periods of time. Therefore, our system must provide full offline support, i.e., allow devices to be used without any network connection and then later re-synchronize themselves when a connection becomes available. Solutions that would only cache partial database contents on each device are not acceptable: if the user were to access pieces of data that are rarely used and available only on a specific device, the operation would fail when the device is offline.

Multi-master replication. Replication between devices must be *multi-master* (also known as *multi-primary* or *update-anywhere*); writes must be possible on all devices – not just on a central cloud database – or otherwise offline operation would be compromised. However, the cloud database *may* still have a privileged position, although it must necessarily be of the "first among peers" variety. For example, database operations that require a high degree of security (such as changing access rights) might only be possible on the cloud database.

No ACID guarantees. A device may be offline for prolonged periods of time and needs to be able to communicate with peer devices also in the absence of cloud. For example, if the user is traveling abroad and has disabled 3G/4G data connectivity in order to save money, a calendar entry made on his mobile phone should still be replicated locally to his tablet and be viewable there. This means that we optimize for availability and partitioning tolerance, not consistency [7]. In particular, changes are atomic only on a single data item level, and transactions are not supported.

Conflict handling. In a multi-master system with offline capability, replication conflicts are bound to happen every now and then. Therefore, a mechanism for handling conflicts is needed. However, since we are targeting primarily a single-user environment (where possible conflicts are caused by the same user who has entered conflicting information on two or more devices), the conflict handling mechanism can be kept simple and automatic. We will discuss conflict handling later in Section IV.

Access control and security. The database needs to be able to isolate applications from each other. For example, a travel application must not be able to read information from the user's calendar application, unless that data has been explicitly shared. However, all the data in the database belongs to the user, and hence there is no need to protect the data from the user himself. For example, if the user roots his device, he will theoretically have access to all the data, since he knows the password used to secure encryption keys. Thus an application must not store any data in the database that should be protected against the user (e.g., licensing information used to limit access to features). In general, the security mechanisms of the database must protect applications against other applications, users against other users, and users against applications, but not applications against the users.

C. System Architecture

EDB system architecture is shown in Fig. 3. Device-side components are depicted on the left, and the cloud side components on the right. White boxes refer to components that we have implemented ourselves, while gray boxes indicate components based on third-party technologies.

The *EDB API* includes the usual CRUD and indexing operations for documents, as well as batch versions. Both web-based (JavaScript) and native versions of the EDB API are offered.

Replicator is our database replicator that will be discussed in Section IV. The *EDB Platform Adaptation Layer* adapts the EDB API and replicator implementation to the specific target platform.

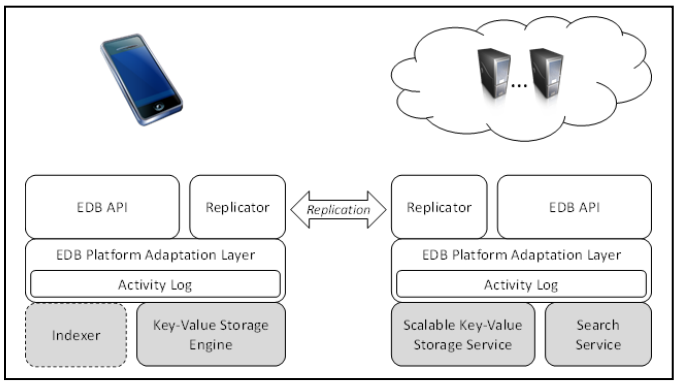

Fig. 3. EDB System Architecture Overview.

Gray, implementation-specific boxes in Fig. 3 will be described in Section V. In the remaining parts of this paper we will use the term 'device' to refer to EDB on a mobile device, and the term 'cloud' to refer to the cloud-resident database components. In addition, we will use the term 'sibling' as a generic term that can refer to either one; in other words, when term 'sibling' is used the intention is to convey that it does not matter whether we are talking about device-side or cloud-side components.

D. Key Concepts

In this subsection we explain the key concepts behind the EDB architecture.

Documents. An EDB database contains *documents* that are JSON objects consisting of named fields (properties). Every document has a property called *meta* – owned and maintained by the system – that contains meta information about the document, such as document ID, type and version number. The *meta* property cannot be directly written to by applications.

Types. Documents in EDB are *typed* – every document has a string property *meta.type*, which specifies the name of type that the document belongs to. A type has the following roles:

1) *To provide for a classification of documents*. One of the most common queries is to get all documents of a given type, and even when using more sophisticated queries it is typically useful to limit them to certain types of documents.

2) *To define how a document of that type must look like*. For example, a contact object must always have a name property. Note that EDB does not require a type to define a schema – if not defined, documents of that type are schemaless.

3) *To provide a basis for access control*. For example, documents of a certain type might only be accessible to system applications.

Types are specified by special system documents, which are managed and replicated just like any other documents (although there are convenience operations that operate on type names rather than document IDs). Their only distinguishing feature (aside from being interpreted as type specifications) is that their *meta.type* property is set to the special value *_type*.

Document versions. Each document in EDB has a *version* property, which contains the subproperties *t*, *v* and *r*, where *t* stands for timestamp, *v* stands for version and is an integer that is incremented each time the document is updated, and *r* stands for random and is a random integer that is re-generated each time when the document is updated.

When comparing two revisions of the same document, the one whose *t* is larger, that is more recent, is considered better. This implies that the clocks on all siblings of the system must be reasonably well in sync. The assumption is that every device fairly regularly connects to the cloud and that the clocks at that point can be adjusted. If *t* is the same on two revisions, the one whose *v* – that is, the document that has been modified more times – is considered better. If *t* and *v* are the same on both revisions, then *r* is used for creating an order. The benefit of this approach is that the revision can be generated in constant time, in contrast to, for instance, CouchDB [8] in which the revision is a version number plus a hash calculated from the body of the document.

Indexing and searching. At the moment, EDB does not provide means for free-form searching; rather, searching must always be performed against local, explicitly created indexes. Note that indexes are not replicated – this allows us to avoid the indexing overhead on devices where the indexes are not needed. Furthermore, unlike data, indexes can be omitted without compromising offline support since indexes can be created on-demand also when a device is offline. When necessary, we can support free-form searching using a separate server-side search service that uses Lucene indexing.

IV. Replication and Conflict Handling

One of the most critical features of a true multi-device database is replication and the associated conflict handling mechanisms. In this section, we will provide an overview of those mechanisms in the context of the EDB system.

Replication support based on activity log. A central goal in the design of the EDB replication mechanism was to reduce the number of roundtrips. Given that most of the participating device siblings are battery powered mobile devices, all extra radio activity has a direct impact on usage time. The assumption is that local processing is energywise much cheaper than extra communication.

In the replication process, one sibling always acts as a client and the other as server. In device-to-device replication, the device that opened the replication connection acts as the client. In device-to-cloud replication, however, the device will always be in client role. There are two reasons for this. First, devices are running on a cellular network and/or behind a NAT, and therefore they are in the general case unreachable from the cloud over IP. Second, only the mobile client is fully aware of the context; for instance, is it running in the home network or is it roaming, is a Wi-Fi network available, is the battery fully charged, and so on.

Our replication mechanism is based on an *activity log*. Conceptually the activity log is a list of IDs of documents that have been changed. The log is addressable in the sense that each position in the log corresponds to a unique value that can be stored. That is, it is possible to store the address of a position and later find out what documents have changed, after the point in time when the address was stored. The activity log is not a change log, since we only store information whether a document has been modified, and not how it was modified. Further, if a document is modified several times, it will still appear in the activity log only once, at the location that corresponds to the latest change.

For each sibling, we maintain a *replication offset* on the activity log that tells us which changes have been replicated to that particular sibling. If a sibling is up to date with the local database, its offset will point to the head of the log. An example is shown in Fig. 4, in which the last time that Sibling 1 and Sibling 2 replicated with the current host was at time 13 in the access log, and those siblings therefore have not seen the changes done since then, i.e., they have not seen the changes to documents D34 and D27 yet. Sibling 3, however, is up-to-date with the current host.

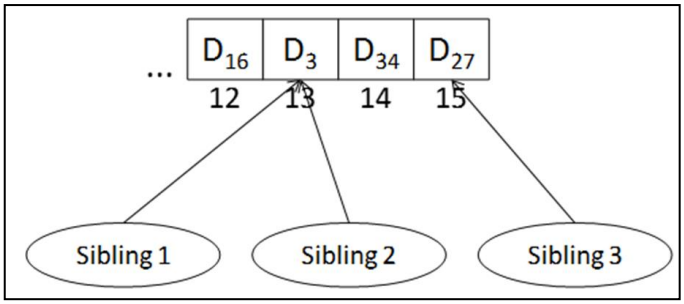

Fig. 4. Activity Log and Replication Offsets.

One benefit of this approach is that the size of the activity log is bound by the number of documents in the system, i.e., the activity log will not grow excessively. A drawback of the approach is that the order in which the documents were changed cannot be maintained. This may seem like a minor concern; however, as documents are replicated in the order that they appear in the activity log (i.e., ordered by last modification), we found that application developers were sometimes surprised by the order in which replication added new documents to the database. For example, if you first create a parent document, then child documents, and finally modify the parent document to contain references to the child documents, the parent document will be replicated last. On the other hand, in a multi-master replicating environment the order in which changes appear can only rarely be guaranteed, and applications should therefore be written defensively in any case.

Lossy conflict handling. The conflict handling approach used in EDB was initially motivated by CouchDB [8]. The conflict handling approach in CouchDB is such that the database deterministically selects a winning version and stores

the history, allowing the developer to later resolve the conflict in any way he sees fit. That is, CouchDB selects the winning version at conflict detection time, but allows the conflict to later be permanently resolved some other way, and requires old data to be erased explicitly by the programmer. When we were experimenting with CouchDB, this approach proved to be problematic, because with a significant amount of conflicts, the disk space consumed by the history would become prohibitive. Unless the history is purged regularly, the history information would soon consume much more space than the actual data.

In our typical use case – multi-master replication between devices owned and used by the same user – conflicts are typically of no real significance and the history of conflicts is of little interest. For instance, if the user ends up editing the same piece of contact information on two devices, entering (and losing some of the) conflicting information does not lead to any dramatic loss of information. Furthermore, we observed that a large fraction of the data was not documents explicitly relevant for the user but instead documents used by various application and system components for managing their state. In those cases the user would not be able to meaningfully resolve the conflicts anyway.

Even in those cases where the documents involved are directly meaningful to the user – for instance, a document corresponding to a contact – it is not self-evident how and when the user should be involved in the resolving of the conflict. Since the database is used by the entire platform and replication takes place at times deemed appropriate by the system and not, for instance, when explicitly instructed to do so by the user, conflicts can be detected essentially at any time. In general, a system in which the user might at any time be asked to resolve a conflict would not be pleasant to use. If the user would have to resolve a conflict when he attempts to use a document with conflicts, it might mean that he could not call someone before he has resolved the conflict. And if the user is allowed to postpone the time of conflict resolution, there is a risk that he would never actually do it.

A decision was made that in case of conflicts, not only is the winning document selected, but the losing one is deleted. This can be described as *lossy* conflict resolution, as opposed to the traditionally employed lossless ones which strive to prevent data loss at all costs. This approach proved out to be the most pragmatic and appropriate for our use cases.

Replication frequency. Although immediate change updates across devices would be the ideal, for power consumption reasons replication cannot be assumed to be always on. Rather, replication frequency depends on the power budget of a device and the availability of local connectivity mechanisms. In many cases, updates may be delayed occasionally in order to conserve energy. While prioritizing data according to its urgency may sound tempting, we cannot rely only on prioritization by third party applications, since they would most likely optimize their behavior for the user experience of each specific application rather than for the overall system power consumption.

Comments and observations from developer perspective. From the developer's perspective, a replicating multi-master database such as EDB poses some interesting challenges. For instance, there are few guarantees regarding the order of changes from other devices. Changes may arrive out of order. Although changes to a single data item are atomic, automatic conflict resolution may mix and match changes made to multiple data items in various interesting ways.

We experienced this problem firsthand when an application would create a tree hierarchy of objects on one device, and the documents would arrive out of order on a second device, causing the application to fail because the developer had expected documents to appear on the second device in the same order as they were created on the first device. Because of this expectation, the developer did not take into account that a child document might appear in a replicated database earlier than a parent document does, even though the parent document was originally created first.

In practice this means that whenever there are any kind of data integrity constraints that span multiple documents, defensive programming style is required. The programmers are much better off redesigning their applications' data models so that there are no multi-document integrity constraints. Our recommendation is that applications should employ a design pattern in which database changes are monitored by the application but they are always validated before updating the application's internal data model, and business logic only reacts to and acts on the internal data model. The idea is that the internal data model isolates UI and business logic from the vagaries of replicated database state, and an application-specific adapter maps between database contents and the application's internal data model, validating and interpreting as necessary. This design approach is similar to the Model-View-ViewModel (MVVM) design pattern [9].

V. Implementation

EDB was implemented in a project that investigates liquid software, with the goal to explore the creation of a software platform for seamlessly operating mobile devices. Both web-based and native (Android) devices were targeted. So far, EDB implementation has been used in the creation of several typical mobile applications, such as a phonebook, a note editor, an email client, and a photo gallery application, all supporting seamless data synchronization across multiple devices.

Technology choices. At the implementation level, EDB is not a full database implementation, but rather a replication and API layer on top of an existing key-value store, as shown earlier in Fig. 3. We first implemented the device-side EDB using SQLite3 as the underlying key-value storage engine. The idea was that SQL query features could be used for implementing more advanced queries in EDB. However, this created too many dependencies between the EDB query capabilities and SQL, casting doubt on our ability to adapt EDB later on for non-SQL database engines. We then simplified the query capabilities supported by EDB natively, and added the ability to use an external indexer (or search service) for free-form searching (see Fig. 3). The simplification allowed us to switch the on-device key-value storage engine to LevelDB (http://leveldb.org/) that only offers a minimal feature

set and thus ensured that our API could be provided practically on any key-value storage engine. EDB indexes were implemented on top of LevelDB.

Amazon DynamoDB (http://aws.amazon.com/dynamodb/) was selected early on as the underlying key-value storage service on the server side. Originally we planned to use DynamoDB secondary indexes for EDB indexes, but that would have severely limited the number of indexes that could be created. This would have also prevented us from encrypting data in DynamoDB. We opted instead to use an on-premise ElasticSearch cluster that allowed us to create an arbitrary number of indexes and encrypt all user data in the Amazon Web Services (AWS) cloud while maintaining searchability, and still rely on the AWS cloud for durability of the data. If the ElasticSearch cluster failed, it could always be rebuilt based on the data stored in DynamoDB. An additional benefit was the possibility to use ElasticSearch for free-form search as well.

The activity log is implemented on the device side as a separate LevelDB database where the key of a log entry is a timestamp concatenated with document ID. The value of a log entry is empty for updates, but for deletes it contains the document metadata so that the document can be completely removed from the document database after deletion. On the server side, the activity log is implemented using last-modified timestamps for documents and by creating a DynamoDB local secondary index for the documents that allows them to be traversed in last-modified time order. Some server state needs to be stored at client (similar to a cookie) to deal with the clock skew issue inherent with multi-server cloud setups.

Storage limits and handling of large binary file content. As was already mentioned earlier, EDB is used primarily for storing application metadata, and large binary file content is kept outside EDB. While the storage capabilities of different devices vary, we require all participating devices to store the entire contents of the database locally, i.e., all the non-file data of the user's working set, including the file metadata – contacts, calendar entries, email headers, picture metadata, and so on. In other words, all devices always have a complete record of what data the user has in his working set, even though the actual (file) data may not be present on all devices. The file data itself is cached on each device, and fetched from cloud storage if missing. The user is afforded some control over what data is kept cached so as to support offline operation (for example, to ensure that a particular movie is cached locally before boarding a long flight).

VI. Background and Related Work

There is a long body of research and development work in the area of distributed databases and file systems dating back to the 1980's. Traditionally, such systems were designed for server environments or client-server systems for keeping data in sync across multiple sites or datacenters over a conventional (wired) network.

Distributed file systems. Historical examples of distributed file systems supporting automatic data (file) replication include *Coda* [10] and *Ficus* [11]. Both systems address the problem of disconnected operation, and they support conflict handling with file type specific conflict resolvers and the ability to take file location into account when choosing resolution policy. Built-in conflict resolvers for directories are provided.

The Coda architecture is somewhat similar to what we would today call a traditional cloud architecture, employed by services such as Dropbox. In this architecture, clients replicate with a server to support disconnected operation, and servers replicate with other servers to support high availability. However, peer-to-peer replication between clients is not supported. In Ficus, in contrast, a peer-to-peer architecture is used. Similar to EDB and Bayou (discussed below), the servers employ multi-master replication where any node can make changes at any time. This means that a set of devices can replicate locally with each other even though connectivity to the main network is missing. For example, two colleagues might collaborate on a document while traveling, using the replication to ensure that they see each other's changes, even though neither has connectivity to the corporate network where the main file servers are.

Distributed databases. A distributed database is a collection of multiple, logically interrelated databases distributed over a computer network. *Bayou* [12] is a distributed, multi-master storage system designed to support offline operation. Client applications access Bayou servers, which replicate data between each other. As in EDB, data may be updated on any replica, and consequently any device that has a server is capable of offline operation. Per-write conflict resolution policies are used to handle update conflicts.

In Bayou changes are at first only tentative. Each data item has a primary server, and changes only become committed (final) once the primary server has handled it (possibly merging data from conflicting versions in order to do so). The primary server can be chosen to coincide with the locus of the update activity. For example, by placing the primary server for a user's own data on the user's laptop, data updates done on that laptop can commit immediately even when offline. Clients can determine the state of an update (tentative or committed) and expose this information to users; for instance, a calendar event might be shown as tentative until it gets committed. Conflict resolution is per-write rather than per-type, and conflicts resulting from application semantics are also supported (e.g., a conflict might result from two meeting room booking data items that have overlapping times for the same meeting room).

Dynamo [13] is a key-value database that has been used by Amazon for some of their core services (note that the DynamoDB service offered by Amazon is an entirely different database). Dynamo employs multi-master replication and eventual consistency to provide high availability. In Dynamo, replication is not meant to support disconnected operation for mobile devices, but it is rather used to support high availability.

Unlike in Bayou, in which conflict handling policy is specified at write time, Dynamo stores multiple versions of data in case of conflict and delivers all of them to a client at read time. The client can then resolve the conflict by updating the value. Another difference to Bayou is that in Dynamo conflicts can only happen because of conflicting updates to the

same data item; conflicts resulting from application semantics are not supported.

Since conflict handling is read time and left up to the application, Dynamo does not have the same concept of tentative and committed updates as Bayou. However, if an application writes a value to the database, an immediately following read operation is not guaranteed to return the same value since it may access a different replica that has not yet received the update. Applications can, however, elect to trade availability for durability and consistency. Each operation on a data item is processed by the top N nodes in the node preference list for that particular data item. By requiring participation from a high enough number of nodes (e.g., > N/2 for both read and write), the application can guarantee that once a value has been successfully written, a subsequent read operation will be return that value (barring further changes).

CouchDB [8] is a distributed database in which multi-master replication is used for guaranteeing reliability and high availability – the word "couch" is actually an acronym for "Cluster Of Unreliable Commodity Hardware". Like Dynamo, CouchDB stores multiple versions of a data item when conflicts are detected during replication. However, CouchDB automatically chooses a winning version which is returned by subsequent read operations. The selection algorithm is deterministic so that eventual consistency is assured even in a partitioned network where several nodes might end up independently making the selection.

From the application viewpoint, however, conflicts are typically detected at write time (as opposed to Dynamo in which they are detected at read time); the application must specify the version of data it wishes to overwrite, and if that is not the current version, an error will result, forcing the application to refetch the document and redo the update. Note that a conflict would not trigger this kind of error if the current value in the database was auto-selected as the winner in a conflict. In this case a conflict can only be detected by explicitly querying for conflicts.

CouchDB has a relatively simple replication scheme in which the replicator is really just another database client. This has resulted in the creation of multiple open source implementations of mobile (on-device) databases that support CouchDB replication. These databases are a popular method for adding robust offline capability to a mobile application.

Cloud-based databases for mobile applications. There are a number of recent cloud-based database systems that offer built-in mechanisms specifically for keeping multiple clients effortlessly in sync. These systems are based on a master-slave architecture, keeping data primarily in a cloud database, but they use a local in-memory database as a latency optimization and to minimize the effects of transient connectivity loss.

Two examples of such systems are *Firebase* [14] and *Meteor* [15]. In both systems, the local database contains only the part of the main database the application currently needs, i.e., the working set of the application. Rather than performing operations directly on the cloud database, the mobile application performs the operation on the local database. Replication is used for transferring data between cloud database and local database. This approach allows client-side database operations to be fast and operational even when the device is disconnected. However, there is a tradeoff regarding consistency – local database changes may be overruled by the cloud. This is rarely a problem, though, as these systems are not meant for prolonged offline use. In systems such as Firebase and Meteor, the use of a local database and replication is only meant to mask wireless network latency and short periods of disconnection from the centralized cloud.

The main difference between Firebase and Meteor is how they handle consistency and conflict resolution. Firebase uses transactions with update functions specified by the client. The update function gets the current value of the data it is updating as a parameter. The function is first applied on the client and later (when connectivity is available) on the server. If a conflict is detected when applying the function on the server side (i.e., the original value is different on the server side than it was on the client side), the client side value is updated to the most recent value and the update function is re-run. The update function may abort the transaction if the current value has changed so that the update no longer makes sense (e.g., a deduction of funds might fail if the account balance is now zero). In this case the update is also rolled back on the client side. However, the rollback will not affect any other transactions that used the tentative client-side value before it was rolled back – the application itself is responsible for handling cascading roll-backs.

Meteor, in contrast, uses traditional backend functions that use the features of the backend database (MongoDB) to guarantee consistency. However, the code of those backend functions can also be made available on the client side, in which case the functions are first run on the client side to create a simulated (tentative) version of the data. This data will be displayed briefly to the user and soon overwritten by the real data that was generated by the backend function running on the server side. Compared to the Firebase approach, this has the advantage that as long as all the dependent updates are performed using backend functions, too, cascading rollbacks are not a problem – the dependent simulated values written on the client side will simply be overwritten by the corresponding real values written by the backend functions on the server side, or rolled back if those functions raise an exception. Another benefit is that the backend functions can perform arbitrary processing to perform their task – the client side logic can deviate from server side logic if needed.

Additional comments and observations. Currently dominant mobile operating systems all provide the beginnings of multi-device support by allowing small amounts of application state to be shared between devices via cloud-based notification services. However, these features are meant primarily for synchronizing configuration data, preferences, and small amounts of app-related data, e.g., which game levels the user has completed. The functionality and scalability of such mechanisms are insufficient for liquid software use cases. File synchronization through cloud storage services such as Google Drive, Microsoft OneDrive or Dropbox is also available. However, using them for database replication (e.g.,

by implementing a key-value store on top of the file system) is problematic from performance viewpoint, and could exceed access quotas in more extensive use.

In principle, basic liquid software support can be implemented at the mobile application level by utilizing a built-in on-device database such as SQLite that is available in dominant mobile operating systems. SQLite itself does not support replication, but replication functionality could be added on top of it. By using an on-device database and replicating with the cloud – rather than remotely accessing the cloud database – applications could provide robust offline capability and also reduce database access latency. However, on mobile devices such solutions could be disastrous from energy management perspective, with each application triggering replication (and thus turning radios on) independently without any coordination by the operating system.

In practice, liquid software requires a platform database that is shared by all applications and offers both adequate database features and efficient replication with appropriate considerations for system-wide power and energy conservation. Our own work in this area was originally motivated and inspired significantly by CouchDB as well our earlier work on the Data API of the Cloudberry HTML5 platform [16]. However, as we realized that there is currently no system that would meet all our requirements, we ended up designing and implementing EDB. For a summary of related work on liquid multi-device software, refer to our earlier paper [4].

VII. Conclusions

We take it for granted that we are at yet another turning point in the computing industry. The dominant era of PCs and smartphones is about to come to an end. So far, standalone devices have been the norm, and software has been primarily associated with a single device. We believe that in the computing environment of the future, the users will have a considerably larger number of internet-connected devices in their daily lives than today. Unlike today, no single device will dominate the user's digital life.

In this paper we have introduced EDB – a database architecture and implementation that has been created specifically to support multiple device ownership and the broader vision of liquid software in which applications and their data are expected to be seamlessly available on all the devices that the user has. EDB is a key-value database with built-in support for multi-master data replication across devices, with a lossy conflict resolution algorithm for handling replication with less overhead than in competing systems. While EDB is not a production system yet, it has already been used for implementing various core mobile applications for a feature-rich multi-device environment.

In summary, while liquid software may still seem like science fiction, the technical ingredients and enablers for realizing the vision are already largely in place. We believe that within the next ten years, seamless multi-device operation will be the norm rather than an exception. Automatically replicating multi-master databases supporting offline operation will play a central role in realizing the broader vision. We hope that this paper, for its part, encourages people to continue work in this exciting area.

References

[1] D. Dearman and J.S. Pierce, "It's on my other computer!": computing with multiple devices. Proc. CHI'2008 (Florence, Italy, April 5-10), 2008, pp. 767-776.

[2] M.A. Nacenta, D. Aliakseyeu, S. Subramanian, and C. Gutwin, A comparison of techniques for multi-display reaching. Proc. CHI'2005 (Portland, Oregon, USA, April 2-7), 2005, pp. 371-380.

[3] S. K. Kane et al., Exploring cross-device web use on PCs and mobile devices. Proc. Interact 2009, pp. 722-735.

[4] A. Taivalsaari, T. Mikkonen and K. Systä, Liquid software manifesto: the era of multiple device ownership and its implications for software architecture. Proc. 38th Annual International Computers, Software & Applications Conference (IEEE COMPSAC'2014, Västerås, Sweden, July 21-25), 2014.

[5] J. J. Hartman, U. Manber, L. L. Peterson, and T. A. Proebsting, Liquid software: a new paradigm for networked systems. Univ. of Arizona Tech Report TR 96-11, 1996.

[6] J. J. Hartman, P. A. Bigot, P. Bridges, B. Montz, R. Piltz, O. Spatscheck, T. A. Proebsting, L. L. Peterson, and A. Bavier, Joust: a platform for liquid software. IEEE Computer, April 1999, pp. 50-56.

[7] N. Lynch and S. Gilbert, Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, Volume 33 Issue 2, 2002, pp. 51-59.

[8] J. C. Anderson, N. Slater and J. Lehnardt, CouchDB: The Definitive Guide (1st ed.), O'Reilly Media, 2009, ISBN 0-596-15816-5. URL: http://guide.couchdb.org/.

[9] J. Gossman, Introduction to Model/View/ViewModel pattern for building WPF apps, Microsoft Developer Network Blogs, Tales from the Smart Client (Oct 8th), 2005. URL: http://blogs.msdn.com/b/johngossman/archive/2005/10/08/478683.aspx.

[10] M. Satyanarayanan, J.J. Kistler and E.H. Siegel, Coda: a resilient distributed file system. IEEE Workshop on Workstation Operating Systems, Nov. 1987.

[11] P. Reiher, J. Heidemann, D. Ratner, G. Skinner and G. Popek, Resolving file conflicts in the Ficus file system. Proc. USENIX Summer 1994 Technical Conference, Volume 1, 1994.

[12] 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. Proc. 15th ACM Symposium on Operating Systems Principles, 1995.

[13] D. Hastorun, M. Jampani, G. Kakulapati, A. Pilchin, S. Sivasubramanian, P. Vosshall, W. Vogels, Dynamo: Amazon's highly available key-value store. Proc. 21st ACM Symposium on Operating Systems Principles, 2007.

[14] Firebase, Inc., Firebase developer documentation. URL: https://www.firebase.com/docs/.

[15] I. Strack, Getting Started with Meteor.js JavaScript Framework. Packt Publishing, 2012.

[16] A. Taivalsaari and K. Systä, Cloudberry: HTML5 cloud phone platform for mobile devices. IEEE Software, July/August 2012, pp. 30-35.