# StackSync: Architecturing the Personal Cloud to Be in Sync

Pedro García López, Marc Sánchez Artigas, Cristian Cotes, Guillermo Guerrero, Adrian Moreno, and Sergi Toda

Universitat Rovira i Virgili,
Tarragona, Spain
{pedro.garcia,marc.sanchez,cristian.cotes,guillermo.guerrero,
adrian.moreno,sergi.toda}@urv.cat

**Abstract.** In the last few years, we have experienced a rush of online storage services with a complete set of tools for file syncing, sharing and collaboration. Unfortunately, commercial Personal Cloud solutions, e.g. Dropbox, Box, SkyDrive and the likes, are closed and proprietary, which supposes a serious impediment to progress research, also forcing people to be stranded into locked systems. In this context, we argue about the necessity for an open-source Personal Cloud framework that provides scalable file synchronization and sharing, and that allows anyone to easily implement and evaluate new ideas. This framework, called StackSync, is modular and extensible, and contains all the software pieces to run a basic Personal Cloud, namely support for metadata management, efficient notification, deduplication, and data storage, among others. Our reference implementation in Java is a push-based architecture built around an asynchronous high-performance message broker (RabbitMQ). We demonstrate the feasibility of our approach using our standard benchmarking test suite.

**Keywords:** Personal Cloud, synchronization, storage, messaging middleware

## 1 Introduction

In the last few years, we have witnessed a clear shift from Personal Computers and local storage to multi-device access and cloud storage services. The Personal Computer (PC) paradigm is slowly waning as traditional PCs are being outnumbered by other devices like smartphones and tablets. This means that in the next years users will access their digital lives in the cloud from a variety of heterogeneous devices and platforms. This is clearly reflected by the massive adoption of services like Dropbox, U1, Google Drive or SkyDrive, among others. The term "Personal Cloud" has been coined in the last years to reflect this trend and many analysts forecast a bright future for these solutions [11][1]. To clarify the term we propose here the so-called 3S Personal Cloud Definition:

*The Personal Cloud is a unified digital locker for our personal data offering three key services: Storage, Synchronization and Sharing. On the one hand, it must provide redundant and trustworthy cloud data storage for our information flows irrespective of their type. On the other hand, it must provide syncing and file exploring capabilities* *in different heterogeneous platforms and devices. And finally, it must offer fine-grained information sharing to third-parties (users and applications).*

Despite the much trumpeted success of the so-called cloud storage revolution, exemplified by Dropbox and the likes, little is known about the architecture of these commercial solutions, and existing open source alternatives (e.g., ownCloud[1], SparkleShare[2]) fall short of addressing all the requirements of the Personal Cloud, mainly in terms of scalability and extensibility. In this sense, architecturing a Personal Cloud is not trivial, as this model is still in early stages, and no reference architecture exists.

Architectural complexity is well captured by file synchronization, as the success of Personal Cloud services lies in great measure in the scalability of their sync services. In this cloud paradigm, a desktop client software typically keeps all the files in a specified folder in sync with the servers, automatically synchronizing changes across the devices of the same user. Since files can be shared with other users, changes in shared folders must be also synced with every account that has been given access to the shared folders.

Making this process scalable is not straightforward, as it involves the intricate interaction of many components. For instance, to improve scalability, Personal Clouds use numerous techniques to only transfer those parts of the files that have been modified since the last synchronization. To achieve this, these systems internally do not use the concept of files, but split every file into chunks, each treated as an independent object. If a chunk is already stored in the storage servers, it is not transferred, saving traffic and storage. However, working at the chunk level requires the sync process to manage more metadata than operating at the file level, making it more critical the way metadata is internally processed. Further, as the amount of metadata is directly proportional to the number of chunks, the efficiency of the chunking algorithm impacts the performance of file syncing.

Analogously, to efficiently maintain the consistency of files, any change performed elsewhere must be advertised as soon as they occur to reduce conflicts [9], in particular, when a file is susceptible to be modified by more than one client at the same time. This requires the sync process to operate quickly to commit changes along with an efficient notification service to inform clients about file modifications.

Therefore, a practical implementation of an open Personal Cloud requires making a big effort. In this context, the CloudSpaces project (`http://cloudspaces.eu/`), which has been initiated in the frame of European Commission's FP7-ICT programme, is attempting to fill this gap. Concretely, as far as we know, we are the first to propose an open architecture for Personal Clouds. It is worth noting here that although at the time of this writing, our reference implementation includes the software necessary to run a basic Personal Cloud, in this work we put more emphasis on the sync protocol, which is the core service of any Personal Cloud. Since a serious impediment to progress in cloud storage research is the lack of a standard framework, we hope to provide a useful framework for developing and testing new ideas in a relatively easy manner.

In summary, we contribute the following:

1. Extensible framework with a plugin-based architecture for storage back-ends, metadata-back-ends, communication middleware, and chunking algorithms.

[1] `http://owncloud.org/`
[2] `http://sparkleshare.org/`

2. Reference open source implementation called StackSync, which is based on a high-performance messaging middleware for asynchronously dispatching incoming and outgoing push notifications.
3. Benchmarking and validation framework for Personal Clouds, which provides trace generators and test scripts.

The rest of the article is structured as follows. We review related work in Section 2. We give an overview of the main technical ingredients of a Personal Cloud in Section 3. We introduce our open architecture in Section 4. In Section 5, we evaluate our reference implementation and conclude in Section 6.

## 2 Related Work

### 2.1 A Brief History of the Personal Cloud Term

In the past years there have been divergent views of the “Personal Cloud” concept. For example, in [13] authors propose an architecture and design for accessing and sharing computational resources in virtual machines. For them, a Personal Cloud is a collection of Virtual Machines running on unused computers at the edge. Another different view focuses on collaborative work [6], where a web infrastructure is defined to provide a unified environment for handling activities and collaborations. Finally, a recent trend [26] goes further and defines the Personal Cloud as a cloud Operating System that offers a core set of services around identity, trust, data access and even programming models.

In this paper, we focus on Personal Cloud Storage platforms that take care of data sync and sharing from heterogeneous devices. In fact, the term “Personal Cloud” have received a lot of attention with the recent research reports from Forrester [11] and Gartner [1]. Like us, these reports associate the term Personal Cloud with online cloud storage services such as Dropbox, Box, or Google Drive among others.

### 2.2 Personal Cloud Systems

In the last few years, we have seen how the market of cloud storage is growing rapidly. Despite the rush to simplify our digital lives, many of the commercial Personal Clouds in operation today like Dropbox are *proprietary*, and rely on algorithms that are *invisible* to the research community, and what is even worse, existing open source alternatives fall short of addressing all the requirements of the Personal Cloud. Next we discuss the existing open source solutions for the Personal Cloud, namely SparkleShare, ownCloud and Syncany.

SparkleShare[3] is built on top of Git, using it as both its storage and syncing back-end. SparkleShare clients use push notification to receive changes, and maintain a direct connection with the server over SSH to exchange file data. When a client is started, it connects to a notification server. The notification server tells the other clients subscribed to that folder that they can pull new changes from the repository after a user change.

[3] `http://sparkleshare.org/`

Using Git as the storage back-end is a double-edged sword. While Git implements an efficient request method to download changes from the server (git pull command), avoiding massive metadata exchanges between server and clients, it is not prepared to process large binary files. Also, this architecture tied to the Git protocol is also difficult to scale and deploy in cloud environments.

ownCloud[4] is the most famous open source Personal Cloud and they have an active community. We refer here to the Community Edition of ownCloud, since their enterprise edition is not available to the public. In ownCloud, clients communicate with the server following a *pull* strategy, i.e. clients ask periodically to the server for new changes. There are two types of data traffic: data and metadata. For data exchange, ownCloud uses a REST API; however, metadata traffic is transferred using the WebDAV protocol. Because both types of traffic are processed by the same server, data and metadata traffic are completely coupled.

Unfortunately, ownCloud is not an extensible and modular framework like StackSync. In this line, their developer community is mainly working around the web front-end. Although we will devote a subsection to ownCloud in the validation, we can advance that their inefficient pull strategy with massive control overheads is not scalable. Furthermore, their sync flows and data flows are tightly coupled, and they do not even provide basic chunking or deduplication mechanisms.

Syncany [5] is an open source Personal Cloud developed by Philip Heckel in Java. It is a client-side Java application that can synchronise against a variety of storage backends thanks to their extensible plugin model. They also provide extensible mechanisms for chunking and their architecture is elegant, clean and modular.

Although we give much credit to Syncany, proof of that is that the StackSync client is a branch project that evolved from Syncany, it presents a number of drawbacks that made us evolve towards the current StackSync architecture. The major shortcoming is the lack of scalability of Syncany due to its heavy pull strategy with metadata and data flows heavily coupled. To support versioning and resolve conflicts, Syncany relies on a metadata file that contains the complete history of each individual file, and that is stored in the storage back-end as a regular file. To determine the most recent version of a file, the Syncany client needs first to download this metadata file, which grows with each new file modification, severely limiting scalability.

## 2.3 Synchronization Algorithms

At the core of personal cloud is file sync. Although a rush of online file sync services have been entering the market during the last years, evidenced by the explosion in popularity of Dropbox and competitors, little is known about the design and implementation of commercial sync protocols. According to a recent characterization of Dropbox [9], file synchronization is built upon third-party libraries such as librsync, but the role of this library is uncertain because of the very nature of the rsync algorithm [25]. Other popular tools like unison [21] that use the same basic algorithm suffer from the same deficiencies.

[4] `http://owncloud.org/`
[5] `http://syncany.org/`

rsync is symmetric, and provides pairwise synchronization between two devices, where the rsync utility running on each computer must have local access to the entire file. This requirement poses the first practical limitation to the adoption of rsync because working at the file level prevents efficient data deduplication. To save storage space and money, services like Dropbox split files into chunks and store them at multiple nodes on the server side. A straightforward adaptation of rsync to this context would be piecing together the chunks and reconstructing whole file at a chosen server and then operate on it. This, unfortunately, would waste massive intra-cluster bandwidth, deteriorating significantly deduplication efficiency. It is worth noting here that, although rsync finds chunks of data that occur both in the old file and the new file, it requires the side acting as a server to compute hashes for all possible alignments in its file in order to find a match. For this, it needs the whole file.

In addition, if a single character is modified in each chunk of the old file, then no match will be found by the server and rsync will be completely useless [17]. To address this limitation, a number of single-round and multi-round protocols have been proposed in the last ten years. Multi-round protocols allow communicating fewer bits in total by using additional communication rounds; see [17] and [22]. However, the fact of taking multiple passes over files presents evident disadvantages in terms of protocol complexity, computing and I/O overheads, and communication latency. Recent single-round protocols [16][12] bypass this difficulty by using variable-length content-based chunking [19]. However, since these protocols only synchronize files between two different machines at a time, they are not directly applicable to Personal Clouds, where file changes occurring elsewhere are automatically notified to any other device sharing that file.

There is a large body of work by the OS community that attempts to detect redundancies in order to reduce storage or transmissions costs, like LFBS [19] and Pastiche [8], among others, which have inspired in one way or another many of today's online cloud storage services. These systems operate at block level by relying on variable-length content-based chunking, rather than at file level. Compared with personal cloud applications, distributed file systems pursue a different objective and can skip implementing some basic functionality in personal clouds like file version management or the scalable notification of updates as soon as they occur. In fact, LBFS uses leases in which the server's obligation to inform a client of changes expires after one minute. The result is that the client will be out of sync once a lease on a file has expired. In Dropbox, however, any change on the central storage is advertised as soon as it is made [9]. For this purpose, the Dropbox client keeps continuously opened a TCP connection to a notification server, used for receiving information about changes performed elsewhere.

Overall, *providing an efficient and scalable sync protocol* poses a grand challenge for engineers in charge of building Personal Cloud storage services, due to the intricate relationships between deduplication, notification and metadata management.

## 3 Understanding Personal Clouds

Personal Clouds are complex infrastructures involving a variety of distributed components. Even for setting up and running a basic service, engineers in charge of building it need to implement many processes. The lack of a reference open source implementation complicates even more the design of these systems.

To put in perspective, we describe next the typical interactions among the different blocks of a Personal Cloud architecture. Among the three key services, here we will focus only on the synchronization service. The sharing service can be considered as a particular case of file synchronization where users set sharing controls at the account level to grant access of shared folders and links to other users. On the other hand, the storage service must offer a clean and simple file-system interface for archiving, backup, and even primary storage of files, abstracting away the complexities of direct hardware management. At the same time, the storage service must guarantee the availability of data files stored by users, which is achieved by adding redundancy to multiple servers. As the implementation of the storage service is more related to hardware issues like redundancy management, and it is in general not architecturally relevant, discussion on the storage service has been skipped due to space constraints.

It must be noted that Personal Clouds usually treat differently desktop clients in PCs and laptops from mobile and Web clients. Whereas the former maintain a copy of the information and check for changes, mobile clients use to retrieve the information on demand. We will focus only on the desktop scenario.

## 3.1 Basic Blocks and Interactions

Personal Clouds usually provide desktop clients that integrate with the OS file explorer capabilities. These clients include a `Watcher` component that monitors file changes in a specified synced folder. We will call this working folder the *Workspace*.

When the `Watcher` is notified of any change in the Workspace by the OS, it then informs the `Indexer` component on the new changes. The `Indexer` maintains a local database with information about the contents of the Workspace including files, folders, hashes, and even versions. Internally, Personal Clouds typically do not use the concept of file, but rather operate on a lower level by splitting files into chunks, each treated as independent object and identified by a fingerprint (like SHA-256 hash [9]). In that case, the local database maps the fingerprints to the corresponding files. The reason to work at the sub-file level is to transfer to the `Storage back-end` (Dropbox uses Amazon S3 as storage back-end) only those parts of files that have been modified since the last synchronization, saving traffic and storage costs.

The component responsible for partitioning files and calculating the hash values is the `Chunker`. The system could either use fixed-sized chunks or variable-sized chunks [19]. Either way, when a new file is added into the Workspace, all the new chunks will be indexed in the local database. If an existing file is updated, only the affected chunks will be indexed. Once the new chunks are indexed, the `Indexer` will then apply the appropriate source-based *deduplication* policy to transfer only the unique chunks to the `Storage back-end`. If deduplication is on a per user-basis, it suffices for the `Indexer` to compare the hashes of the new chunks with the ones in the local database. If some of the chunks already exist, only the new ones will be uploaded to the `Storage back-end`. If deduplication is cross-user, then the `Indexer` will have to ask first the synchronization service, `SyncService` for short, by sending to it the hash signatures of chunks. The `SyncService` will use the hash signatures to check for the existence

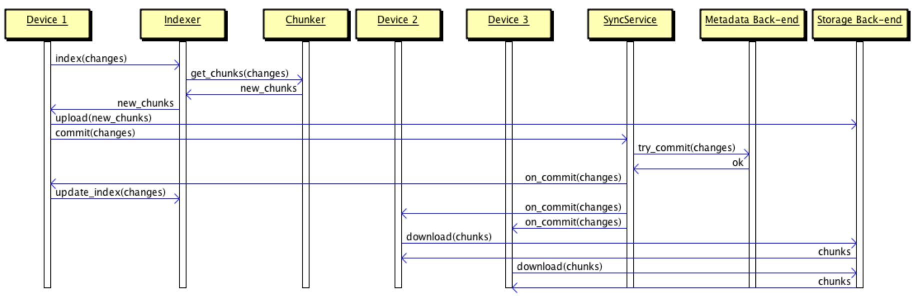


**Fig. 1.** Interaction between the components of sync engine for a Personal Cloud.

of chunks and then notify the `Indexer` to upload the missing chunks to the `Storage back-end`. Here we find the first functional dependency between components, as the efficiency of the `Chunker` determines the amount of data to transfer to the `Storage back-end`. Ideally, only those parts of the file that have been modified should be sent over the network, for which the choice of the chunk size is critical. Clearly, the best choice depends on both the number and granularity of changes. If a single character is changed in a file, a large-sized chunk will require transferring a lot of duplicate data, while a smaller chunk will offer a greater saving in network bandwidth. The downside of using small chunks is that it will increase the amount of metadata to be managed by the `SyncService`, directly impacting on the system's scalability. It is partly for this reason that an open architecture with a modular design like StackSync can facilitate the study of these issues and promote further advances in the future.

Once the unique chunks are successfully submitted to the `Storage back-end`, the `Indexer` will communicate with the `SyncService` to commit the changes to the `Metadata back-end`, which is the component responsible for keeping versioning information. The `Metadata back-end` may be a relational database like MySQL or a non-relational data store like Cassandra[6] or Riak[7], now frequently called NoSQL databases. Irrespective of the chosen database technology, the `SyncService` needs to provide a consistent view of the synced files. Allowing new commit requests to see uncommitted changes may result in unwanted conflicts. As soon as more than one user works with the same file, there is a good chance that they accidentally update their local working copies at the same time. It is therefore critical that metadata is consistent at all times to establish not only the most recent version of each individual file but to record its (entire) version history. Although relational databases process data slower than NoSQL databases, NoSQL databases do not natively support ACID transactions, which could compromise consistency, unless additional complex programming is performed. Since the nature of the `metadata back-end` strongly determines both the scalability and complexity of the synchronization logic, an open modular architecture like StackSync can reduce the cost of innovation, adding a great flexibility to meet changing needs.

Finally, when the `SyncService` finishes the commit operation, it will then notify of the last changes to all out-of-sync devices. The device that originally modified the

[6] `http://cassandra.apache.org/`
[7] `http://basho.com/riak/`

local working copy of the file will just update the `Indexer` upon the arrival of the confirmation from the `SyncService`. The other devices will both update their local databases and download the new chunks from the `Storage back-end`. Here we are assuming that an efficient communication middleware mediates between devices and the `SyncService`. This middleware should support efficient marshaling and message compression to reduce traffic overhead. Very importantly, it should support scalable change notification to a high number of entities, using either pull or push strategies. To deduplicate files and offer continuous reconciliation [7], recall that the local database at the `Indexer` must be in sync with the `Metadata back-end`, for which notification must be performed fast.

Fig. 1 illustrates the interaction between all the components of a file sync engine. Observe that not all the different components described in this section are present in the architecture of a Personal Cloud. In some architectures, our overall model could be simplified and one component could be responsible for many tasks. Some architectures can even lack some components. For example, ownCloud does not provide deduplication and chunking.

## 4 StackSync Reference Architecture

Personal Cloud storage services have become commonplace but currently, commercial existing offerings are proprietary, and open source alternatives are not mature enough. Researchers and practitioners interested in pursuing Personal Cloud questions have few tools with which to work. Clearly, the lack of research tools is unfortunate given that most fundamental questions are still unanswered: *What is the appropriate distributed architecture for a Personal Cloud? How do we manage versions so that versioning is flexible, correct and scalable? What types of service level agreements should a Personal Cloud service provide?* This is the basic reason why we have developed StackSync, an open source software framework that implements the basic components of what is commonly referred to as a Personal Cloud. All source code and data traces are available in `https://github.com/cloudspaces/stacksync`. Both the server and client code has been developed in Java: The client is a branch from the Syncany [15] project and the server is using our novel communication middleware called ObjectMQ [4] that will be presented in brief. The implementation currently has approximately $33,000$ lines of code, distributed in the following way: 1. *ObjectMQ middleware* — $2,100$ lines; 2. *StackSync client* — $24,400$ lines; and 3. `SyncService` — 5,800 lines.

Indeed, an important design decision of our reference implementation was to rely on a messaging middleware for communication. Since Personal Cloud storage services exhibit significant read-write ratios [2], we decided that the sync engine should support *persistent client connections, push-based notifications, and asynchronous and stateless interactions*. A message-oriented middleware fits well with these requirements, because of its support to *loosely coupled communication* among distributed components thanks to *asynchronous message-passing*.

StackSync is currently using RabbitMQ as its message broker and PostgreSQL as its default `Metadata back-end`. By default, we use OpenStack Swift as `Storage back-end`, though others are also possible. Indeed, the StackSync framework presents

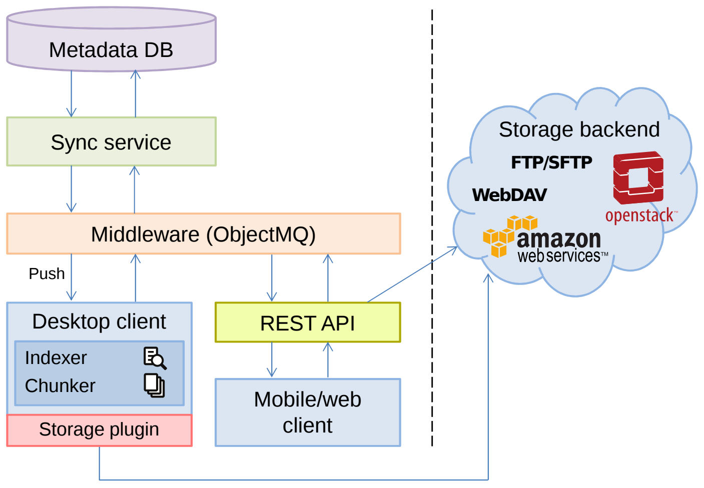


**Fig. 2.** StackSync architecture

extension hooks that enable others to replace the `Storage back-end`, the `Metadata back-end`, the message broker, the synchronization protocol or even the chunking and deduplication strategies.

An overview of our architecture with the main components is shown in Fig. 2. The StackSync client and the `SyncService` interact through the ObjectMQ middleware layer abstracting the RabbitMQ message broker. The `SyncService` interacts with the `Metadata back-end`. The StackSync client directly interacts with the `Storage back-end` to upload and download chunks. Finally, the REST API is used for accessing storage and sharing services from mobile and web clients. Security components like the authentication and authorization services are not included here.

## 4.1 StackSync Synchronization Protocol

Since file synchronization lies at the heart of any Personal Cloud service, we present a reference synchronization protocol that is available in the StackSync framework, whose main novelty is that it is based on a *message-oriented middleware* (MOM). In particular, MOM is ideally suited here for:

1. *Scalable change notification:* The high read-write ratio of Personal Cloud services makes it more appropriate to employ *one-to-many push communication* for quickly notification. Server-based protocols like this are required when replicas need a high degree of consistency [23].
2. *Asynchronous message dispatching:* Synchronization operations require significant server processing time for ensuring consistency. Decoupling message dispatching from message processing in these scenarios [18] is important for scalability reasons.
3. *Implicit load balancing:* When many consumers read from a queue, messages are distributed among them. This way, we can start multipe instances of the `SyncService`

to alleviate the load of the incoming request queue, *allowing to scale up and down very easily as computing needs change*.

Let us describe the overall synchronization protocol:

**Communication middleware** Our reference architecture relies on a high-performance message broker compatible with the Advanced Message Queueing Protocol (AMQP) [20] standard. In order to simplify the interaction with this messaging middleware, we have built the ObjectMQ communication middleware.

ObjectMQ is a lightweight remote object layer constructed on top of a messaging middleware compatible with AMQP [20]. We are combining RPC and MOM models to devise four MOM-RPC invocation abstractions: *two one-to-one calls* (synchronous, asynchronous) and *two one-to-many calls* (multi, event).

In general, objects receive method calls as messages in incoming queues, and then reply the results to clients in response queues. One-to-many calls are distributed using the appropriate AMQP Exchanges (broker message dispatchers). Recall that Exchanges determine message delivery and provide different delivery schemes, like *direct* (deliver this message to a particular queue) and *publish-subscribe* (deliver this message to all queues subscribed to a certain topic). Let us define the four calls:

- @async: This is an asynchronous non-blocking one-way invocation where the client publishes a message in the target object request Queue ($Q_{Request}$). By default, the client expects to receive no response and it is even not notified if the message was handled correctly.
- @sync: This is a synchronous blocking remote call where the client publishes a message in the target object request Queue ($Q_{Request}$), blocking until a response is received in its own client response queue ($Q_{Response}$). This call can be configured with a timeout and a number of retries to trigger the exception if the result does not arrive.
- @event: This is an asynchronous non-blocking one-to-many notification triggered by a server. The server triggers an @event by publishing the message in the Event fan-out Exchange ($E_{Event}$). Any client subscribing to this event in the server must bind their incoming Event Queue ($Q_{Event}$) to the server Exchange ($E_{Event}$).
- @multi: This is an asynchronous non-blocking one-to-many invocation from one client to many servers. The clients invokes the method in all servers by publishing an event in the Multi fan-out Exchange ($E_{Multi}$). All servers bind their incoming Request Queue ($Q_{Request}$) to the Multi Exchange ($E_{Multi}$).

The ObjectMQ communication middleware supports different compression and transport protocols (Kryo [3], Java Serialization, JSON). It also handles transparently all the error management and communication services on top of the MOM broker. We outline that the initial code of the StackSync client has been reduced in around $3,000$ lines of code after introducing our ObjectMQ communication library.

Finally, it is worth mentioning that our communication middleware could be easily replaced by a "traditional" RPC layer using a pull approach for change notification. Although our reference implementation is based on messaging and the push model, the StackSync framework is also open in the communication layer.

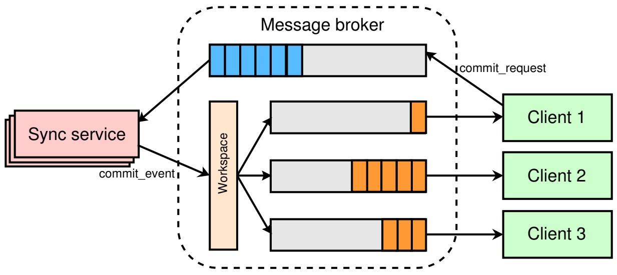


**Fig. 3.** Message broker communication flow

**SyncService and metadata back-end** The `SyncService` is a server-side component implemented as a remote object using the ObjectMQ communication middleware. This service directly benefits from the invocation abstractions offered by ObjectMQ.

In our current design (see Fig. 3), ObjectMQ is using a global request queue for the `SyncService`, a response queue for each device (`SyncService` Proxy), and a fan-out Exchange for each workspace. Each device will bind its request queue to the appropriate workspace Exchange to receive notification changes in this workspace. In any case, queue message programming is abstracted thanks to ObjectMQ, so that the protocol will be defined in terms of RPCs or method calls.

```
@RemoteInterface
public interface SyncService extends Remote {

    @SyncMethod(retry = 5, timeout = 1500)
    public List<ObjectMetadata> getChanges(Workspace workspace);

    @SyncMethod(retry = 5, timeout = 1500)
    public List<Workspace> getWorkspaces();

    @AsyncMethod
    public void commitRequest(Workspace workspace, List<ObjectMetadata> objectsChanged);

}
@event
public interface CommitEvent extends Event {

    public List<ObjectMetadata> objectsChanged getChanges();
}
```

**Fig. 4.** SyncService interface

In Fig. 4 we can see the interface definition of the `SyncService`. Clients can request the list of Workspaces they have access to with the *getWorkspaces* operation. Once the client obtains the list of Workspaces, it can then perform two main operations: *getChanges* and *commitRequest*. Furthermore, the client will be notified of changes by means of the event *CommitEvent*.

`getChanges` is a synchronous operation (@sync) that StackSync clients perform on startup. This is a costly operation for the `SyncService` as it returns the current state of a Workspace. Once the client receives this information, it registers its interest in receiving committed updates, i.e., *CommitEvent*s (@event) for this Workspace. From that point on, any change occurring on this Workspace will be notified to the client in a push style.

`commitRequest` is an asynchronous operation (@async) that clients employ to inform the `SyncService` about detected file changes in their Workspaces. This is a costly operation since it must guarantee the consistency of data after the new changes.

`CommitEvent` is triggered by the `SyncService` in an asynchronous one-to-many operation (@event) to all out-of-sync devices in the specified Workspace. This operation is only launched by the `SyncService` once the changes has been correctly stored in the `Metadata back-end`.

**Algorithm 1** Pseudocode of the commitRequest function in the SyncService

```
 1: function COMMITREQUEST($workspace, List < ObjectMetadata > objects\_changed$)
 2:     $commit\_event \leftarrow$ new instance of CommitEvent
 3:     for $new\_object$ in $objects\_changed$ do
 4:         $server\_object \leftarrow metadata\_backend.get\_current\_version(new\_object.id)$
 5:         if not exists $server\_object$ then        ▷ To commit the first version of the new object
 6:             $metadata\_backend.store\_new\_object(new\_object)$
 7:             $commit\_event.add(new\_object, \texttt{confirmed} = True)$
 8:         else if $server\_object.version$ precedes $new\_object.version$ then
 9:                                                   ▷ No conflict, committing the new version
10:             $metadata\_backend.store\_new\_version(new\_object)$
11:             $commit\_event.add(new\_object, \texttt{confirmed} = True)$
12:         else
13:                           ▷ Conflict detected, the current object metadata is returned
14:             $commit\_event.add(new\_object, \texttt{confirmed} = False, server\_object)$
15:         end if
16:     end for
17:     $trigger\_event(workspace, commit\_event)$
18: end function
```

Algorithm 1 reports the pseudocode of the `commitRequest` operation. When a `commitRequest` message is received in the global request queue, the ObjectMQ middleware will invoke the appropriate `commitRequest` method in the `SyncService`. This method then receives a proposed list of change operations in a concrete Workspace. For every change operation, it will then check if the current version of the object in the `Metadata back-end` precedes the change proposed by the client. In this case, the changes are (transactionally) stored in the `Metadata-back-end` and confirmed in the `CommitEvent`. If there is a conflict with versions, the `commitRequest` is set as failed and information about the current object version is added to the `CommitEvent`. The reason for adding the current object version to the `CommitEvent` is to piggyback the information about the "differences" between the two versions, such that the "losing" client can identify the missing chunks and reconstruct the object to the current version. As usual, in StackSync, a conflict occurs when two users change a file at the same time. This implies that the two clients will propose a list of changes over the same version of the file. The first `commitRequest` to be processed will increase the version number by one, but the second `commitRequest` will inevitably propose a list of changes over a preceding version, resulting in a conflict.

To resolve the conflict, the `SyncService` adopts the simplest policy in this case, which is to consider as the "winner" the client whose `commitRequest` was processed first. This way, the `SyncService` avoids rolling back any update to the `Metadata back-end`, saving time and increasing scalability. At the client, the conflict is resolved by renaming the "losing" version of the file to "... (conflicted copy, ..)".

Finally, the `CommitEvent` will be triggered to the Workspace AMQP Exchange, and it will be received by all interested devices in their incoming event queues.

As just elaborated above, note that the `CommitRequest` is an important operation in the sync service since it has to provide *scalable request processing, consistency, and scalable change notification*. Scalable request processing is achieved because the method is *asynchronous* and *stateless*. Multiple `SyncService` instances can listen from the global request queue and the message broker will transparently balance their load. Consistency is achieved using the transactional ACID model of the underlying `Metadata back-end`. Finally, scalable change notification to the interested parties is achieved using one-to-many push notifications (@event).

The `SyncService` interacts with the `Metadata back-end` using an extensible Data Access Object. Our reference implementation is based on a relational database although the system is modular and may be replaced easily.

**StackSync Client and Storage back-ends** The StackSync client is a local Java library that monitors the local folder and synchronizes it with a remote repository. As described before, the StackSync client now interacts with two main remote services: the `SyncService` through the ObjectMQ middleware, and with different `Storage back-end`s to upload and download chunks.

This decoupling of sync control flows from data flows implies that the client must authenticate with both entities. But it also enables a user-centric design where the client directly controls its digital locker or storage container. The synchronization protocol have been designed to put the load in the client side, whereas the `SyncService` just checks if the change is consistent, and then apply all changes proposed by clients.

Every client has a local database and a `Watcher` thread that monitors the state of the sync folders. On startup, the client must ask the server for its list of workspaces and its current state. From that point on, the `Indexer` will be notified by the `Watcher` of changes in the local folder and it will periodically send them using the *commitRequest* operation. Besides that, the client can receive commit events (*commitChanges*) from the server that will be immediately applied to the local workspace. Regarding potential conflicts due to offline operations, we provide similar policies than Dropbox, so that we create a copy of the conflicted document and let the user decide about this.

To save storage space and bandwidth, the `Indexer` uses source-based deduplication at the block level to store and transmit only a single copy of each duplicate block to the server. In our reference implementation, deduplication is applied on a per-user basis, as cross-user deduplication has been proven to be insecure [14]. This means that deduplication is carried out separately for each user, and therefore, data blocks of other users are not utilized to detect if an identical copy of the block is already at the server. Specifically, the client software maintains a chunk database to identify and locate duplicate data chunks. The database indexes each chunk by its fingerprint, which by default is the 20 bytes of its SHA-1 hash, though weaker fingerprints like the 32-bit Adler32 checksum may be used to reduce the index size. The database maintains the information of the chunks that form each file, so that whenever a file is modified, the client avoids transferring duplicate blocks to the server. Thanks to our message- oriented sync protocol, keeping such a database in sync with the server is inexpensive, as any changes on the central storage are advertised by means of asynchronous *CommitEvent*s as soon as they are performed.

StackSync's `Chunker` currently supports both fixed-sized and content-based chunking. Fixed-size chunking partitions a file into fixed-size blocks. Although fixed-size chunking does not perform well due to the boundary-shifting problem [10], it is useful to keep it as it incurs significantly lower computational costs that their content-based counterparts. For content-based chunking, StackSync supports Rabin-based chunking [19] and the Two-Threshold Two-Divisor (TTTD) chunking algorithm [10]. TTTD produces variable-sized chunks with smaller size variation than other chunking algorithms, leading to superior deduplication. In any case, the chunks are compressed before transmission using Gzip or Bzip2, albeit other compression algorithms can be easily plugged into the architecture.

Finally, and based on the original Syncany architecture, we provide an extensible plugin-based architecture for connecting to third-party `Storage back-ends`. We now provide plugins for Amazon S3, OpenStack Swift, Dropbox, WebDAV, SMB and FTP. It is straightforward to provide new plugins and we aim to leverage the open source community of Syncany for adding additional plugins to our platform.

## 5 Validation

In the evaluation of the reference implementation of StackSync framework, we set out to answer three basic questions: 1) *How much metadata overhead does the system need to support?* 2) *How much time is needed to have multiple devices in sync?*; and 3) *How does the system internally scale?*

### 5.1 Setup

To evaluate the reference implementation of StackSync, and further advances to come, we developed a tool written in Java to generate synthetic datasets. This tool can be downloaded from `https://github.com/cloudspaces/stacksync`, together with the generated datasets used in our evaluation, in order to make our experiments reproducible by others. Very succinctly, this tool generates an initial file system with subfolders and files (`TreeFSGenerator.java`), and then creates a trace with actions to change the initial file system over time (`DatasetGenerator.java`). These changes include the creation, modification, and removal of files in any subfolder of the initial file system. Alternatively, the tool enables the generation of the trace from a real file system. Either way, the chosen file system became the starting Workspace of StackSync clients in all experiments. The resulting trace was input into the class `DatasetExec.java`, which was in charge of performing the changes in both space and time in the local Workspace of clients.

The tool is highly configurable to enable the setting of several parameters such as the file system depth, file system total size, interarrival time between changes, etc. Precisely, the fact that interarrival time between actions is configurable is what makes it possible the evaluation of the scalability of any component at the server side, in our particular case, the synchronization service. To wit, by fixing an interarrival time of $0,01$ seconds, a researcher can evaluate the scalability of the system when the number of requests per seconds is $100$.

In our experiments, the size for each file was drawn uniformly at random from the range $[512\text{KB}, 4\text{MB}]$, which is the most common file size in many OS file systems [5]. The total size file system was set to $750$ MB with a folder depth of $3$ and $2$ subfolders per level. Over this structure, we run a trace with $1000$ changes to let the initial file system evolve over time. The trace altered a $5\%$ of the whole file system, approximately $37.5$ MB of file data.

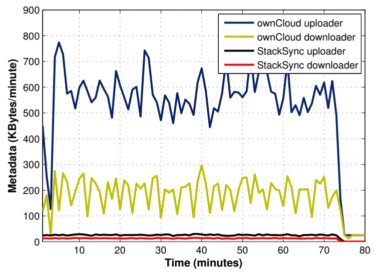
To decide how to change the files, we followed the same philosophy as in the dataset generator for deduplication evaluation introduced in [24], which currently supports three modification types: $B$ — the file is modified in the beginning by prepending some bytes; $E$ — the file is modified at the end; and $M$ — the file is modified somewhere in the middle. As in [24], we also supported combinations of these patterns, namely $BE$, $BM$, $EM$. Indeed, we used the same the change pattern of the *Homes* dataset (weekly snapshots of students' home directories), which is the dataset that represents better user behavior in Personal Cloud services. Very succinctly, the probability for a $B$ change was of $38\%$; for a $E$ change was of $8\%$, and for was $M$ change is of $3\%$. The rest of the probability mass was given to combination of these changes.

Finally, the testbed scenario included a server deployment with one front-end and several Desktop PCs acting as StackSync clients. The front-end was an OpenStack Swift deployment with one proxy node and three storage nodes. The proxy node also contained RabbitMQ, `SyncService`, and the PostgreSQL database acting as the `Metada back-end`. Let us review the node specs:

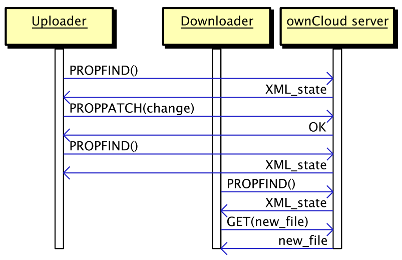
– Proxy node: Ubuntu 12.04.2 LTS(64 bits); Intel(R) Xeon(R) CPU E5-2407 @ 2.20GHz 4 cores/8 threads ; Memory: 12 GB.
– Storage nodes: Ubuntu 12.04.2 LTS(64 bits); Intel(R) Xeon(R) CPU E5-2403 @ 1.80GHz 4 cores/4threads; Memory: 8GB.
– Desktop PCs: Ubuntu 12.04.2 LTS(64 bits); Intel(R) Xeon(R) CPU E5-2403 @ 1.80GHz 4 cores/4threads; Memory:8GB.

### 5.2 ownCloud Vs StackSync

In this first experiment we want to compare the metadata overhead produced by ownCloud and StackSync. As stated in the related work, ownCloud is using a pull-based approach based on the WEBDAV protocol. The ownCloud client discover changes by continuously pulling the server with WEBDAV PROPFIND requests for the root user



(a) ownCloud vs StackSync metadata overhead



(b) ownCloud sync protocol

**Fig. 5.** ownCloud vs StackSync

folder every 30 seconds approximately. Each of these requests returns a XML file with a full list of the files and their associated metadata.

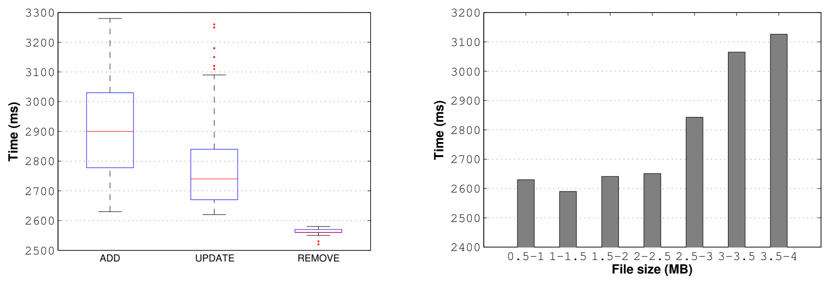
Since the PROPFIND request only returns the metadata of the files in the same level, it will not contain change information of files or folders in the levels below. This implies that a change in a subfolder will trigger recursive PROPFIND requests in all the levels of the path. When the change is located in the subfolder, the client then performs a GET request to retrieve the file and stay synced with the rest of the clients.

Due to obvious reasons, this behaviour can be highly inefficient since the amount of metadata exchanged between the server and the client to stay synced could be important (i.e. discover a new file stored in a five-level depth folder produces five PROPFIND requests). It is also important to take into account that each PROPFIND request over a folder involves to get the metadata of all the files inside this folder.

In Fig. 5 we can see the metadata overhead for ownCloud and StackSync when one node modifies files (uploader) and other must synchronise these changes (downloader). As we can observe in the figure, ownCloud's uploader node is producing for our experiment around 600-800 KBytes/minutes of metadata traffic, and the downloader is producing around 100-300 KBytes/minute of medatada traffic. This massive traffic is a direct consequence of the inefficient ownCloud pull protocol.

In Fig. 5 we can see the sequence diagram of ownCloud sync protocol. The node committing new changes (uploader) first obtains the current state (PROPFIND), then uploads the file (PUT), then requests the new change (PROPPATCH), and then rechecks with PROPFIND that the change was correctly applied. The rest of nodes (downloaders) will just ask for changes (PROPFIND) and then retrieve (GET) the entire file.

Note that PROPFIND and PROPPATCH are synchronous calls that are blocking the client and the server. Note also that if the change is not located in the root folder, it will require recursive PROPFIND requests for the entire path to the change for all clients. This is not an optimized protocol but a simple pull interface to WEBDAV. As we can see in Fig. 5, StackSync is producing an order of magnitude less overhead than ownCloud for the same experiment (around 10KBytes/Minute for both uploader and downloader nodes).



(a) Boxplots of synchronization time (b) synchronization time against file size

**Fig. 6.** Time to synchronize six clients.

Besides the inefficient metadata processing based on WebDAV pulling, data traffic is coupled to the same metadata server. Furthermore, ownCloud lacks any chunking, deduplication or even caching mechanisms. It just uploads or downloads the entire file if anything has changed.

ownCloud has devoted most efforts in developing the web document manager and web applications. They are providing a very simple and minimalist WEBDAV front-end to their web server. This could work for a single user (home repository) but the amount of useless traffic make it clearly infeasible for a large deployment.

## 5.3 Synchronization Time

Another interesting question to be examined is what is the delay experienced by users to have their devices in sync. To answer this question, we measured the time to synchronize six clients, measuring synchronization time for each type of Workspace modification, that is, the creation of a new file (ADD), and the modification (UPDATE) and removal of an existing file (REMOVE). The synchronization time was measured as the time elapsed after the modification was detected by the `Watcher` of the client that performed it until the local working copies of the other five clients were in sync. In the case of file creation and modification, this time included the delay incurred to upload and download the unique chunks from the `Storage back-end`, hosted in a local cluster running OpenStack Swift.

The results are depicted in Fig. 6(a). As can be seen in the figure, all the operations take only a few seconds to have all the clients in sync, even in the case of the ADD operation where an appreciable amount of time is taken up to access the `Storage back-end`. Because the REMOVE operation does not trigger any data flow to and from the `Storage back-end`, the synchronization time becomes a good estimator of the processing time incurred by the tandem ObjectMQ-`SyncService`. As shown in the figure, the time to reconcile a file removal in five clients is less than 2.6 seconds, which is quite good, taking into account that the `Metada back-end` is a relational database. As a boxplot enables to assess the dispersion of a given distribution, we gain important qualitative insights from Fig. 6(a). One important observation is that the distribution of the synchronization time for the UPDATE operation is right skewed, exhibiting synchronization times significantly greater than the median value of 2.75 seconds. This is especially noticeable by the significant number of UPDATE operations exceeding the upper whisker. This skewness is explained by the use of fixed-size blocks. A major problem with static chunking is that inserting even a single byte at the beginning of a file will shift all chunk boundaries [10], requiring the retransmission of the whole file to the `Storage back-end`. This defect can be simply corrected by applying any of the content-based chunking algorithms supported by StackSync like TTTD [10].

Since the time taken up by the ADD operation is affected by the file size, one interesting question is to assess how file size affects the synchronization time. Fig. 6(b) shows the synchronization time as a function of file size. As can be seen in the figure, the larger the file size, the longer the synchronization time. However, what is most interesting is the fact that the increase in time is only linear when file size is larger than 2.5 MBs, which indicates that for small files the time to transfer chunks from and to the `Storage back-end` is not significant compared with the time incurred by the tandem ObjectMQ-`SyncService`. Clearly, this poses the need for further research in file synchronization, since faster synchronization for small files can only be achieved by improvements in the `SyncService`. This emphasizes the value of. 6(a) a tool like StackSync, which fills an important unexplored niche in the cloud computing design space.

## 5.4 Scalability and load-balancing

Finally, we performed an initial experiment to assess the scalability and load-balancing of the StackSync platform. To this end, we performed stress tests generating 100, 200, and 400 commit requests per second from different clients. We wanted to measure how our service handles a high number of operations per second, and how the load can be balanced among different `SyncService` instances.

In this experiment, we used several system setups: 10 Desktop PCs generating each one 10 requests per second (100 commits/sec in the server), 20 Desktop PCs generating each one 10 requests per second (200 commits/sec in the server), and 20 Desktop PCs generating each one 20 requests per second (400 commits/sec in the server). The used trace was the same of the prior experiments but injected from different clients. For each system configuration, we executed the experiment using either one single-threaded or four single-threaded `SyncService` instances.

For each commit request, we recorded the time since the client sent out the commit request until the commit event confirming this changes was received in the client. Note that this time captured the entire life-cycle of the protocol, including message dispatching, time to process the operation in the `SyncService` (including access to the database), and finally the dispatching of the commit event to the requesting node.

In Fig. 7, we illustrate the time for 1 and 4 `SyncService`s under different loads: 100, 200, 400 commits/sec. Our service shows promising numbers in the processing of concurrent requests since it is able to handle a commit in less than 0.1 seconds with 4 instances when receiving a bulk of 200 requests/sec. This clearly indicates that balancing the load among multiple instances produces a significant performance boost. If we consider that all services are running in the same machine, our results show that the

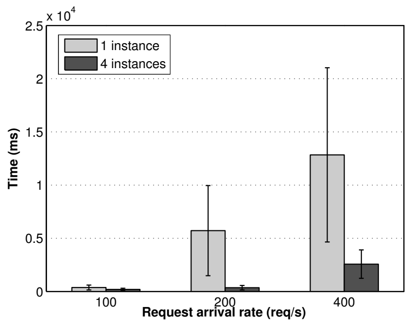


**Fig. 7.** Scalability and load balancing

bottleneck is in the transactional database rather than in the communication layer, i.e., ObjectMQ. The performance gains observed here open the way to future optimizations to fine tune the scalability of the overall service.

## 6 Conclusions

In this article, we have introduced StackSync, an open framework for Personal Cloud systems. Its architecture is highly modular, with each module represented by a well-defined API, allowing researchers to replace components for innovation in versioning, deduplication, live synchronization or continuous reconciliation, among other relevant topics. StackSync provides a reference implementation and useful tools for rapid prototyping and evaluation. The reference implementation of the file synchronization engine has been built on top of a lightweight MOM-RPC middleware, called ObjectMQ, whose one-to-one and one-to-many abstractions has considerably simplified the design of StackSync. This middleware is ideally suited to support push notification over persistent connections, which is critical for live synchronization.

StackSync is now under active development in the context of the FP7 CloudSpaces project with the collaboration of several partners. Further work includes adding privacy measures and supporting adaptive storage in collaboration with EPFL and EURECOM. Since today most of these systems are proprietary, relying on infrastructures that are invisible to the research community, we believe that StackSync will help to advance state of the art in Personal Cloud systems.

## 7 Acknowledgements

This work has been partially funded by the EU in the context of the project *CloudSpaces*: Open Service Platform for the Next Generation of Personal clouds (FP7-317555).

## References

1. Gartner consumer research: Personal cloud. `http://www.gartner.com/technology/research/personal-cloud/#`.

2. How we've scaled dropbox. http://www.youtube.com/watch?v=PE4gwstWhmc.
3. Kryo: Fast, efficient java serialization and cloning. http://code.google.com/p/kryo/.
4. Objectmq mom-rpc middleware. https://github.com/cloudspaces/objectmq.
5. N. Agrawal, W. J. Bolosky, J. R. Douceur, and J. R. Lorch. A five-year study of file-system metadata. *Trans. Storage*, 3(3), October 2007.
6. L. Ardissono, A. Goy, G. Petrone, and M. Segnan. From service clouds to user-centric personal clouds. In *Proc. of IEEE CLOUD*, pages 1–8, 2009.
7. S. Balasubramaniam and B. C. Pierce. What is a file synchronizer? In *Proc. of ACM/IEEE MobiCom*, pages 98–108, 1998.
8. L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: making backup cheap and easy. In *Proc. of OSDI*, pages 285–298, 2002.
9. I. Drago, M. Mellia, M. Munafo, A. Sperotto, R. Sadre, and A. Pras. Inside dropbox: Understanding personal cloud storage services. In *Proc. of ACM IMC*, pages 481–494, 2012.
10. K. Eshghi and H. K. Tang. A Framework for Analyzing and Improving Content-Based Chunking Algorithms. http://www.hpl.hp.com/techreports/2005/HPL-2005-30R1.pdf, 2005.
11. F. E. Gillett, C. Mines, T. Schadler, M. Yamnitsky, H. Shey, A. Martland, and R. Iqbals. The personal cloud: Transforming personal computing, mobile, and web markets. In *Forrester Research, BT Futures Report*, 2011.
12. Y. Hao, U. Irmak, and T. Suel. Algorithms for low-latency remote file synchronization. In *Proc. of IEEE INFOCOM*, pages 156–160, 2008.
13. A. Hari, R. Viswanathan, T.V. Lakshman, and Y.J. Chang. The personal cloud – design, architecture and matchmaking algorithms for resource management. In *Proc. of 2nd Usenix Hot-ICE*, pages 3–3, 2012.
14. D. Harnik, B. Pinkas, and A. Shulman-Peleg. Side channels in cloud services: Deduplication in cloud storage. *IEEE Security & Privacy*, 8(6):40–47, 2010.
15. P. Heckel. Syncany open source file synchronization. http://www.syncany.org/.
16. U. Irmak, S. Mihaylov, and T. Suel. Improved single-round protocols for remote file synchronization. In *Proc. of IEEE INFOCOM*, pages 1665–1676, 2005.
17. J. Langford. Multiround rsync. www.cs.cmu.edu/~jcl/research/mrsync/-mrsync.ps, 2001.
18. D. A Menasce. Mom vs. rpc: Communication models for distributed applications. *IEEE Internet Computing*, 9(2):90–93, 2005.
19. A. Muthitacharoen, B. Chen, and D. Mazières. A low-bandwidth network file system. *SIGOPS Oper. Syst. Rev.*, 35(5):174–187, 2001.
20. OASIS. Amqp: Advanced message queueing protocol. http://www.amqp.org/.
21. B. C. Pierce. Unison File Synchronizer. http://www.cis.upenn.edu/˜bcpierce/unison/.
22. T. Suel, P. Noel, and D. Trendafilov. Improved file synchronization techniques for maintaining large replicated collections over slow networks. In *Proc. of ICDE*, pages 153–, 2004.
23. A. S. Tanenbaum and M. Van Steen. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
24. V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok. Generating realistic datasets for deduplication analysis. In *Proc. of USENIX ATC*, pages 24–24, 2012.
25. A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, Australian National University, Dept. of Computer Science, June 1996.
26. P. Windley. From personal computers to personal clouds. http://www.windley.com/archives/2012/04/from_personal_computers_to_personal_clouds.shtml, 2012.