StackSync: Bringing Elasticity to Dropbox-like File Synchronization

Pedro Garcia Lopez
Universitat Rovira i Virgili
Tarragona, Spain
pedro.garcia@urv.cat

Marc Sanchez-Artigas
Universitat Rovira i Virgili
Tarragona, Spain
marc.sanchez@urv.cat

Sergi Toda
Universitat Rovira i Virgili
Tarragona, Spain
sergi.toda@urv.cat

Cristian Cotes
Universitat Rovira i Virgili
Tarragona, Spain
cristian.cotes@urv.cat

John Lenton
Canonical Ltd.
London, UK
john.lenton@canonical.com

ABSTRACT

The design of elastic file synchronization services like Dropbox is an open and complex issue yet not unveiled by the major commercial providers, as it includes challenges like *fine-grained programmable elasticity* and *efficient change notification* to millions of devices. In this paper, we propose a novel architecture for file synchronization which aims to solve the above two major challenges. At the heart of our proposal lies ObjectMQ, a lightweight framework for providing programmatic elasticity to distributed objects using messaging. The efficient use of *indirect communication*: i) enables *programmatic elasticity* based on queue message processing, ii) simplifies change notifications offering simple *unicast* and *multicast* primitives; and iii) provides transparent *load balancing* based on queues.

Our reference implementation is StackSync, an open source elastic file synchronization Cloud service developed in the context of the FP7 project CloudSpaces. StackSync supports both predictive and reactive provisioning policies on top of ObjectMQ that adapt to real traces from the Ubuntu One service. The feasibility of our approach has been extensively validated with an open benchmark, including commercial synchronization services like Dropbox or OneDrive.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Client/Server; H.3.4 [**Information Storage and Retrieval**]: Distributed Systems

General Terms

Design, Performance, Measurements

Keywords

Cloud Computing; Middleware; Elasticity; Storage

Middleware'14 , December 08-12 2014, Bordeaux, France
Copyright 2014 ACM 978-1-4503-2785-5/14/12 ...$15.00
http://dx.doi.org/10.1145/2663165.2663332

1. INTRODUCTION

In the last years, we have witnessed a rush of Personal Cloud Storage services offering file synchronization to millions of users. In this line, Dropbox [1] has achieved massive scalability thanks to a decoupled architecture that separates control flows (Dropbox sync servers) from data flows (Amazon S3 Object Storage).

While the elasticity of Cloud Object Storage Services like Amazon S3 is ensured, the design of elastic and scalable file synchronization protocols is complex [2]. Among the major challenges, we outline the following two issues: *fine-grained programmable elasticity* and *efficient change notification* to millions of users.

The first challenge is related to the observation that scaling up some types of cloud applications is not straightforward using traditional VM resource utilization metrics (CPU, RAM, etc.) [3], because, for instance, they are not CPU or memory intensive, but I/O bound, as is the case for file synchronization [2]. In those cases, it is better to rely on metrics such as the average and message handling response times exhibited by VM instances to cope with the varying demand. This implies that fine-grained elasticity management components must be built for the synchronization service as argued in the paper.

The other challenge is that the high read-write ratio of file syncing services makes it more suitable to make use of one-to-many push communication for rapid notification. Analogously, to efficiently maintain the consistency of files, any change performed elsewhere must be advertised as soon as they occur to reduce conflicts [1], in particular, when a file is susceptible to be modified by more than one client at the same time. This requires the file syncing service to operate as quickly as possible to commit changes, along with an efficient notification service to inform clients about file mutations.

To face the above challenges, we propose a novel architecture for elastic file synchronization. The major contributions of our work are:

1. ObjectMQ: a lightweight framework for providing programmatic elasticity to distributed objects using message queues as their underlying communication middleware. The efficient use of indirect communication in our middleware removes the need for pre-processing client stubs for scaling out and down, it provides transparent load balancing mechanisms based on queues, it simplifies one-to-many communications, and it enables flexible programmatic elasticity based on queue message processing.

2. StackSync: an elastic file synchronization architecture decoupling metadata and data flows in structured and object storage services. StackSync implements predictive and reactive provisioning policies on top of ObjectMQ that adapt to real traces from the Ubuntu One service. Furthermore, the ObjectMQ unicast and multicast communication primitives have considerably simplified the code of the synchronization protocol. It also enables efficient change notification in a transparent way on top of the underlying messaging service.

3. StackSync has been extensively tested using real traces from the Ubuntu One system to validate its elasticity and efficient use of resources. Furthermore, we extended an open benchmark [4] for Personal Clouds which provides trace generators and test scripts. Using this benchmark, we compared our service with Dropbox, Box, OneDrive, and Google Drive. StackSync is a stable open source project after two years of development that is being used in several public institutions and data centers.

The rest of the article is structured as follows. We review related work in Section 2. We introduce ObjectMQ in Section 3. Section 4 describes the StackSync architecture. We evaluate our reference implementation in Section 5 and conclude in Section 6.

2. RELATED WORK

Elasticity is the ability of a distributed application to dynamically augment or reduce its use of computing resources, to preserve its performance in response to varying workloads. In this line, elasticity is important in cloud settings for coping with demand but also to reduce costs by avoiding over-provisioning and using only the required resources.

Programmatic elasticity has been recently proposed [3] as a mechanism to provide fine-grained adaption to a group of distributed applications where traditional cloud elastic services (Amazon Auto Scaling) are not enough. In such applications like key-value stores, distributed lock managers and consensus protocols, the externally observable resource utilization metrics (CPU, RAM, etc.) are insufficient to achieve an efficient use of resources.

In this line, ElasticRMI [3] is a framework for engineering elastic object-oriented distributed applications. It follows the RMI scheme and masks the low level details of elasticity like monitoring, load balancing and self-adjusting mechanisms. Our ObjectMQ middleware achieves many of the goals of Elastic RMI but using Message Queues as the underlying infrastructure. This simplifies the overall architecture since load balancing is provided by the messaging middleware and we avoid the use of custom load balancing and leader election mechanisms. Furthermore, the indirect communication middleware moves load balancing to the server side and thus avoid preprocessors in client stubs. Finally, we provide additional services like one-to-many communications (@multi), flexible programmatic elasticity based on queue message processing, and persistent message handling to cope with bursts of demand.

We must outline that the use of MOM-RPCs and queued method invocations is not the novel contribution of our middleware. Other approaches like [5], [6] already rely on such invocations mediated by message queues. Furthermore, existing projects are using message queues as a load balancing mechanism in web applications [7].

There have been some previous attempts in the literature to combine message brokers and remote objects. We can outline [8], [9], and [10] as prominent examples of previous research. But again, they are not offering the novel call abstractions and elastic middleware that we provide in ObjectMQ.

Applications servers like JBoss also offer RMI Clustering and load balancing services. But again, like [3] they rely on dynamic client stubs that require preprocessing and updates by the server. The client stubs must know the IP addresses of all available server nodes, the algorithm to distribute load across nodes, and how to failover the request if the target node is not available. With every service request, the server node updates the stub interceptor with the latest changes in the cluster.

With respect to **"live" file synchronization**, little is known about the design and implementation of commercial systems. According to a recent characterization of Dropbox [1], file synchronization is built upon third-party libraries like *librsync*. While the specific role of this library is to calculate *deltas* values (the differences between the immediate previous version), the exact details of the file syncing protocol are still very murky, including metadata organization and consistency, and the failure recovery method. The same argument applies to the rest of Personal Clouds like Box, Google Drive, etc., whose file sync protocols have not been reversed engineered yet.

Either way, massive file synchronization protocols like Dropbox [2] and many others seem to share two major architectural decisions: (i) *decoupling control flows from data flows* and (ii) *push-based change notification*. In this line, Dropbox relies on Amazon S3 for their data flows and it uses its own massive private Cloud for the control flows. Furthermore, the Dropbox client keeps continuously opened a TCP connection to a notification server, used for receiving information about changes performed elsewhere.

Our middleware uses messaging for providing both programmatic elasticity of file synchronization and scalable change notification. Messaging is highly suited for this kind of communications because synchronization operations require significant server processing time for ensuring consistency. In these scenarios, decoupling message dispatching from message processing is key for scalability reasons [11]. During high demand peaks, message queues can temporarily store incoming messages, so that the processing components cannot be able to saturate the resources of the server side (e.g., database). The processing of events can then be controlled to never overwhelm the available resources.

3. OBJECTMQ

ObjectMQ is a framework which provides programmatic elasticity to distributed objects using a message queue system. In Fig. 1, we can see the basic architecture of our middleware, from left to right:

- **Client Stub:** It allows to call a remote object by utilizing the MOM communication layer. To make a remote call, the stub sends a message to a queue where the remote object is subscribed. Further, every stub has its own queue to receive responses from the server side.

- **MOM System:** It is the communication layer between stubs and skeletons. Every stub has its own queue to receive replies from remote objects. Fig. 1 shows the two types of queues a remote object is subscribed to. Specifically, the uppermost queue is a *global* queue shared among all the different remote objects. The lower queues correspond to the *private* queues where each individual object is listening to incoming calls.

- **Remote Objects:** They are remote objects that listen to the queues and execute RPCs. To add a remote object instance into the system, our middleware provides the method: *bind(oid, remoteObject)*, which binds a particular object instance with the identifier *oid*. Internally, ObjectMQ will create a

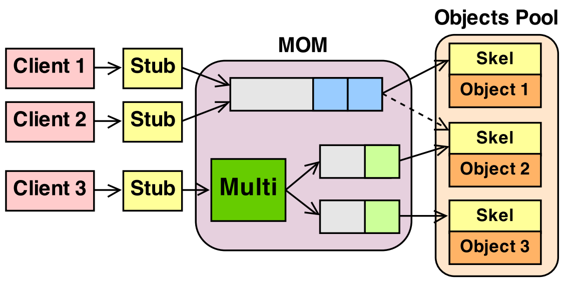


Figure 1: ObjectMQ architecture.

queue called *oid* where the *remoteObject* will be able to listen for new RPCs. If the queue already exists, the new instance will be simply subscribed to the queue. This binding mechanism will help scale out the system by dynamically creating new objects and subscribe them to a particular named queue, with the MOM system providing automatic load-balancing to all the objects subscribed to the queue. Due to the fact that only one of the subscribed remote objects can consume a specific message, i.e., the same message is not delivered to any other object, a separate private queue for every object is needed to support multicast.

Fig. 1 also illustrates the two types of remote invocations supported by ObjectMQ: *unicast* and *multicast*. Unicast invocations, issued by Client1 and Client2 in this example, are processed through the global queue. For this type of call, the MOM system will deliver the RPC to the first remote object that is idle. In multicast invocations, issued by Client3 in this example, the same RPC will be sent to all the private queues bound with the same *oid*, i.e., Multi(N queues, 1 object per queue). More technically, as our current implementation of ObjectMQ is built over the AMQP protocol [12], we simply use a type of *exchange*[1] called *fanout exchange* to support multicast. This type of exchange broadcasts all the messages it receives to all the queues that have been bound to a specified name *oid*.

The major building blocks of our architecture are: (i) a lightweight communication layer with small stubs and skeleton; (ii) novel communication primitives offering stateless one-to-one and one-to-many synchronous and asynchronous invocations, and (iii) an extensible provisioning model enabling third parties to create their own policies controlling the size of server object pools.

3.1 Communication Layer

Our major aim is to create a minimalist communication layer delegating complex communications to the messaging services. The programming model must be simple and it must completely hide queue and message management from developers. Our middleware avoids any stub compilation or preprocessing phase thanks to the use of dynamic stubs. Although we are inspired in Java RMI, we decided to create our own naming service and method decorators in order to simplify the overall communication model.

Our middleware delegates as much responsibilities as it can to the underlying MOM system. The underlying MOM system will be the responsible for balancing load while avoiding the loss of messages. Further, it will help us implement a naming service for the objects. Inspired by Java RMI, ObjectMQ also provides the methods *bind*, to bind a remote object to a specified name, and *lookup*, to return the remote reference bound to a given name. However, instead of using a centralized naming registry, we will use the queues to bind objects with their identifiers. As a result, whenever a stub wants to interact with a remote object, it will not need to look up the registry. Instead, it will suffice to know the name of the queue where it wants to send a message. Let us explain these functions specified in the `omq.Broker` class:

- *Broker.bind(oid, remoteObject)*: A call to this method binds a remote object with the identifier *oid*. Once done, the object will be ready to receive RPC requests. If necessary, a queue named *oid* will be created to receive unicast invocations. It will also create a unique private queue to receive multicast requests.
- *Broker.lookup(oid, aClass)*: An invocation to this primitive will generate a Proxy object for the class *aClass*. This Proxy will be used to submit messages to the queue named *oid* and receive responses in the private queue of the client.

Observe that binding more than one object with the same identifier also means that the load from the clients will be evenly distributed among multiple remote objects. This will help us to scale up and down the service by adding and removing remote object instances dynamically. In this case, there is no need to modify client stubs and they do not need to be aware of changes in the pool of remote objects offering a service.

Let us show a simple HelloWorld example:

```
@RemoteInterface
public interface HelloWorld extends Remote {
    @AsyncMethod
    public void helloWorld();

}

Broker broker = new Broker(environment);
helloServer = broker.bind("hello", new HelloServer());

helloClient = broker.lookup("hello");
helloClient.helloWorld();
```

Figure 2: ObjectMQ HelloWorld example.

As shown in the figure, developing remote objects using ObjectMQ is very simple. We omitted here the connection parameters of the Broker object referring to the location of the messaging service.

Since we aim to create a robust but very lightweight middleware, we do not provide shared state or consistency mechanisms between distributed objects. If many servers with the same identifier want to maintain consistent shared state, they should rely on a database or consistent data store. We consider that consistency is not responsibility of our middleware and that other services are ideally suited to this end. We also want to avoid any implicit or transparent state between servers and prefer to bet on a simple stateless model.

3.2 Communication Primitives

In the development of our communication abstractions, we decided to treat the local and remote entities separately, following the well known recommendation of Waldo et al. [13]. In ObjectMQ, remote object transparency is not desirable, because developers should be aware of when they are using remote or local entities

[1]In the AQMP parlance, an *exchange* can be viewed as a mailbox, distributing copies of the message to one or more queues according to specified bindings.

to program in a way that reflects the indeterminacy and concurrency constraints inherent in the use of remote objects. For this reason, ObjectMQ offers explicit mechanisms using method decorators to define method invocation primitives. In particular, we offer three main invocation abstractions: *asynchronous*, *synchronous*, and *multi-calls*. Let us define the three calls:

- @AsyncMethod: 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.

- @SyncMethod: 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.

- @MultiMethod: This is a one-to-many invocation from one client to many servers. This call can also be combined with @AsyncMethod or @SyncMethod. The former produces a non-blocking multiple invocation to many servers, whereas the latter produces a blocking multiple invocation that collects the results received from many servers in a determined timeout.

On the one hand, asynchronous invocations fit seamlessly with the underlying asynchronous messaging layer. They reduce the burden of handling messages and queues and do not impose additional overhead in the communication. On the other hand, synchronous invocation imply that the proxy will block during a timeout to wait for the result. Synchronous calls in our model must traverse an intermediary (messaging server) that is not needed in direct client-server models like Java RMI. This may impose a small communication overhead since messages must travel through the queues of server and client objects. The benefit is that we delegate communication to the messaging layer, so the server object in ObjectMQ cannot be saturated like in Java RMI, because our server layer only handles the messages the server can process.

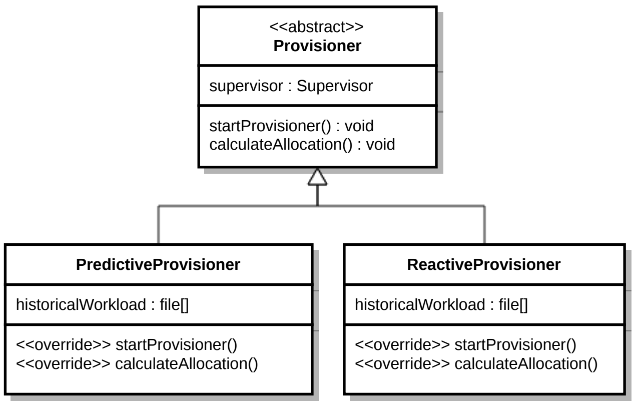
We also offer a new one-to-many communication primitive (@MultiMethod). In our case, when many servers listen in the same identifier (queue) they can receive group calls in a Multi queue. Since we rely on the one-to-many communication services of the messaging layer, the system is very efficient and achieves good performance numbers (the ones provided by the messaging layer). This abstraction has proven to be very useful for group communication and it can be combined very easily with the previous abstractions (@AsyncMethod, @SyncMethod).

Finally, an interesting advantage of one-to-many communications is the invocation of methods in a dynamic group of servers that can grow or shrink due to elasticity decisions. Let us explain how our middleware handles programmatic elasticity.

3.3 Programmatic Elasticity Framework

We have created an extensible framework that allows third-parties to create their own provisioning policies of server object pools. Our model follows a Master/Slave architecture where the `Supervisor` represents the centralized Master entity that takes care of enforcing `Provisioner`'s policies by launching or removing server objects in `RemoteBroker` Slave servers.

The `Provisioner` interface is the hotspot or extensible hook in our framework that must be inherited to offer a new provisioning policy. For example, in Fig. 3, we can see the predictive and reactive provisioners that we will explain in the next section. A Provisioner may use information from the HasObjectInfo introspection class to take decisions on server object provisioning. For example, it can observe that messages are not being processed at the adequate speed and ask for another server instance. Or decide that one server is idle and decide to suppress it.



<<abstract>>
Provisioner
supervisor : Supervisor
startProvisioner() : void
calculateAllocation() : void
PredictiveProvisioner
historicalWorkload : file[]
<<override>> startProvisioner()
<<override>> calculateAllocation()
ReactiveProvisioner
historicalWorkload : file[]
<<override>> startProvisioner()
<<override>> calculateAllocation()

Figure 3: Class diagram of the provisioning framework.

In general, a provisioner will propose a number of server objects required to handle the demand. The `Supervisor` is the responsible entity of enforcing the provisioning policy. It will launch (spawn) or remove (delete) server objects in RemoteBrokers. `RemoteBrokers` are ObjectMQ servers that can launch or shutdown remote object servers. The Supervisor uses multi call abstractions with RemoteBrokers for fault tolerance and introspection information. It periodically ask them about the state of their object servers and it maintains this information updated in HashObjectInfo object for provisioners. If the Supervisor detects that one server failed or that the number of servers is not the one required by the supervisor, it will then spawn or shutdown object servers. Of course, other technologies may be used to spawn or shutdown available servers such as Apache Mesos.

One interesting advantage of our framework is that adhoc policies can be designed depending on the target application. This offers a more fine-grained approach than the traditional coarse grained elasticity offered by cloud providers. We could also use variable like CPU load or memory, but we can also adapt to message processing time in queues offered by our middleware. If we want to enforce a determined processing time per server in an application, we can easily design an adhoc provisioner to this end.

3.4 Fault Tolerance

Unlike RMI or ElasticRMI where objects reside in main memory and any crash could lead to receive no response, ObjectMQ does not lose any information piece. For instance, if a remote object falls during a remote operation, this operation will be dispatched to another server instance. In this way, no remote invocations can be lost. This happens because every message sent to a remote object will be stored in the queue system until the object sends an ACK stating that the operation has finished. This occurs even if the operation is asynchronous. By using this approach, the message system can also know which instances are busy or not to balance the load.

Another important property is that, when a remote object crashes, it can be monitored by a `Supervisor`. Every second, the `Supervisor` asks the `Brokers` if they have a particular instance using a multicall. If some of the objects have failed, the `Supervisor` will then react by remotely binding a new object to any Broker missing that object.

Also, the `Supervisor` can crash. If this occurs, it will be impossible to know which objects are alive or not. To address this issue, every `Broker` in the system will periodically check if the `Supervisor` is up and running. Whenever the actual `Supervisor` crashes, a leader-election algorithm will be called using the unique identifier of the `Brokers`. Finally, to tolerate `Broker` failures, the messaging system can be instrumented to store all the messages present in the queues, so that when the system is restarted, the unprocessed messages can be recovered. Either way, high availability can be achieved by using clusters of messaging brokers.

Finally, we must outline that we have a stable implementation of ObjectMQ in Java `https://github.com/cloudspaces/objectmq` that uses AMQP protocol [12] and RabbitMQ as the messaging middleware. ObjectMQ supports different transport protocols (Kryo [14], Java Serialization, JSON) and it also handles all the error management and communication services on top of the MOM broker. Our architecture is generic so that we could use other cloud scalable messaging services such as Amazon SQS or Microsoft Service Bus.

4. STACKSYNC: ELASTIC FILE SYNCING SERVICE

Here we describe StackSync, our open source implementation of a Personal Cloud. We put the emphasis on the file syncing protocol and how it can be made elastic by letting ObjecMQ handle all the low-level mechanics, though our current implementation has all the software components to run a Personal Cloud.

At a very high level, its architecture is similar to a three-tier Web application, where the queues are the presentation/load balancing tier, the syncing service is the business logic tier and the metadata DB is the only stateful tier.

As any Personal Cloud service, such as Dropbox, Google Drive, and OneDrive, StackSync is characterized by two main components: a front-end client that runs on user devices, and a back-end service that has two main functionalities: storage of the user files, typically hosted in huge data centers, and the management of the metadata associated with the files, including versioning information, attribute change times, last modification, etc. Following the same approach as Dropbox, StackSync decouples metadata from the storage flows, splitting the back-end service into two separated components: the `Storage back-end`, which hosts the user files and objects, and the `Metadata back-end`, which is responsible for managing the file syncing metatada. The `Metadata back-end` may be a non-relational data store like Cassandra[2] or Riak[3]. However, we opted for a relational database to benefit from the ACID semantics, and this way simplify the maintenance of consistency. At the time of this writing, StackSync utilizes PostgreSQL as the `Metadata back-end` and OpenStack Swift object storage as the `Storage back-end`, though others are also possible. In fact, StackSync presents extension hooks to ease the replacement of the `Storage back-end`, the `Metadata back-end`, as other components like the message broker, the synchronization protocol or even the chunking and deduplication strategies.

The key component that interacts with the `Metadata back-end` is the file syncing service, referred to as `SyncService` for short in the rest of the paper. This service processes the commit requests from the clients. Mainly, it checks if the proposed changes by the clients preserve consistency, and then applies all the changes in the affirmative case.

[2]`http://cassandra.apache.org/`
[3]`http://basho.com/riak/`

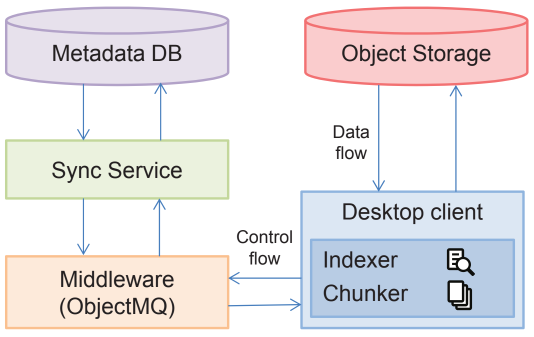


Figure 4: StackSync architecture.

One desirable property of the `SyncService` is that it was elastic. As observed by several measurements [1, 15, 16], Personal Cloud services present strong diurnal seasonality. The workload typically peaks around noon every day and reaches its minimum level in the middle of the night. Hence, provisioning for the peak demand will result in excess of resources during off-peak phases, incurring both capital costs and operational costs in terms of energy and cooling.

However, since file synchronization is I/O-bound, rather than CPU- and memory-intensive, traditional resource utilization metrics such as CPU and RAM can be misleading. Actually, the CPU utilization of our current implementation of `SyncService` never surpasses 30%, irrespective of whether the workload is light or heavy. As one of the main driving forces behind "live" file syncing is to exhibit an optimal synchronization time, we will use the response time as SLA for the `SyncService` and the request arrival rate observed in the queues as a fine-grained metric. This is where ObjectMQ comes into play by providing a simple programming model that masks the low level details of adding elasticity to the `SyncService`.

An overview of our architecture with the main components is shown in Fig. 4. The StackSync client and the `SyncService` interact through the ObjectMQ middleware layer. The StackSync client directly interacts with the `Storage back-end` to download and upload chunks. The `SyncService` interacts with the `Metadata back-end` to commit changes. For clarity, security components such as the authentication and authorization services have not been depicted in the figure.

Source code and data traces are available in `https://github.com/cloudspaces/stacksync`. Both the server and client code have been developed in Java: The client is a branch from the Syncany [17] project and the server has been built upon ObjectMQ [18], our novel elastic middleware. The current implementation has approximately $33,000$ lines of code (LOC), distributed in the following way:

- ObjectMQ —$2,762$ LOC.
- StackSync client —$24,400$ LOC.
- SyncService —$5,800$ LOC.

4.1 StackSync Desktop Client

Personal Clouds usually provide desktop clients that integrate with the OS file explorer capabilities. The desktop client is a local Java library that monitors the local folder and synchronizes it with the remote repository. In our architecture, the client interacts with two main remote services: the `SyncService` through the ObjectMQ middleware and with the `Storage back-end` to keep the files up to date and in sync. This decoupling of sync control flows from storage flows implies that the client must be authenticated with both entities. But it also enables a user-centric design where the client directly controls its digital locker or storage container.

Every desktop client has a local database and a thread that monitors the state of the synced folders. We will refer to each of these folders simply as a `workspace`. As many other Personal Cloud services, internally, StackSync does not use of the notion of file, but rather operates on a lower level by splitting files into chunks of 512 KB, each treated as an independent object. Each chunk is identified by a fingerprint, which by default is the 20 bytes of its SHA1 hash. 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` only those parts of files that have been modified since the last synchronization, saving traffic and storage costs. It must be noted that deduplication is applied on a per-user basis, as cross-user deduplication has been proven to be insecure [19]. This means that deduplication is carried out separately for each user, and therefore, file chunks of other users are not utilized to detect if an identical copy of the block is already at the server.

Every time a change in any workspace is detected by the OS, the `Indexer` component will look up the local database to identify the affected chunks. Concretely, the `Indexer` will call the `Chunker`, which will partition the modified file into chunks and calculate the hash values for each chunk. Then, the `Indexer` will compare the hashes of the new chunks with those in the local database. If some of the chunks already exist, only the new ones will be uploaded to the `Storage back-end`. After uploading the unique chunks to the `Storage back-end`, the `Indexer` will communicate the changes to the `SyncService` using an asynchronous ObjectMQ call.

Besides that, the client can receive notifications of committed changes from the `SyncService` that will be immediately applied to the affected workspace. Thanks to our message-oriented file syncing protocol, keeping the local database in sync with the `Metadata back-end` is inexpensive, as any committed change is advertised as soon as possible by means of asynchronous notifications. In case of conflicting changes due to offline operations, we follow the same policy as Dropbox: we create a copy of the conflicted document and let the user decide about this.

A final remark is that the `Chunker` supports both fixed-sized and content-based chunking [20]. By default, it uses static chunking, splitting the files into chunks of 512 KB. Although static chunking does not perform well due to the boundary-shifting problem [21], it is useful to keep it as it incurs significantly lower computational costs that their content-based counterparts. In any case, the chunks are always compressed before transmission using Gzip or Bzip2, albeit other compression algorithms can be easily plugged into the system.

4.2 StackSync Synchronization Protocol

Because file synchronization lies at the heart of any Personal Cloud service, we introduce here the file syncinc protocol that is available in StackSync, whose novel feature is that it has been built upon the ObjectMQ middleware, which masks all the low-level mechanics of handling *persistent client connections, push-based notifications, and asynchronous interactions*. A middleware like ObjectMQ fits well with these requirements, because of its support to *loosely coupled communication* among distributed components thanks to *asynchronous message-passing*.

4.2.1 SyncService

The `SyncService` is a server-side component implemented as a remote object using ObjectMQ. This service directly benefits from the invocation abstractions offered by ObjectMQ. In fact, thanks to communication abstractions provided by ObjectMQ, the core of the `SyncService` algorithm can be described in a few lines of pseudocode, as shown in Algorithm 1.

As shown in Fig. 5, ObjectMQ is using a global request queue for the current design of the `SyncService`, a response queue for each device (`SyncService` Proxy), and a multi fanout for each workspace. Each user device will bind each request queue to the appropiate workspace to receive notification changes. 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);

}
public interface RemoteWorkspace extends Remote {

@MultiMethod
@AsyncMethod
void notifyCommit(CommitNotification notification);
}
```

Figure 6: SyncService interface.

In Fig. 6 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 the committed changes by means of `CommitNotifications`.

`getChanges` is a synchronous operation (@SyncMethod) that StackSync clients perform only on startup. This operation is costly 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 for this workspace. From that time onwards, any change occurring on this workspace will be notified to the client in a push style as a `CommitNotification`.

`commitRequest` is an asynchronous operation (@AsyncMethod) that clients use to inform the `SyncService` about detected file changes in their workspaces. This is a relatively cheap operation in terms of processing time, though it must still guarantee a consistent view of the affected files after the new changes.

Finally, the `SyncService` pushes a `CommitNotification` to all out-of-sync devices in the workspace by calling the asynchronous one-to-many operation (@MultiMethod) named as `notifyCommit` in the `RemoteWorkspace` interface. This operation is invoked by the `SyncService` only after the proposed changes has been correctly committed in the `Metadata back-end`. The `SyncService` interacts with the `Metadata back-end` using an extensible Data Access Object. Our current implementa-

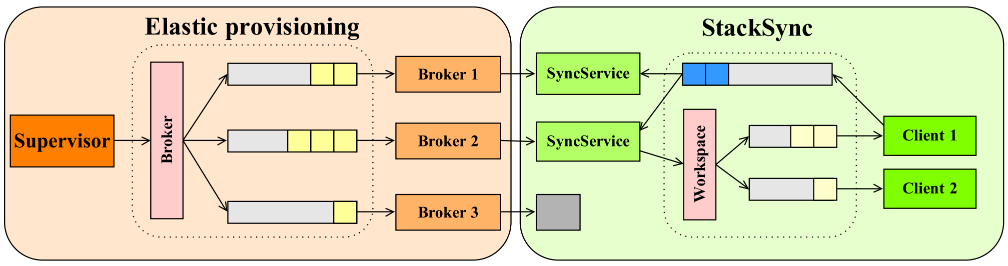


Figure 5: Message broker communication flow.

tion is based on a relational database, though the system is modular and may be replaced easily.

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

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

In Algorithm 1, we give the pseudocode of the `commitRequest` operation. When a commit request message is received in the global request queue, ObjectMQ calls the `commitRequest` method in the `SyncService`. This method receives a proposed list of changes for a concrete workspace. For every change operation, this method first checks if the current version of the object in the `Metadata back-end` precedes the proposed change. If the case, the change is persisted in the `Metadata back-end` and then confirmed in the `CommitNotification`. If there is a conflict with versions, the `commitRequest` is marked as failed and information about the current object version is added to the `CommitNotification`. The fundamental reason for adding the current object version to the `CommitNotification` 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. In StackSync, a conflict occurs when two users change a file at the same time. This means 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. Consequently, 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.

Finally, the `CommitNotification` is pushed to all interested devices in their incoming multicast queues.

As just discussed above, the `CommitRequest` is the most critical operation of the `SyncService`, since it involves *scalable request processing, consistency, and scalable change notification* in just a single operation. Scalable request processing is achieved because the method is *asynchronous* and *stateless*. Multiple instances of the `SyncService` can listen from the global request queue and the message broker will transparently balance their load, which allows for rapid elasticity. Consistency is achieved through the ACID semantics of the underlying `Metadata back-end`. Finally, scalable change notification to the interested parties is achieved using one-to-many push notifications (@MultiMethod).

4.3 Elastic File Synchronization

Here we show how an application developer can tap into ObjectMQ to enable elastic scaling in file synchronization. Existing commercial Cloud solutions typically fall back on observable resource utilization metrics such as CPU and RAM to drive scaling decisions. For a Personal Cloud system, however, these metrics are ill-suited, as one of the main driving forces behind "live" file synchronization is to guarantee a maximum synchronization time. This requires handling elasticity at the application level by exploiting the knowledge of the application workload, so that it can be utilized a more versatile set of scaling mechanisms.

In what follows, we show how a Personal Cloud system can benefit from the simple programming model of ObjectMQ to achieve elasticity in the file syncing protocol. Particularly, we adopt the model of Urgaonkar et al. [22] for dynamic resource provisioning, though many others could be chosen. The advantage of this model is that it makes use of both a predictive and reactive approach, allowing us to prove the versatility of ObjectMQ. The goal of the predictive method is to allocate resources on large time scales, of the order of days and hours, while the reactive approach is used for shorter time scales, such as seconds and minutes. This allows the system to correct prediction mistakes made by the predictive model, such as unpredictable "flash crowd" patterns. Actually, the predictive method is very useful for online file synchronization. As reported by several independent studies [1, 15, 16] , Personal Cloud systems exhibit strong diurnal and weekly patterns. This allows the predictive provisioning method to allocate servers well ahead of the expected workload peak, and dramatically reduce the odds for clients to experience degraded performance. Indeed, the effectiveness of predictive provisioning will be shown to be very high in our trace-driven experiments with the UB1 cloud-based file syncing service, confirming our intuition that predictive resource provisioning is ideally suited for Personal Cloud systems.

As we set out to enable elasticity for control flows in this work, we assume that the SLA is specified in terms of a suitable high percentile of the response time distribution. We denote this value as d. For instance, a SLA may specify that 95% of the commit requests should incur an end-to-end response time of no more than 5 seconds. As in [22], we assume that all the different instances of the `SyncService` run in homogeneous machines and model each synchronization server as a G/G/1 queuing system, to allow for an arbitrary arrival distribution and arbitrary service times. This enables our elastic scaling scheme to adapt gracefully to changes in the workload intensity caused by time-of-day effects, or even time-of-hour effects. By well-known formulae, the rate δ at which a synchronization server can process commit requests can be simply computed as:

$$\delta \geq \left[s + \frac{\sigma_a^2 + \sigma_b^2}{2(d-s)}\right]^{-1}, \tag{1}$$

where s is the average service time for a commit request, and σ_a^2 and σ_b^2 are the variance of interarrival time and the variance of service time, respectively.

Observe that d is known, while s as well as the variance of interarrival and service time σ_a^2 and σ_b^2 can be monitored online and adjusted correspondingly. By substituting these values into (1), we can obtain a lower bound on the request rate δ that can be serviced by a single server. Once the capacity of a single server is known, the number of required instances η to service a peak request rate of λ can be simply obtained as:

$$\eta = \left\lceil \frac{\lambda}{\delta} \right\rceil. \tag{2}$$

Depending on the value of η and the current number of instances of the `SyncService`, the `Supervisor` will decide to add or remove instances to preserve the performance in response to varying workloads. In practice, the decision of scaling up and down will be performed periodically, once every t time units, to avoid unnecessary oscillations. We will denote by $\lambda_{obs}(t)$ the actual arrival rate seen during the time interval t. Note that the value of $\lambda_{obs}(t)$ can be obtained very easily in our file-sync architecture, since all the commits are queued in a single request queue, as shown in Fig. 5.

4.3.1 *Predictive Provisioning*

This technique uses a workload predictor to anticipate the peak demand over the next time period, and then uses (2) to determine the number of instances that are needed to meet this peak demand. Concretely, the predictor estimates the peak demand that will be seen over the next period of T time units, at the beginning of each period. To do so, it maintains a history of the observed arrival rate for each time period t of duration T over the past several days. From the history, the predictor then derives the probability distribution of the arrival rate for that time period. The peak workload $\lambda_{pred}(t)$ for a particular period t is finally estimated as a high percentile of the arrival distribution for that period.

4.3.2 *Reactive Provisioning*

Even in the case that predictive provisioning was perfect, sudden spikes or “flash crowds" are unpredictable phenomena. To react to unforeseen events, reactive provisioning acts on shorter time scale to handle short term fluctuations. Basically, it compares the current observed arrival rate $\lambda_{obs}(t)$ over the past few minutes to the predicted rate $\lambda_{pred}(t)$. Specifically if $\frac{\lambda_{obs}(t)}{\lambda_{pred}(t)} > \tau_1$ or drop rate τ_2, then corrective action is necessary. In this case, it recomputes the number of instances by invoking (2).

5. VALIDATION

We divide the evaluation into two parts. In the first part, we evaluate the performance of the `SyncService` without auto-scaling and set out two basic questions: 1) *How much overhead does the system need to support?* and 2) *How much time is needed to have multiple user devices in sync?* In some of these tests, we compare StackSync against other popular Personal Cloud services, including Dropbox, Microsoft OneDrive, Amazon Cloud Drive, Google Drive and Box.

In the second part, we evaluate the ability of the `SyncService` to handle multi-time-scale variations seen in a real Personal Cloud workload. Concretely, we evaluate the effectiveness of the different provisioning techniques, as well as the fault tolerance of ObjectMQ.

5.1 Testbed

For all the experiments, we used the following testbed. The testbed included a front-end server and several desktop PCs acting as clients. The front-end was an OpenStack Swift deployment with one proxy node and 4 storage nodes. Unless otherwise noted, the proxy node hosted the `SyncService`, and the PostgreSQL database acting as the `Metadata back-end`. Let us review the node specs:

- The proxy node was equipped with Intel Xeon CPU E5-2407 and 12 GB RAM.
- The storage nodes had Intel Xeon E5-2403 processors and 8 GB RAM.
- The desktop PCs had Intel Core i5 processors and 4 GB RAM.

All the machines ran Ubuntu 12.04 except the desktop PCs that ran Debian 7.3. Finally, the software versions were OpenStack Swift (*Havana*); RabbitMQ 2.8.7 and PostgreSQL 9.1. Table 1 shows the software versions for the Personal Cloud desktop clients used in the evaluation.

5.2 Performance without Elasticity

5.2.1 *Setup and Benchmarking Tool*

To evaluate the system with the auto-scaling feature disabled, we developed a benchmarking tool to generate realistic workloads. We implemented this tool because we found no publicly available trace containing both the files and the history of modifications to those files that allowed us to evaluate our file-syncing service.

To determine the size of the files, we used the distribution presented in [16], a five-month study involving around $20,000$ users. Some of their conclusions were that the 90% of files are smaller than 4 MB and that updated files tend to be read sooner rather than later. Also, they noticed that most of files are read-only.

Imitating the real behavior of users, our tool creates a trace with 3 different actions: ADD representing the addition of a file, UPDATE signaling a modification, and REMOVE meaning the removal of the file from the workspace.

In order to determine the action to be performed to a file, we applied the Markov model proposed in [23]. In this model, each file can be in 4 possible states: `N` — new; `M`— modified; `U` — unmodified; and `D` — deleted. For each of those states, a set of probabilities govern the transition to the rest of states. To set up the transition probability matrix, we extracted the transition probabilities from the “*Homes*” dataset [23], which is the public trace that most resembles the user behavior in a Personal Cloud service.

Client name	Version
StackSync	1.6.4
Dropbox	2.6.33
Microsoft OneDrive	17.0.4035.0328
Amazon Cloud Drive	2.4.2013.3290
Google Drive	1.15.6430.6825
Box	4.0.4925

Table 1: Used Desktop Clients Version

To decide how to modify the files, we followed the same approach as in [23], which currently supports 3 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 [23], we also supported combinations of these patterns, namely `BE`, `BM`, and `EM`. For the transition probabilities, we used the change pattern of the "*Homes*" dataset: the probability for a `B` change was of 38%; for a `E` change was of 8%, and for a `M` change was of 3%. The rest of the probability mass was granted to combinations of these changes. We only applied these probabilities in files smaller than 4 MB, since more than 90% of the I/O requests are for these files, as just discussed above.

Our trace generator requires only 3 parameters: 1) initial number of files; 2) number of training iterations; and 3) number of snapshots. For our experiments, we set the initial number of files to 20, and the number of iterations and snapshots to 5 and 100, respectively. The resulting trace contained 940 ADDs, 72 UPDATEs and 228 REMOVEs. The ADD operations generated a total data volume of 535.41 MB whereas the UPDATEs only produced ≈ 14 KB. The average file size was of 583 KB. Fig. 7(a) plots the CDF of file size for our trace.

Finally, to compare StackSync against other popular Personal Cloud services, we modified the benchmarking tool developed by Drago et al. [4] to conduct trace-driven experiments with the output of our generator. More specifically, we adapted their tool to measure the overhead of the different file syncing protocols under a sequence of ADD, UPDATE, and DELETE operations using real content.

5.2.2 *Protocol Overhead*

In this test, we compared the protocol overhead of StackSync with the overhead of the commercial Personal Cloud services reported in Table 1. As in [4], we defined the overhead as the total storage and control traffic over the benchmark size, which was of 535.41 MB. For this experiment, our benchmarking tool considered each of the operations in the trace one at a time, to measure the overhead accurately. That is, the next operation did not start until the current one was successfully committed.

The results are shown in Fig. 7(b). As seen in this figure, Dropbox exhibits the highest overhead, sending up to 150 MB of additional unnecessary traffic. This results agree very well with other studies such as that of [16]. On the contrary, StackSync has a low overhead, *comparable to that of commercial Personal Cloud services*.

To gain a deeper understanding of the overhead, we prepared a new variant of this test. In this new variant, we grouped all the actions of the same type to generate 3 separate traces, and this way study the overhead per type of action. In this case, we run this experiment only for StackSync and Dropbox, mainly because Dropbox is the most popular and well-studied Personal Cloud in the literature. The results are depicted in Fig.7(c) for control traffic and Fig.7(d) for storage traffic.

	Batch size	Control	Storage	Total
Dropbox	5	8.30 MB	633.06 MB	641.36 MB
	10	5.13 MB	638.26 MB	643.39 MB
	20	3.28 MB	635.82 MB	639.10 MB
	40	2.23 MB	632.05 MB	634.28 MB
StackSync	5	2.14 MB	569.89 MB	572.03 MB
	10	1.58 MB	570.10 MB	571.68 MB
	20	1.37 MB	570.07 MB	571.44 MB
	40	1.25 MB	567.77 MB	569.02 MB

Table 2: Effect of File Bundling

As shown in Fig.7(c), Dropbox produces a huge amount of control traffic when adding new files, about 25 MB of unnecessary traffic, while StackSync only needs ≈ 3.2 MB. This indicates that control signaling is significantly much lighter than that of Dropbox. With respect to storage traffic, StackSync transferred a total amount of 565.63 MB to the `Storage back-end`, which is significantly smaller than the 660.32 MB of storage traffic incurred by Dropbox.

For the UPDATEs, StackSync is negatively affected by the fact that a modification of a few bytes requires to upload at least one chunk of 512 KB, incurring a huge overhead. Since Dropbox uses delta encoding [4], a specialized compression technique, it outperformed StackSync. It is important to note here that although the overhead for StackSync is apparently high, in practice, StackSync transferred only 5 MB and Dropbox 2 MB. Both values are relatively high compared with the amount of modified data (13.50 KB).

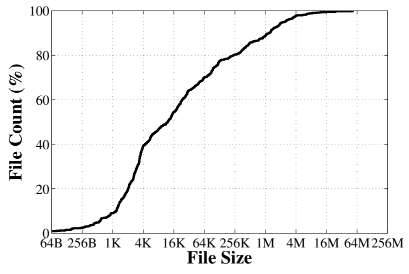
As the setup of these tests were not favorable to Dropbox, because the actions were performed one after another without benefiting from the Dropbox file bundling feature, we created a new test that performs more one action at the same time. Table 2 lists the results obtained after executing the trace for both Dropbox and StackSync with different batch sizes. Dropbox reduces traffic overhead in 30 MB but it continues to be much higher than the rest of the Personal Clouds.

5.2.3 *Synchronization Time*

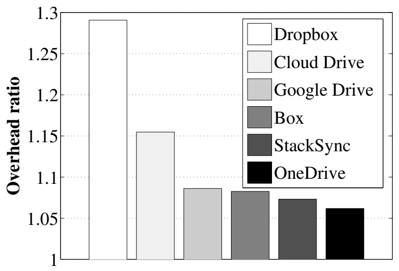
Another basic question to be examined is the delay experienced by users to have their devices in sync. To answer this question, we measured the time to synchronize 6 clients for each type of workspace changes, i.e., ADD, UPDATE and REMOVE. The synchronization time was measured as the time elapsed after 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 ADDs and UPDATEs, this time included the delay incurred to upload and download the unique chunks from the `Storage back-end`, hosted in our local cluster.

The results are depicted in Fig. 7(e). 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`.

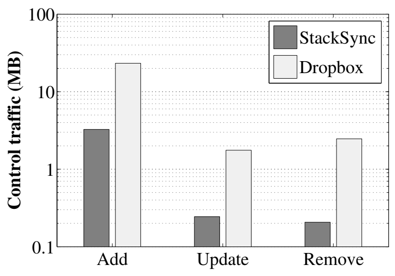
As shown in the figure, the time to reconcile a file removal in five clients is less than 2.6 seconds, which is quite good, assuming that the `Metada back-end` is a SQL database. As a boxplot enables to assess the dispersion of a given distribution, we gain important qualitative insights from Fig. 7(e). One key observation is that the



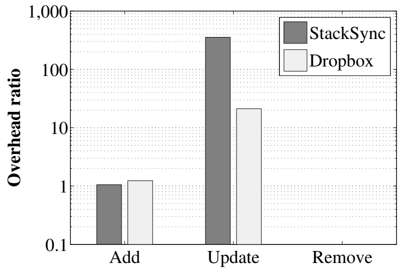
(a) File size vs file count distribution.



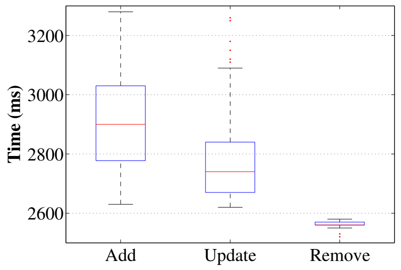
(b) Overhead



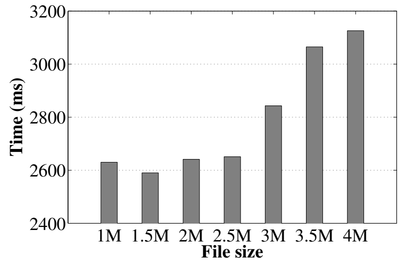
(c) Control traffic overhead



(d) Storage traffic overhead



(e) Boxplots of synchronization time



(f) Synchronization time against file size

Figure 7: Performance of StackSync file-syncing protocol.

distribution of the synchronization time for the UPDATE operation is right skewed, exhibiting file synchronization times significantly greater than the median value of 2.75 seconds. This is evidenced by the significant number of UPDATE operations exceeding the upper whisker. This skewness is explained by the use of fixed-size blocks, which suffer from the *boundary shifting problem* [21].

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. 7(f) 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`.

5.3 Performance of `SyncService` Auto-Scaling

In this section, we evaluate the performance of the `SyncService` when the ObjectMQ auto-scaling feature is enabled. To evaluate it, we conducted three experiments. The first experiment evaluates the behavior of the `SyncService` when both the predictive and reactive provisioning algorithms are in use. The second experiment repeats the first test but with the predictive method misestimating the workload pattern. Finally, the third test gauges the fault tolerance of ObjectMQ auto-scaling when the instances of the `SyncService` crash.

5.3.1 Setup

For all the experiments, we set the response time for committing a request to $d = 450$ msec. The `PredictiveProvisioner` was called every 15 minutes while the `ReactiveProvisioner` was invoked once every 5 minutes.

To compute the mean service time s and the variance of the service time σ_b^2, we conducted several offline measurements on the desktop PCs that hosted the instances of `SyncService` during the tests. The variance of interarrival times σ_a^2 was updated once every 15 minutes based on online measurements of the global request queue. The parameters can be found in Table 3.

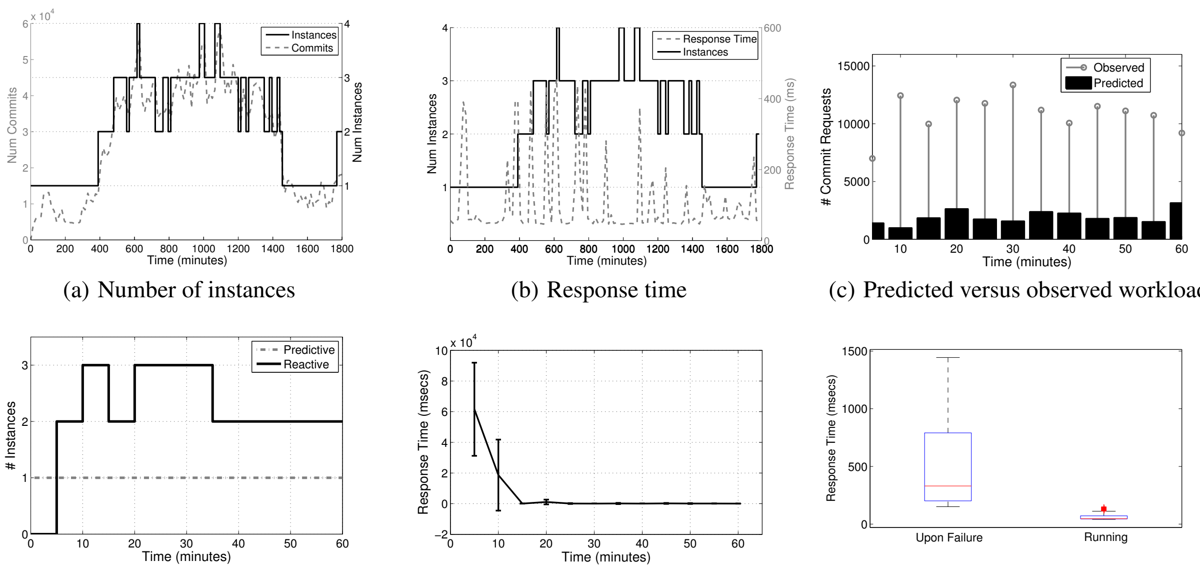
Parameter	Value
d	450 msec
s	50 msec
σ_b^2	200 msec
τ_1	20%
τ_2	20%

Table 3: Parameters for the UB1 Workload

Testbed. For all the experiments in this section, we used the same experimental testbed as in the first part of the evaluation. However, instead of deploying the `SyncService` as a single instance in the Proxy node, we added a pool of desktop PCs to host multiple instances of the `SyncService` as dictated by the `Supervisor`. Recall that the instances are server objects which are activated and passivated by the `Supervisor` through calls to the `RemoteBrokers`.

Workload. Instead of a synthetic workload as in previous works [3], we preferred to use a trace from a real system. More concretely, the workload was generated based on a (anomymized) traces from **Ubuntu One** (UB1), a popular Personal Cloud service operated by Canonical Ltd. which shut down on April 2014. The trace contains detailed information about the files uploaded to UB1 on November 2013.

Based on this trace, we created two new traces to drive our tests. The first trace contained the number of arrivals per second to the UB1 control servers over a one week period. The objective of this trace was to feed the predictive provisioner with a sufficiently large history to calculate accurate summaries. The second trace was the input trace for the experiments and only contained the number of commit request arrivals seen on day 8. This day was a typical day, meaning that the workload closely resembled that observed on the previous week. This enabled us to determine whether the prediction algorithm captured well the time-of-day effects and if it was able to



(a) Number of instances (b) Response time (c) Predicted versus observed workload

(d) Number of instances (only reactive) (e) Response time (only reactive) (f) Fault tolerance

Figure 8: Effectiveness of ObjectMQ auto-scaling for file synchronization.

assign sufficient capacity to the `SyncService` at all times. The exact workload for day 8 can be seen in Fig. 8(a). The peak demand for this trace was of $8,514$ commit requests per minute.

5.3.2 Both Reactive and Predictive Provisioning

In this test, we measured the effectiveness of ObjectMQ auto-scaling when both the predictive and reactive provisioning mechanisms are enabled. The workload for the experiment was that for day 8 after feeding the `PredictiveProvisioner` with 15-minute arrival rate summaries from the previous week.

The results are shown in Fig. 8(a) and 8(b). As seen in Fig. 8(a), the number of instances mimics the workload pattern at all times thanks to the accurate estimations of the predictive method. If we look at the resultant response times, plotted in Fig. 8(b), we can see that no commit request exceeds the targeted response time of 450 msecs. The presence of spikes indicate the moments of arrival and removal of instances. For example, if we consider the minute 600, the response time was high because the `Supervisor` was about to add a new instance to meet our SLA of 450 msecs. Overall, we can conclude that ObjectMQ auto-scaling is clearly effective.

5.3.3 Misprediction of Predictive Provisioning

We repeated the same experiment as above but with the predictive provisioning method making a great mistake on the optimal number of instances. To do so, we fooled the `PredictiveProvisioner` into thinking that the expected workload pattern is that of hour 30 of the day-8 trace when it is that of hour 20. In Fig. 8(c), we depict the difference between the expected request arrival pattern and the observed one by the `ReactiveProvisioner` throughout this experiment.

As can be seen in Fig. 8(d), the `ReactiveProvisioner` is able to correct the erroneous prediction made by the predictive method. This is clearly observable in the first 5 minutes of the trace. For this time period, the `PredictiveProvisioner` allocated just one `SyncService` instance when more were needed, leading to high response times during this period as reported by Fig. 8(e). After this lapse of time, the `ReactiveProvisioner` reacted and added the right number of instances, resulting in a sharp reduction of the response time as shown in Fig. 8(e).

5.3.4 Fault Tolerance

Finally, we evaluated the fault tolerance of ObjectMQ auto-scaling. One central advantage of ObjectMQ is that the underlying queueing systems takes care that messages do not get "lost" in the event of a system failure, which tremendously simplifies the task of making auto-scaling fault-tolerant. In this sense, handling the failure of a `SyncService` instance consists of creating a new one whenever possible. Concretely, the `Supervisor` checks every second if all required instances are up and running. If an instance is missing, the `Supervisor` immediately starts a new one.

To test the responsiveness of the `Supervisor` in the presence of failures, we executed the first 10 minutes of the day-8 trace, which requires a single instance of the `SyncService`. The instance was programmed to crash every 30 secs. We recorded the response time of the `SyncService` when the instance was running and down, respectively. The results are shown in Fig. 8(f). The boxplots show that although the response time increases notably in the presence of failures, in practice it does not introduce delays greater than 1 sec, meaning that ObjectMQ provides enhanced reliability with a slight penalty on the system performance.

6. CONCLUSIONS

In this article we present StackSync, an elastic file synchronization architecture for open Personal Clouds. A core contribution of our architecture is to rely on a lightweight communication framework for providing programmatic elasticity to distributed objects using message queues as their underlying communication middleware. StackSync implements predictive and reactive provisioning policies on top of ObjectMQ that adapt to real traces from the Ubuntu One service. Also, the ObjectMQ unicast and multicast communication primitives have considerably simplified the code of the synchronization protocol. It also enables efficient change notification in a transparent way on top of the underlying messaging service.

StackSync provides a reference implementation and useful tools for rapid prototyping and evaluation. It has been extensively tested using real traces from the Ubuntu One system to validate its elasticity and efficient use of resources. Furthermore, extending an open benchmark [4] of Personal Clouds, we obtained good results comparing our service with Dropbox, Box, and OneDrive. StackSync is a stable open source project after two years of development that is being used in several public institutions and data centers.

Finally, an interesting open question is if our model and invocation abstractions can be generalized for offering programmatic elasticity to cloud applications. Since major cloud providers already offer scalable messaging services, it could be possible for them to offer equivalent programmable middleware and abstractions on top of their infrastructures. Messaging services could then become part of the existing load balancing fabric in the data center.

7. ACKNOWLEDGEMENTS

This work has been partially funded by the EU FP7 project *Cloud-Spaces* (FP7-317555) and Spanish research projects DELFIN (TIN-2010-20140-C03-03) and *Cloud Services and Community Clouds* (TIN2013-47245-C2-2-R) funded by the Ministry of Science and Innovation.

8. REFERENCES

[1] I. Drago, M. Mellia, M. Munafo, A. Sperotto, R. Sadre, and A. Pras, "Inside dropbox: Understanding personal cloud storage services," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2012, pp. 481–494.

[2] "How we have scaled dropbox," https://www.youtube.com/watch?v=PE4gwstWhmc.

[3] K. Jayaram, "Elastic remote methods," in *ACM/IFIP/USENIX Middleware 2013*, 2013, pp. 143–162.

[4] I. Drago, E. Bocchi, M. Mellia, H. Slatman, and A. Pras, "Benchmarking personal cloud storage," in *Proc. of ACM Internet Measurement Conference (IMC)*, 2013.

[5] R. Dievendorff, P. J. Helland, G. Chopra, and M. Al-Ghosein, "Queued method invocations on distributed component applications," Jul. 23 2002, US Patent 6,425,017.

[6] A. Lima, W. Cirne, F. Brasileiro, and D. Fireman, "A case for event-driven distributed objects," in *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, 2006, pp. 1705–1721.

[7] "Mongrel: message-queue-based-load-balancing," http://zef.me/4502/message-queue-based-load-balancingMongrel2.

[8] P. Eugster, "Type-based publish/subscribe: Concepts and experiences," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 29, no. 1, p. 6, 2007.

[9] P. T. Eugster, R. Guerraoui, and J. Sventek, "Distributed asynchronous collections: Abstractions for publish/subscribe interaction," in *European Conference on Object-Oriented Programming (ECOOP)*, 2000, pp. 252–276.

[10] C. Pairot, P. García, and A. F. G. Skarmeta, "Dermi: a decentralized peer-to-peer event-based object middleware," in *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS)*, 2004, pp. 236–243.

[11] D. A. Menasce, "Mom vs. rpc: Communication models for distributed applications," *IEEE Internet Computing*, vol. 9, no. 2, pp. 90–93, 2005.

[12] OASIS, "Amqp: Advanced message queueing protocol," http://www.amqp.org/.

[13] S. C. Kendall, J. Waldo, A. Wollrath, and G. Wyant, "A note on distributed computing," Mountain View, CA, USA, Tech. Rep., 1994.

[14] "Kryo: Fast, efficient java serialization and cloning," http://code.google.com/p/kryo/.

[15] R. Gracia-Tinedo, M. Sánchez-Artigas, A. Moreno-Martínez, C. Cotes-González, and P. García-López, "Actively Measuring Personal Cloud Storage," in *Proc. of IEEE CLOUD'13*, 2013, pp. 301–308.

[16] H. F. G. Y. Songbin Liu, Xiaomeng Huang, "Understanding data characteristics and access patterns in a cloud storage system," in *Proc. of IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGRID)*, 2013, pp. 327–334.

[17] P. Heckel, "Syncany open source file synchronization," http://www.syncany.org/.

[18] "Objectmq mom-rpc middleware," https://github.com/cloudspaces/objectmq.

[19] D. Harnik, B. Pinkas, and A. Shulman-Peleg, "Side channels in cloud services: Deduplication in cloud storage," *IEEE Security & Privacy*, vol. 8, no. 6, pp. 40–47, 2010.

[20] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *ACM SIGOPS Operating Systems Review (OSR)*, vol. 35, no. 5, pp. 174–187, 2001.

[21] 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.

[22] B. Urgaonkar, P. Shenoy, A. Chandra, P. Goyal, and T. Wood, "Agile dynamic provisioning of multi-tier internet applications," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 3, no. 1, pp. 1:1–1:39, 2008.

[23] V. Tarasov, A. Mudrankit, W. Buik, P. Shilane, G. Kuenning, and E. Zadok, "Generating realistic datasets for deduplication analysis," in *Proc. of USENIX ATC*, 2012, pp. 24–24.