

**Managed File Transfer**

functional design specifications

|  |  |
| --- | --- |
| Title: | Managed File Transfer Functional Design Specification |
| Abstract: | This document details the specification for the development of a Chunk file Uploader Downloader Service and its related UWP Client Proxy |
| Status\* | In Progress |
| Document Owner: | IBV |
| Document Author: | Gian Paolo Santopaolo |
| \*Draft, In progress, Reviewed, Authorized, Approved, Completed | |

|  |  |  |
| --- | --- | --- |
| **Document Control** | | |
| Document Distribution: | Internal: | All dev Teams |
|  | External: |  |
| Document Authorization: | Authorizer(s): | Gian Paolo Santopaolo |

Index

[Introduction 3](#_Toc72512909)

[Purpose of the document 3](#_Toc72512910)

[Overview 4](#_Toc72512911)

[Goal 4](#_Toc72512912)

[The architecture at a glance 5](#_Toc72512913)

[File Download: 5](#_Toc72512914)

[File Upload: 5](#_Toc72512915)

[File operations 6](#_Toc72512916)

[File Chunking 6](#_Toc72512917)

[File Checksum 7](#_Toc72512918)

[State machine with retries 7](#_Toc72512919)

[Prepare download (server) 7](#_Toc72512920)

[Downloader (client) 8](#_Toc72512921)

[Uploader (client) 9](#_Toc72512922)

[Chunks cleaner (server) 9](#_Toc72512923)

[Configuration 10](#_Toc72512924)

[Enterprise Cache 10](#_Toc72512925)

[Considerations 10](#_Toc72512926)

[Database 10](#_Toc72512927)

[Secure protocols and bandwidth 10](#_Toc72512928)

[Reliability 11](#_Toc72512929)

# Introduction

Sending and receiving large files over any network has always been too problematic. Based on our multi decade experience, the larger the file, the more problem it will cause when it comes to transferring it over any network.

In the end, the problem is always the same: time lost, productivity drained, and the assignment's success imperiled.

## Purpose of the document

This document details the specification for developing a service that allows a client to download and upload a file by chunking it.

# Overview

We wanted to create a file uploader/downloader SOA architecture that could easily be added to any existing .NET SOA architecture with a straightforward and easy-to-use client API to hide to the client developer all the complexity hidden in a process that involves chunking, state machines, queues, and so on.

Essentially, we needed a file uploader/downloader utility that could download and upload huge files to users around the world who were sometimes located in remote regions with slow and often faulty network links.

Even for users who are running high-speed and reliable connection, we want to allow them to have a system where files can be uploaded and downloaded in a safe, reliable, and secure way (for secure we mean that the file has to cross the network eventually by having a double encryption: at transport level but even more in-depth at the packet level)

We also needed to ensure that when huge files are being uploaded/downloaded, those files aren't buffered into server memory. This will lead to the SOA architecture to hang. Depending on his power, there will be a break point when memory limits are reached.

Counter wise the goal of this solution is to use minimal server memory to prevent server failures.

## Goal

Our goal is to create a SOA service architecture, including a UWP client, able to handle both upload and download the same way by:

* Splitting file in chunks (giving both server and client the ability to configure the chunk size)
* Download/upload the individual chunk on a separate thread (with a maximum configurable number of threads)
* Allow retries on errors when sending receiving chunks
* Easy and configurable parallel operation
* Work under load balancer if needed
* Secure booth transport and the singe chunk

# The architecture at a glance

## File Download:

The user will ask for the list of his file; then, he can choose which one to download.

A download request to the server is performed by the user (1). The request will also contain the chunk size client prefers.

The server starts looking for the file (2) and, if found, it will be split in the appropriate number of chunks and saved into a shared repository (4), so that solution is also loaded balancer compliant.

Once the chunking process is finished, the client is notified that he can start the download (5) with a message containing the following information:

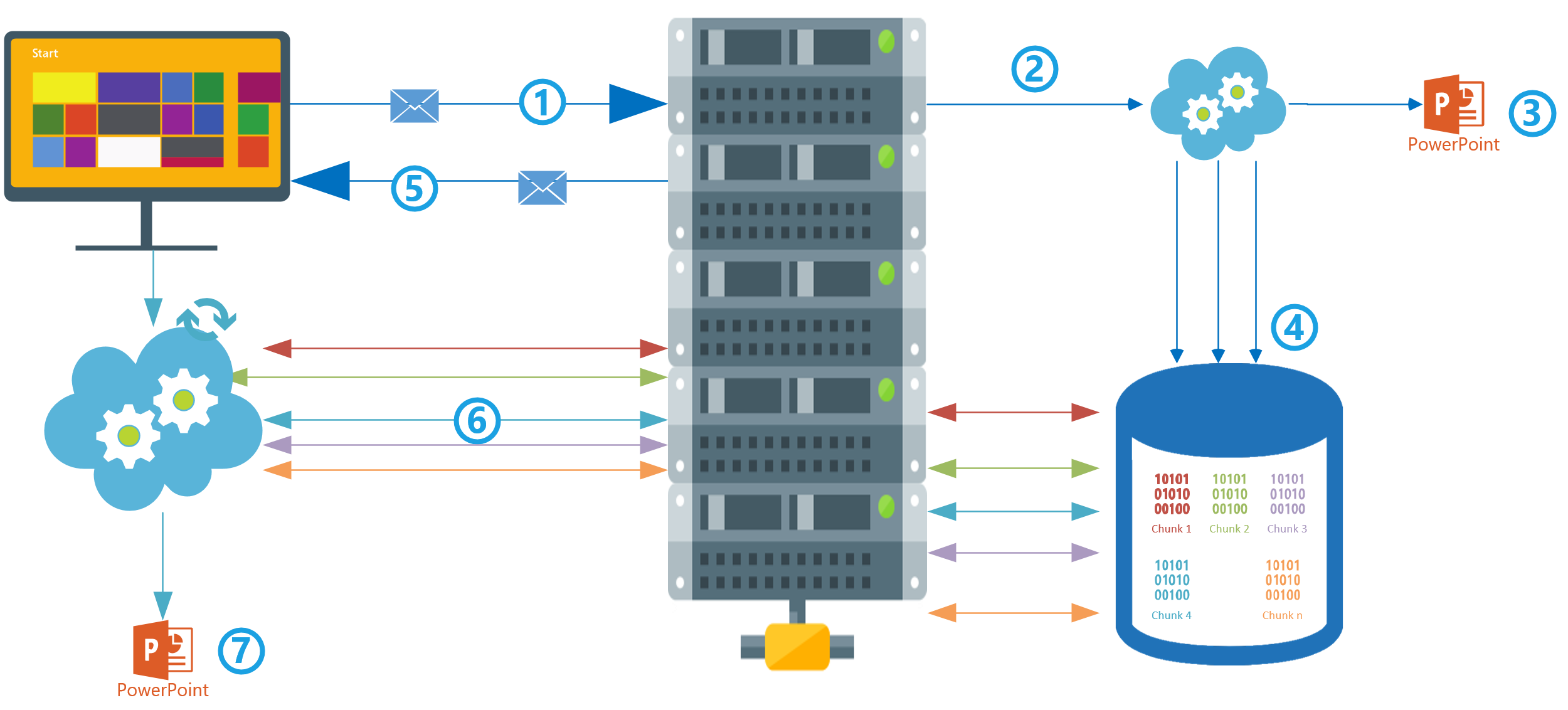
* Link for download
* Process id
* Chunk size
* Number of chunks
* Cancellation token
* File checksum

TBD if this message will be provided to the client via SignalR or client log polling.

Client SDK starts downloading each chunk thanks to a state machine (6) (described in the following chapter). The state machines internally will provide all the necessary architecture to download single chunks and retry to download if required.

User can pause or cancel the process anytime (thanks to the cancellation token)

Once the client SDK is done, it will calculate the file checksum again. If this is the same sent from the server, then the file is successfully downloaded (7), it will notify the client, thanks to an event, that file has been downloaded and available for being used.



*Architecture at a glance - File download diagram*

## File Upload:

The client asks the server for an upload (1) with a message containing:

* Filename
* Path
* Checksum
* Chunk size

The server will analyze the file if already present on a file share (2). If yes, the checksum is calculated

The server will reply (3) with a message containing

* If the file has to be uploaded or not (depending on checksum check)
* Process id
* Cancellation token

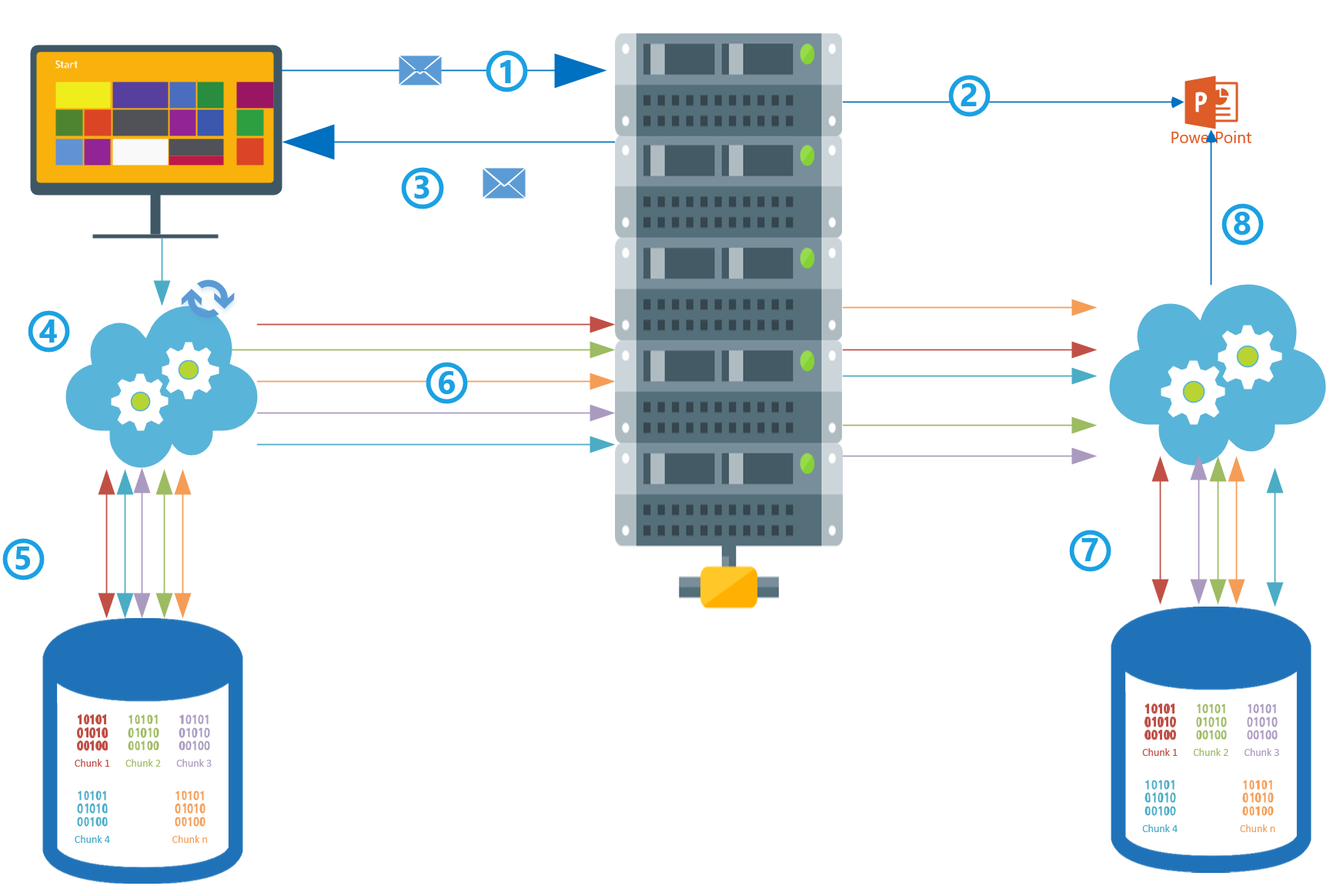
Depending on the file checksum client will start or not the chunk upload.

If the checksum is different, of file not present on the server, the client will start chunking file (4) on a local repository 5).

Once the chunk operation is finished, the client will start the state machine's upload operation (6).

The server will receive each chunk and will store it in its repository. (7)

When all chunks are uploaded server will create the file out of chunks (8) and save it in the appropriate location



*Architecture at a glance - File upload diagram*

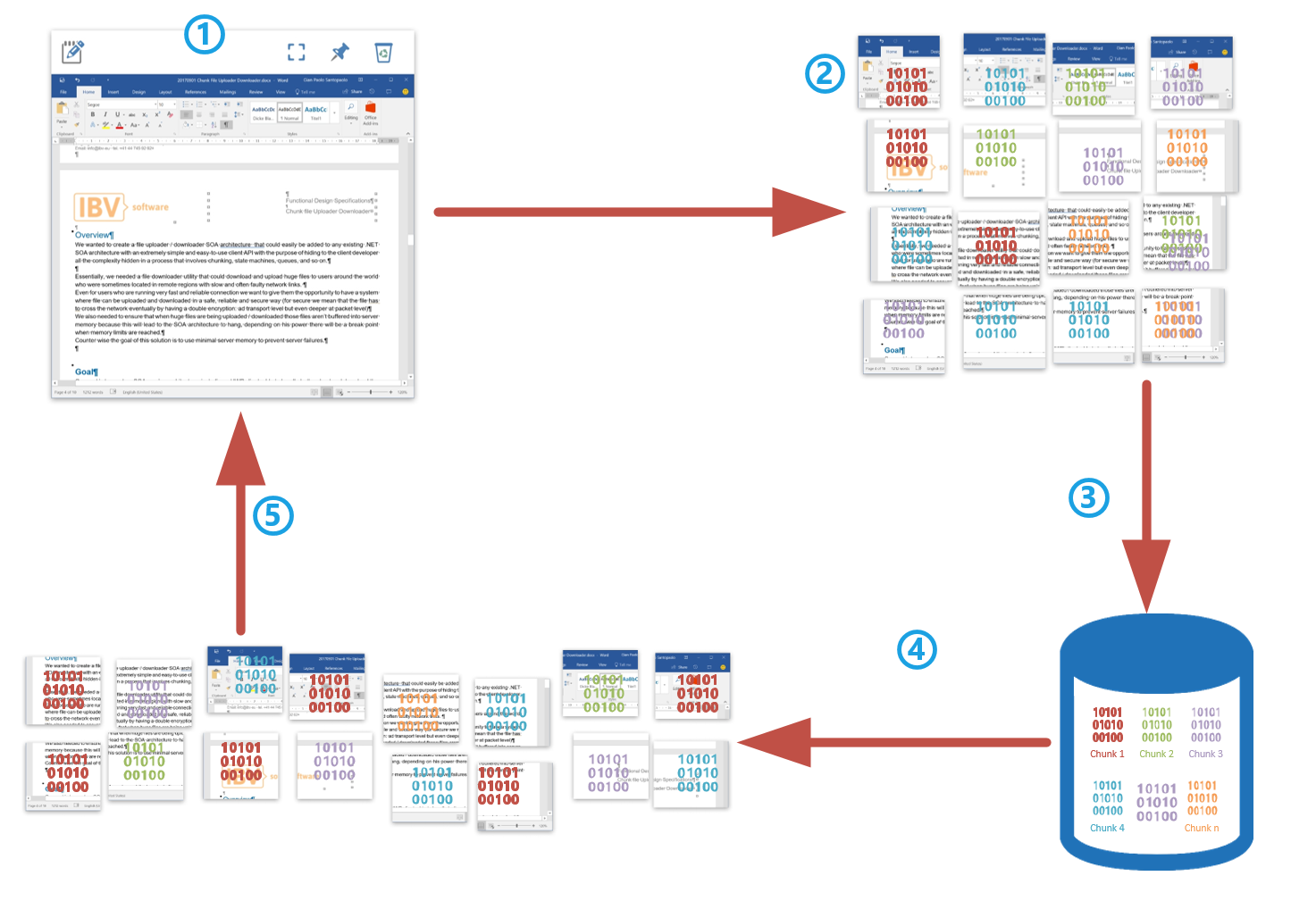
## File operations

### File Chunking

The concept of splitting a file is fundamental. To split the file (1) in a binary stream (2), from position zero, up to the last byte in the file, saving chunks of binary data to a shared repository (3) that all the server participating in the cluster can reach.

This approach works if the solution is deployed on a single server on an array of servers running under a balancer.

Then, when needed, every single chunk can be taken (4) from the shared repository and the initial file recreated (5)



*File chunking operation diagram*

### File Checksum

We should better investigate which kind of file checksum to use that best fits this project's requirements.

Our recommendation is to use a cryptographic checksum algorithm where the encryption key is stored on the global enterprise configuration.

### State machine with retries

All these state machines will be architecture and developed so that they can run on a windows service, with a configurable number of threads for each instance as well as on any private or public cloud ecosystem.

File checksum is the primary and most important phase of any process, both client and server-side.

SOA service and client sdk will share the same checksum algorithm with the same cryptographic key (configurable per customer). Once the document moves (uploads or downloads) from a side to the other, the application will verify that what was transferred is identical to the original copy.

### Prepare download (server)

Once a request has by collected by the client:

A new message is stored on the sate machine with status **ready to process**

The first available process/dequeuer will start processing it by adding a lockedby information and moving the status to **processing**

The first operation will check if the file exists and if the user has the right to pick it up. If there is a problem, the status will be changed to a faulting ending status.

If there will be an exception, the status will be changed to **ready** and retries augmented by one.

In case everything is fine file is spitted into chunks accordingly to the client chunk size request

Every chunk is stored on a database with an appropriate id and chunk number, as well as a checksum, to recreate lather the original document.

If, in the meanwhile, a cancel operation is received by the client (providing the appropriate cancellation token) process is stopped, and the object is set to an ending faulting state **operation canceled**

When the chunking operation has ended correctly object will be unbooked, and the status will be changed to **ready to download**.

Once the object is in ready to download state client will be notified that the download can start.

To be verified: if notification will be sent as a push (SignalR) or if the client will perform a long polling operation

When a new prepare for download operation is requested process will verify if the file on the file system (the same with the same chunk) is already present on the chunked files. If yes status will be updated to ready to download.

### Downloader (client)

When the client receives a message from the server that download can start a new state downloader object is created with a row for each chunk that has to be downloaded every row will have the **ready to download** starting status

Depending on the number of threads (configurable on the server that will be acquired by the client) some processes will start.

Each process will book one row of chunks to be downloaded, and the status is changed to **processing**.

If the download process goes well, the chunk is downloaded and saved on the local repository, the row is unbooked and set in, and the ending status **download complete**.

If the download process goes wrong, the state downloader object row is placed again in ready to download starting status, unbooked, and retry increased by one.

If only one ready to download starting status y one chunk exceeds the maximum number of retries the whole state downloader object is placed in a faulting ending status, **error** while downloading and Ui will be notified that there was a problem during download.Server will be notified as well that client was unable to finish the process.

Then the UI can request the file again.

When the process is correctly finished client will notify the server and the prepare download state machine object will be placed in an ending status **download completed.**

If the download can't be finished, it will also be notified to the server.

### Uploader (client)

When the user request for a file for being downloaded:

Client ask to the server for file properties (checksum, change date, if it already exists, and so on)

Client verifies if the file is already present locally and if is the same or not.

If it same, it is shown to the UI.

If it needs to be downloaded, a download request is sent to the server, the response will contain important information about the process.

The client waits for being communicated by the client that download can start (server is doing his work, Prepare download (server)).

Once the client is notified that download can start, a new state object is created on the uploader state machine.

For each chunk, there is a row in the state machine with status ready to upload.

The state machine, depending on configured threads, starts uploading each chunk.

If the chunk upload is ok then the row is marked as uploaded (not ending status)

If the chunk upload went wrong, state machine row is unbooked retry is increased by 1, and the row is set to status ready to upload.

If the retry of a single chunk exceeds the maximum configured retries process is stopped, every row is set to faulting ending status upload failed. The server is notified that the upload failed, so it can also act accordingly. An upload failed event is raised to the UI so it can act accordingly.

On the server, each chunk is saved on an upload state machine row

When the client notifies all chunks where uploaded, then the server start activity of:

* Reconstructing file
* Verify if chunk sent from the client
* Save it in the appropriate file system position
* Notify client activity is done

The client receives the message from the server. The file is correctly saved, and the client SDK pushes an event so that UI can handle it appropriately.

### Chunks cleaner (server)

There will be a process (service) running looking only at ending status chunks every state machine object on any ending status is picked up, booked, and if the time of the state machine object creation plus a configurable delta of time has passed then process will physically delete byte chunks from the repository.

The same process will also care about files that are in ready to download status for a delta of time defined on configuration if this is exceeded, then Prepare download state object will be placed in faulting ending status (download didn't start) ready for being deleted

When both the uploading or downloading process are completed, there will also be a cleaning process deleting all unneeded row from the shared repository.

## Configuration

There will be a global configuration for both servers and clients. Each parameter on the configuration will be provided to the server and clients will come from a global configuration, and any changes will be reflected hot, without the need to restart servers or clients. This behavior is described later in the chapter "Enterprise Cache."

### Enterprise Cache

The concept here is that server will have some cache ids. These ids are loaded the first time the server bootstraps.

Then, every each amount of configurable time a query is performed against the database by id (the fastest possible), asking only if that particular cache has changed. If yes, the server reloads its parameters by performing a more accurate query asking for all the needed parameters.

Cache id is updated so it is aligned with the database, and the cycle continues.

On the client, at the startup client asks via rest API the cache id, and then the enterprise cache concept is the same as for the server.

# Considerations

## Database

For an outstanding database, health is strongly recommended that the database running these state machines will be configured with a simple recovery model (like described in this official TechNet document)

That way log file won't store all the transactions after these are terminate that will lead to a healthy log file as well as a healthy database.

## Secure protocols and bandwidth

The formula below is usually used to measure performance on an SOA architecture when it comes to measuring the response time of a SOA service.

|  |  |
| --- | --- |
| Variable | Definition |
| RT | Response time. The total time from the user requesting a REST API (by clicking a button, and so on) to when the message come back and the UI shows desidered content with information coming from REST response. Typically measured in seconds. |
| Payload | Total bytes sent to the SOA service, RESTfull object + http(s) header and so on |
| Ecr | Encryption for the request, if absent 1 |
| Ect | Encryption for the transport channel eg. https, if absent 1 |
| Bandwidth | Rate of transfer to and from the server. This may be asymmetrical and might represent multiple speeds if a given page is generated from multiple sources. Usually, it is averaged together to create a single bandwidth expressed in bytes per second. |
| RTT | The time it takes to round-trip, regardless of bytes transferred. Every request pays a minimum of one RTT for the page itself. Typically measured in milliseconds |
| Concurrent Requests | Number of simultaneous requests a browser will make for resource files. By default, Internet Explorer performs two concurrent requests. This setting can be adjusted but rarely is |
| Cs | Compute time on the server. This is the time it takes for code to run, retrieve data from the database, and compose the response to be sent to the browser. Measured in milliseconds |
| Cc | Compute time on the client. This is the time it takes for a client for code to run, analyze the response and render the UI |

The main challenge in calculating this formula is by getting an exact measurement of each element.

This is challenging but not impossible, there are tools, and maybe some performance counters would help measure compute times.

This chapter aims not to dig into the formula itself but to look at two very important variables: Ecr and Ect so about security.

The higher these values will be, the more secure your service will be, but as you can clearly see from the formula, these two variables will exponentially impact the response time.

When deploying in production, our recommendation is to deeply impact the security level you choose to have the right benefit on both sides: Security and performance.

## Reliability

It is essential to specify that our solution and the way we work is by scaling the first instance on the code. Thus, our code is highly optimized, and we only use talented senior developers, most of the time, they are Microsoft MVP as well.

Because of the architecture that we develop, the solution can run on a single layer and different tiers.

Of course, there is something that cannot be reached only by developing the best code possible, and this is when it comes to very high volume, and the solution needs to scale up the hardware way or when it comes to reliability.

That's all said; we came to concepts such as load balancer or balancer of balancer that they can add the level of reliability that the customers want to reach.

The same is when it comes to SQL server; our application can run on a single instance database. It's on the customer to understand which level of reliability/security is needed for the database server.

Environments

For the deployment to be successful, our suggestion is to have at last, but then it depends on specific customer rules, two different environments:

* One for staging
* One for production