TJFS - Distributed File System

Tobiáš Potoček <tobiaspotocek@gmail.com>
Janak Dahal <jdahaldev@gmail.com>

TJFS

- Term project for CSCI 6450 at University of New Orleans
- Distributed file system
- Written in Java
- Synchronized using Zookeeper
- Vaguely based on Google File System
- Fault-tolerance guarantee: **Able to survive crash of any 1 server without data loss.**

Overview

Follows standard Google FS structure

- One **master server** that holds the metadata
- One **shadow server** that mirrors the master server
- Couple of **chunk servers** with the actual data stored in chunks
- **Zookeeper** that is responsible for the overall synchronization
- Separate **clients**

Overview

Follows Google FS structure

- Each file is divided into **chunks** of a fixed size (which is configurable; currently we're using **16 MB** per chunk)
- Each chunk is stored on at least **two chunk servers** (to maintain the fault tolerance guarantee)

General API

- **get**(String *path* [, int *byteOffset* [, int *numberOfBytes*]])
- **put**(String *path*, *data* [, int *byteOffset*])
- **delete**(String *path*)
- **size**(String *path*)
- **time**(String *path*)
- **list**(String *path*)
- **move**(String *sourcePath*, String *destinationPath*)

The whole API is **stream-based** (and yet memory efficient).

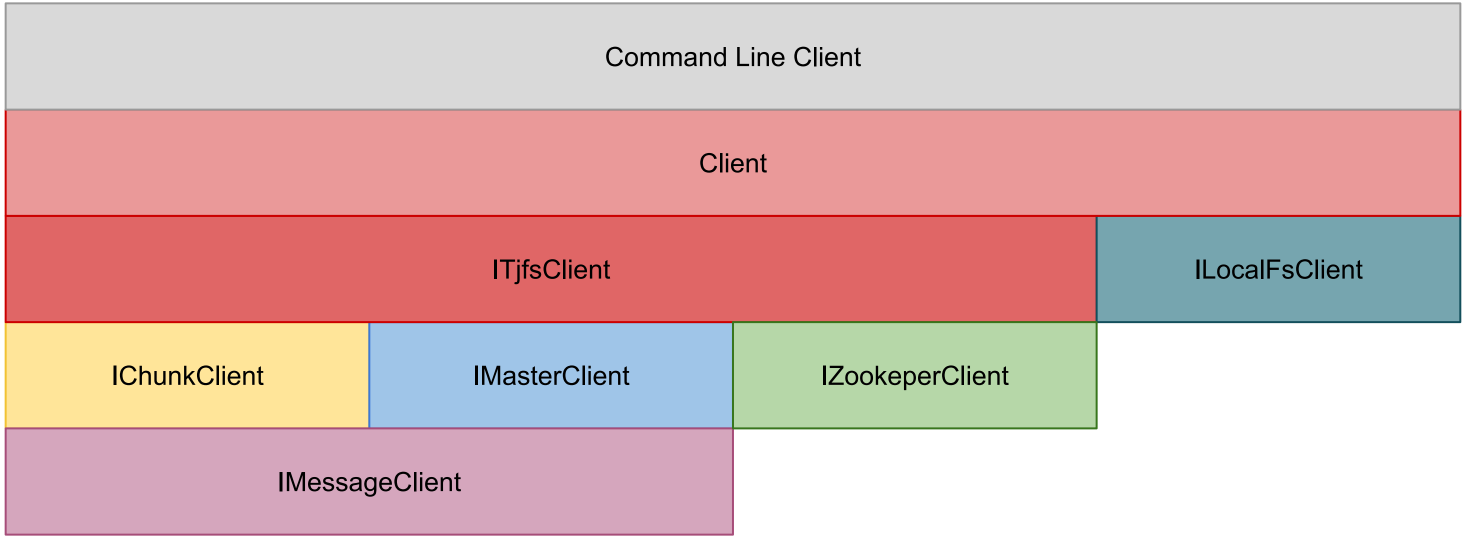
(Still) no folders!

- Internally, there aren’t any folders because we don't need them
- Files are stored in a **key value** storage
- **Key** is the path and **value** is actual data
- Folders are supported through the format of the path
    - Each path starts with slash, no slash at the end of the path
    - Folders are separated by slashes
    - Last segment is the name of the file
    - Example: */foo/bar/johndoe*
    - To mimic a behavior of standard file systems, we don’t allow to create files with names that already “exists” as folders.

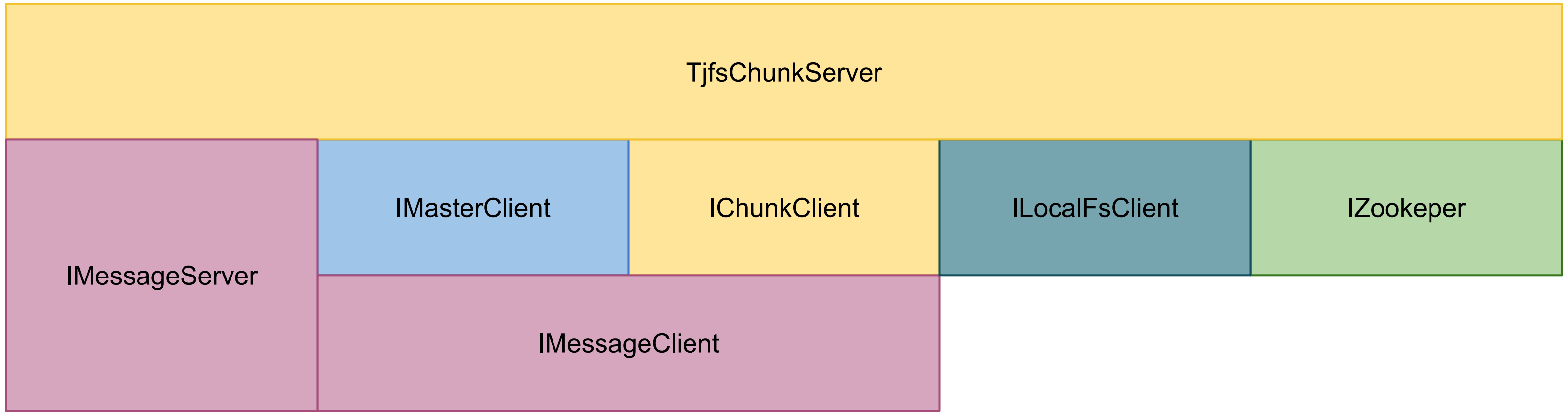
Base application architecture

- The whole application consists of separated, reusable, **JUnit-tested** components.

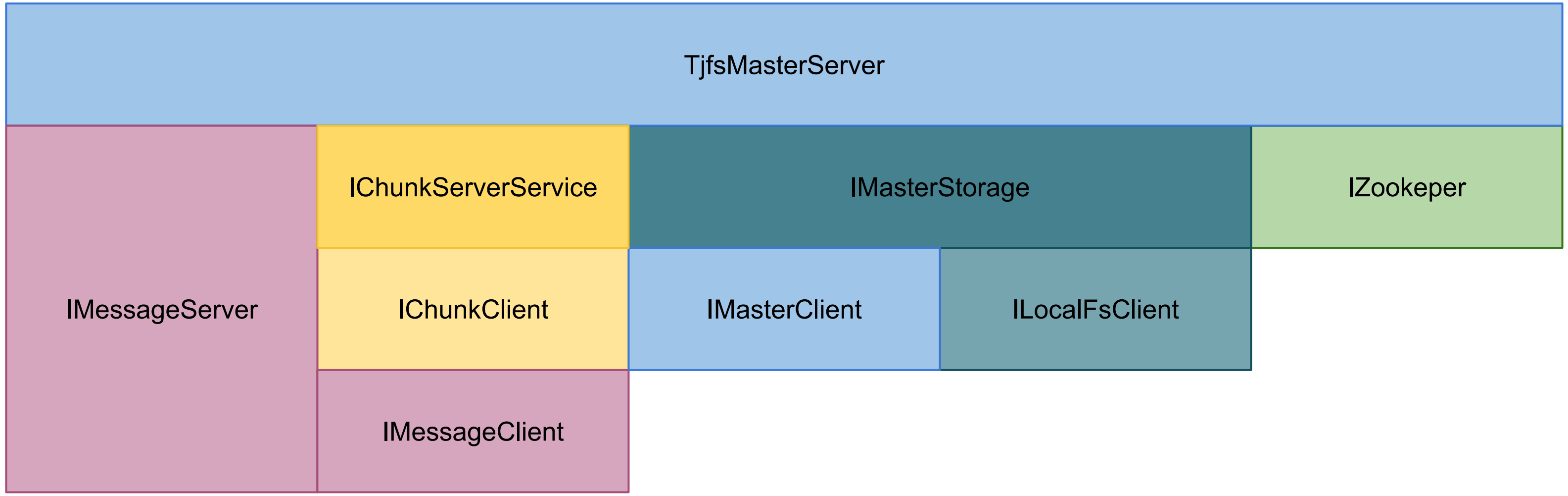
Base architecture: client



Base architecture: chunk server



Base architecture: master server



Messaging Layer

Messaging Layer overview

- low-level layer responsible for all network communication
- build over TCP sockets
- request/response (or client/server) principle
- in this manner similar to HTTP but heavily simplified
- messages optimized to efficiently carry both small amount of structured data and large data chunks

Messaging Layer - message structure

2byteHeader + 10byteJsonLength + 10byteDataLength + JsonMessage + actualDataBytes

- Header defines how to parse the json message
- Json message (request/response) arguments for all scenarios have been defined
- Headers for both requests and responses are defined
- Messages exchanged using TCP sockets
- Simple multithreaded socket server is implemented to handle concurrent requests

Message Layer (cont..)

Message Client

- Parses the **Request into the message** and writes to the socket
- Reads from the socket and parses the **message into a Response**

Message Server

- Listens on the socket and parses the **message into Request**
- Calls the server (Chunk, Master) to process the request
- Parses the **Response into message** and writes to the socket

Message Layer - error

- Error message (response argument) is defined
- Message server parses any errors from server into a error message
- Message client would consume the error message and report to the upper layer

Chunk Server

Chunk server

- Simple chunk storage
- Each chunk is stored as a normal file with its name. No metadata is maintained on the chunk server.
- Provides API for putting and getting chunks.
- Handles the chunk replication between servers when requested
- **ChunkClient** exposes the functionality of the chunk server

ChunkDescriptor

Describes a chunk. Contains the following information about the chunk

- String ***name***
  - Chunk name is a uniquely generated hash (combination of threadID, current time and random UUID)
- Machine[] ***chunkservers***
  - Machines where the chunks are placed
- int ***size***
  - Size of the chunk
- int ***positionInFile***
  - The ordering of the chunks of a file

Chunk Client API

- **get**(ChunkDescriptor *chunkDescriptor*)
- **put**(ChunkDescriptor *chunkDescriptor,* byte[] *data*)
- **delete**(Machine *machine*, String *chunkName*)
- **list**(Machine *machine*)
- **replicate**(Machine *machineTo*, String *chunkName*)

Master Server

Master Server overview

- central file system component
- contains the **actual file system content** (fs metadata: hierarchy of files stored in the fs, names, dates, sizes, file-to-chunk mappings)
- keeps track of existing chunks in the system (where they can be found)
- responsible for **chunk replication** when a chunk server goes down
- can run in a **master** or backup **shadow mode** (synchronizes with the actual master)
- **does not handle client synchronization** (locks). Zookeeper does that.

Master Server

- Contains the metadata of the **files** in a **persistent storage**
- Contains the **chunk** to chunkserver mappings in a **transient storage**
- **MasterClient** exposes the functionality of the master server
- Master server contains an instance of **MasterStorage** which stores the file metadata in a local file system (we decided not to use the database)
- Contains an instance of **ChunkServerService** which contains the latest chunk to chunk server mappings
- Multiple instances of master can run
    - One which acquires the lock on zookeeper becomes the **real master**
    - And others become **shadow**

Master Server

Real master

- Processes incoming requests from **clients and shadow master**

Shadow master

- **Rejects** incoming requests
- Listens to the **MasterServerDown** event
- **Polls master** for the latest updates (**Replication**)

File Descriptor

Contains information about the metadata of the file including all the chunk information

- Path ***path***
  - The path of the file in the filesystem
- Date ***time***
  - The last modified time of the file
- ChunkDescriptor[] ***chunks***
  - The chunk information about the file

Master Client API

- **allocateChunks**(int *number*)
  - client requests certain number of chunks in advance when reading a file
- **getFile**(Path *path*)
  - returns the latest **FileDescriptor** for the given path
- **putFile**(FileDescriptor *file*)
  - updates the **metadata** of the given file
- **getLog**(int *version*)
  - returns the **list of FileDescriptors** by parsing log files in master numbered > version
- **list**(Path *path*)
  - returns list of files in given path (mimics folder listing)
- **getLatestSnapshot**()
  - returns the latest snapshot from the master

MasterStorage

- the master works as a simple **FileDescriptor** storage
- **key-value** basis; a path is the **key** and the **FileDescriptor** is the **value**
- most of the interaction with the filesystem is done through **getFile** and **putFile** methods
- **getting a file** means getting a FileDescriptor with updated chunk locations
- **putting/updating a file** means putting an updated FileDescriptor to a given path.

MasterStorage

- Stores the json of the **FileDescriptor** into a log file. A file descriptor represents a **change** in the file system.
- The log files are numbered **incrementally**
- **Higher numbered** log file represents the **latest** information
- The log is simply a **folder** containing incrementally named file descriptors
- The **highest number** is the current **version** of the file system
- **Empty file descriptor** (without any chunks) represents a **deleted file**
- Moving a file would require **putting** an **empty file** descriptor and **putting** a **new file** descriptor (it’s done by the client)

ChunkServerService

- Provides a **layer of abstraction** for master to keep track of all the chunk servers
- Listens to the **zookeeper events** (ChunkServerUp and ChunkServerDown)
- Starts **chunks replication** when a chunk server goes down **(when chunk replica count < 2)**
- Updates the **chunk mappings** when a chunk server **comes up**

Master server snapshotting

- Master server periodically creates **snapshots** of the current state:
    1. Current version is fixed
    2. If there is an existing older snapshot, it's loaded into memory
    3. Current log is loaded (entries starting at the last snapshot version, ending at the current fixed version).
    4. The information from the old snapshot and the log is combined into a new state
    5. The state (list of FileDescriptors) is serialized (JSON) and stored on the disk under the fixed version.

Master Server startup sequence (real master)

1. Master goes up.
2. Initializes the master storage service; loads file system into memory
3. Initializes the chunk server replication service
4. Attempts to register with Zookeeper as new master; **succeeds**
5. Starts snapshotting
6. Starts chunk server synchronization
7. Starts processing incoming requests.

Master Server startup sequence (shadow)

1. Master goes up.
2. Initializes the master storage service; loads file system into memory
3. Initializes the chunk server replication service
4. Attempts to register with Zookeeper as new master; **fails**
5. Disables chunk server synchronization
6. **Clears current file system state**
7. Fetches and loads last snapshot and log from the real master
8. Starts periodically fetching log updates from the real master
9. **Refuses any incoming requests**

When master goes down...

1. The shadow master is notified through the event system
2. Attempts to register with Zookeeper as new master (**succeeds**)
3. **Stops replication** (nobody to replicate from)
4. Starts chunk server synchronization
5. **Starts processing incoming requests**

Client

Client

- Responsible for most of the work (**50% more** lines of code than master)
- **Trusted** from all other components (can cheat any time it decides)
- **Stream-based** (i. e. as you are downloading a file from the Internet you can directly store it in tjfs)
- **Never stores the whole file in memory** (can theoretically handle arbitrarily large files; at this point we use **integer** to handle file sizes which limits it to approximately 2 GB)

Command line client

- The actual stream-based **TjfsClient** is wrapped by a **Client** that adds support for local file system (transferring files between local file system and the distributed file system).
- The **Client** is wrapped by a **Command line client** which allows the user to operate the distributed file system from a simple command line

Writing a file

1. The client uses the Zookeeper to **get a lock** for the file (might fail).
2. The client sends a request to master to get the **current file descriptor** (if the file does not exist yet, an empty file descriptor is returned)
3. The client, according to the file size and the configured chunk size, asks the master to **allocate new chunk names**.

Writing a file

4. The master checks the connected chunk servers and **evenly distributes the new chunks between them** (random generator is a friend), making sure each chunk is assigned to two chunk servers (the duplication).
5. The master sends the distribution back to the client (sends a list of **chunk descriptors** containing a **name** and and **two target chunk servers**)
6. The client splits the data into **chunks**, assigns them allocated **names** and **pushes** them **to target servers**. If we're overwriting an existing file, the client first fetches the old chunk, **replaces its data** and then pushes a new chunk.

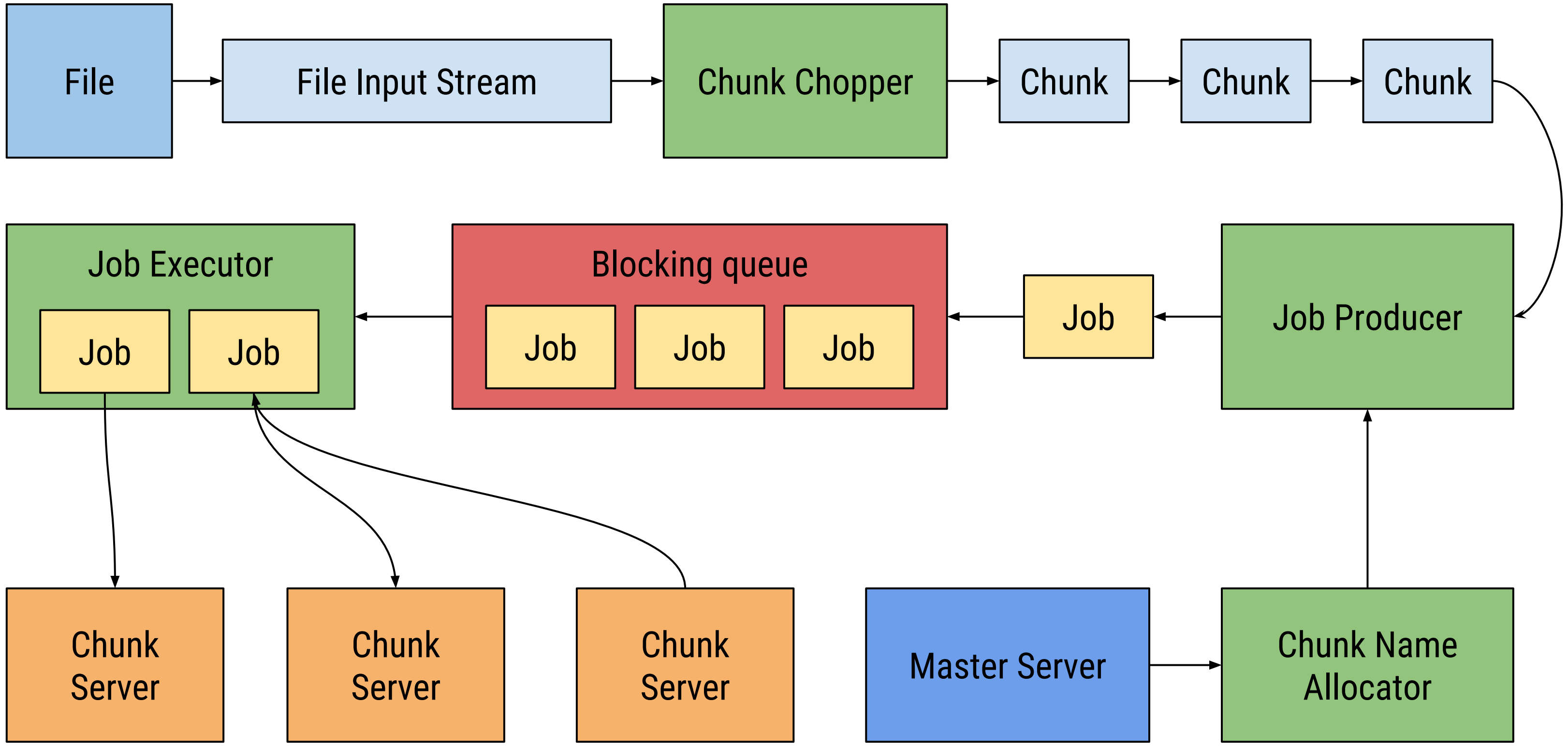
Writing a file

7. The client **updates the file descriptor** (adds the new updated chunk descriptors)
8. The client **pushes the file descriptor to the master**
9. The master **replaces the old file descriptor** with the new one and **remembers** all new available **chunks** (on which chunk servers they are available)
10. The client **releases the file lock** in Zookeeper

Writing a file

- To achieve good performance and promised memory efficiency, it’s all done in parallel.

Writing a file



Writing a file in parallel

- **Put Chunk Job**: "take *this data* and try to push it to one of these *two chunk servers* under this *chunk name*".
- Jobs produced by **Job Producer** that uses **Chunk Chopper** to split incoming data into chunks of the right size and **Chunk Name Allocator** to obtain chunk names and target servers from the master.
- **Job Executor** maintains a pool of **worker threads** that concurrently take out jobs out of the **Blocking Queue** and try to get them done.

Writing a file in parallel

- The size of the blocking queue and the number of worker threads are configurable.
- Together with the chunk size they determine **how much memory is required by the client**
- The higher the number of concurrent jobs is, the better utilization of network is achieved (even if there is a chunk server that responds slowly, the general slowdown will be diminished)

Writing a file: handling chunk server failures

- Each allocated chunk has assigned **two target servers**.
    - If the client **succeeds** to push the chunk to the first one, it initiates **asynchronous replication** from the first chunk server to the other and starts doing something else (doesn’t care about the result).
    - If the client **fails** to push the chunk to the first one, it tries to **synchronously push it to the second one**. If it **fails as well**, the whole procedure is killed and **writing the file fails**
    - Writing a file is considered successful if **each chunk is successfully transferred to at least one chunk server** (it can be replicated later)

Fault-tolerance?

- Master goes down
  - If it goes down after when the new file descriptor is written, it depends on **whether the change has been replicated to the shadow**. Regardless of that, the system remains in a **consistent** state but there is a really short time window during which **the change might get actually lost** (it's written to the log but the log might be lost as well). Currently the replication goes in **10 seconds intervals** (which is the length of the window) but it can be significantly reduced without performance impacts.
  - If it goes down before, we might end up with some **unused/abandoned chunks** on chunk servers. That is not a problem and we might use garbage collector to get rid of them later. And the client is notified that the writing failed.

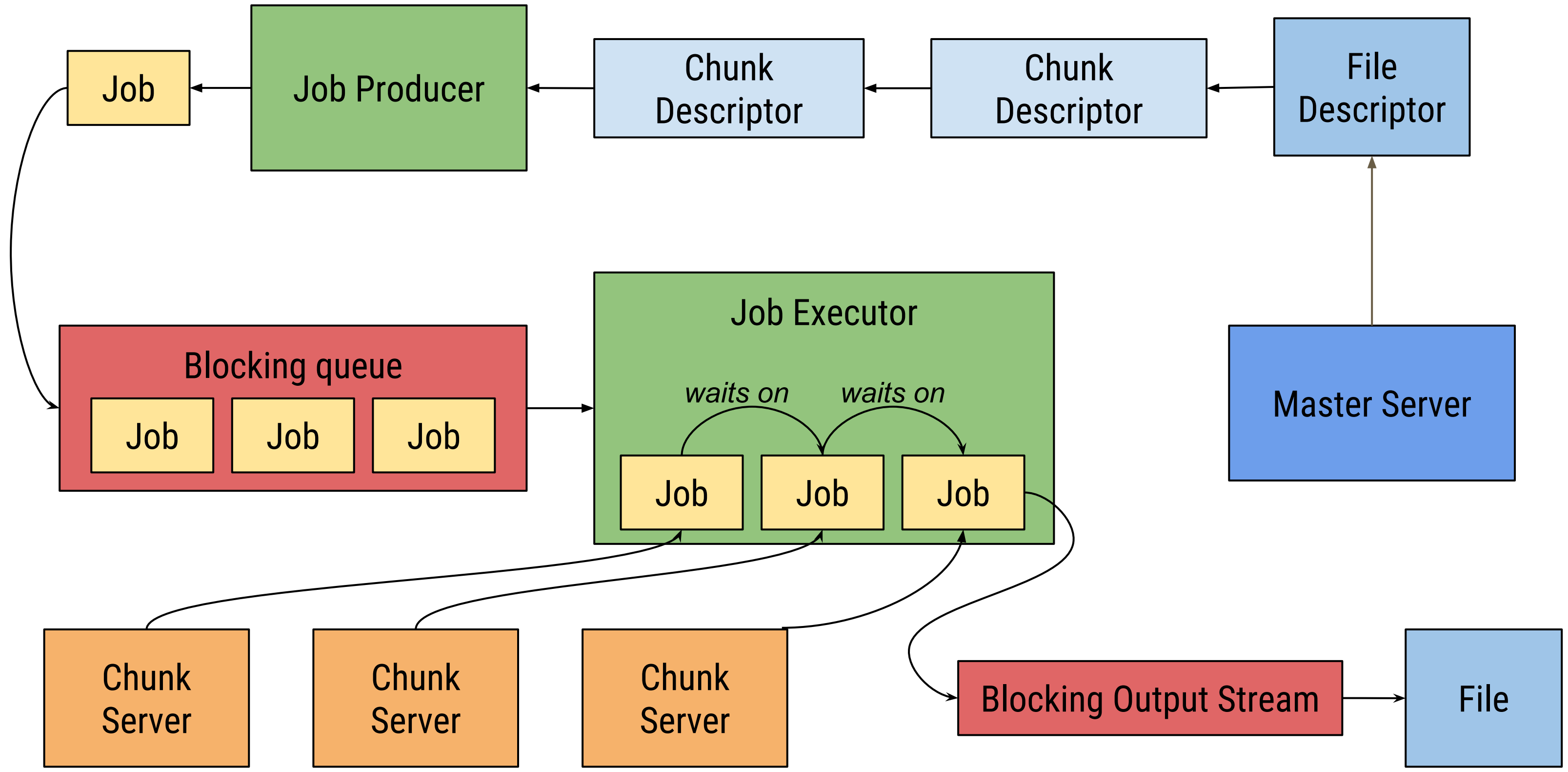
Fault-tolerance?

- Chunk server goes down
    - We still have **one extra copy** of each chunk on other chunk servers and master knows what chunks were on that chunkserver (even those new ones) and can initiate a replication routine.

Getting a file

1. The client **acquires the read lock** from Zookeeper for the file
2. The client **fetches the file descriptor** from the master (file descriptor contains list of all chunks together with up-to-date chunk locations, i. e. on which chunk servers they are available).
3. The client **fetches all chunks** and combines them into the final file
4. The client **releases the lock** from Zookeeper

Getting a file



Getting a file in parallel

- **Get Chunk Job**: “Here is a *chunk name* and *list of servers* where it should be available. Try to download the chunk and write it to the output stream”
- **The jobs have to finish in an exact order.**
- Therefore each job downloads the chunk and then **it waits until the previous job is finished**.
- Only then it writes the data to the output stream.
- The output stream has a **limited buffer** and is **blocking**. Therefore if the user is not reading from the stream, no chunks are being downloaded.

Zookeeper Client

Zookeeper Client

- Extra layer over Zookeeper API
- **Synchronizes the state** with Zookeeper (the current master and list of active chunk servers) using Zookeeper *watches*
- **Emits events** to the rest of the application (master server up/down, chunk server up/down)
- Provides API for **registering as new master/chunk server**
- Provides API for **getting current master and chunk servers** (really fast)
- Provides API for **acquiring and releasing locks** on files

Locking

Simple locking routine based on **sequential** and **ephemeral** nodes:

1. Client creates a **new sequential** (and ephemeral) **node** and gets back the **actual node name** with the **sequential suffix** assigned by Zookeeper
2. Client fetches the list of sibling nodes and checks whether **his sequential suffix is the lowest**
3. If it is, the lock **has been acquired**. **The end**. If it's not, the lock has been acquired by **someone else**.
4. Client **removes the node** and depending on situation either completely **quits** (writing a file) or **waits for an event** (attempt to become a master).

Locking

- **Deleting the node is not necessary** but it makes more sense from our API perspective (if I fail to obtain the lock, I clean after myself and try again later).
- Multiple types of locks for file: **READ** and **WRITE**
- Concurrent **READ** locks are allowed but it is not allowed to obtain a **WRITE** lock if somebody has the **READ** lock

Zookeeper node structure

- **/master/**_machine0000000002_ – _sequential_ node containing _current master_ (its IP address and port)
- **/chunkservers/*** – list of _active chunk servers_, each node will represent one chunk server, the name of the node is the IP address and port, i. e. **/chunkservers/**_192.168.1.101:6002_ (the node is empty)
- **/fs/*** – list of nodes corresponding to files (node name is a hashed path), each node contains current locks (empty _sequential_ nodes) held for the file, i. e. **/fs/-435643121244/**_read0000000005_

Conclusion

Benchmarking

- No real benchmarking done, we used Windows Task Manager to monitor the bandwidth going through the network card.
- The write/read speed reaches **800-900 Mbits** per second but occasional drops can be experienced (especially in the beginning and at the end).
- As everything is random, it might happen that one chunk server is under heavier load than others. The jobs running in parallel might studder a bit.
- It’s probable that the fs will perform better with larger number of servers.

What's left

The filesystem is 99% finished, seems to be stable and performs well but there is couple of things that could be done:

- No garbage collection yet (*easy fix*)
- File sizes represented in integers limiting the file size to 2 GB (*easy fix*)
- The chunk replication routine is at the moment invoked only by the chunk server down event. Therefore under certain circumstances some chunks might remain not replicated (*easy fix*)
- Not really friendly error messages (*lots of work*)
- The short time window during which we might lose some changes (*needs analysis*; enforcing log replication on every update might be enough)