Secure Distributed File System (SDFS)

Kelsey Francis   Christopher Martin
`francis@gatech.edu`   `chris.martin@gatech.edu`

1 Protocol

1.1 Certificates

Every node has a `pki` directory containing Java key stores:

- `ca-certs.jks` - Contains the certificates needed to trust the CA.
- *[your-keystore]*`.jdk` - Contains your cert (signed by the CA) and your private key.

1.2 Messages

Similarly to HTTP, each message consists of a header string followed by two newline characters and an optional message body. Headers are realitively short (no more than 6 lines), and we impose a maximum header size of 8KB as a sanity check.

1.2.1 Get

Get messages are sent by clients to request a file.

Lines of the message header

1. `get`
2. A new correlation ID
3. The name of the file

Example header

```
get
c36c6d81-2051-49e5-8e17-1c746aa9533b
sas.txt
```

1.2.2 Put

Put messages are sent by both clients and servers. When the server sends a *put* (in response to a client *get*), the file body immediately follows. When the client sends a *put*, it waits to receive an *ok* response before sending the file content.

Lines of the message header

1. `put`
2. A correlation ID. For a client-to-server message, this is a new ID. For a server-to-client message, this is the ID sent in the *get* message that was used to request this file.
3. The name of the file
4. The SHA-512 checksum of the file contents, base 64
5. The file size in bytes, base 10

Example header

```
put
c36c6d81-2051-49e5-8e17-1c746aa9533b
sas.txt
3DkQJagLJhzzo8JhjKV77CFP [...]
706120
```

1.2.3 Delegate

Delegate messages are sent by clients.

Lines of the message header

1. `delegate`
2. A new correlation ID
3. The name of the file
4. The CN of the principal receiving delegated rights
5. Some non-empty space-delimited combination of `get`, `get*`, `put`, `put*`
6. The expiration time of the delegation, as a UNIX timestamp (number of milliseconds since 1970 UTC)

Example header

```
delegate
c36c6d81-2051-49e5-8e17-1c746aa9533b
sas.txt
bob
get* put
1367021993012
```

1.2.4 Ok

After the server receives a *put*, it replies with an *ok* message if the client has permission to write the file.

Lines of the message header

1. `ok`
2. The correlation ID of the message to which the server is responding
3. The name of the file for which write access was requested

Example header

```
ok
c36c6d81-2051-49e5-8e17-1c746aa9533b
sas.txt
```

1.2.5 Unavailable

The server responds to a *get* or *put* with an *unavailable* message if the requested resource is currently locked. This indicates a temporary failure condition, and the client should wait briefly and retry.

Lines of the message header

1. `unavailable`
2. The correlation ID of the message to which the server is responding
3. The name of the file on which some operation was requested

Example header

```
unavailable
c36c6d81-2051-49e5-8e17-1c746aa9533b
sas.txt
```

1.2.6 Prohibited

The server responds to a *get* or *put* with an *prohibited* message if the client does not have permission to perform the requested operation.

Lines of the message header

1. `prohibited`
2. The correlation ID of the message to which the server is responding
3. The name of the file on which some operation was requested

Example header

```
prohibited
c36c6d81-2051-49e5-8e17-1c746aa9533b
sas.txt
```

1.2.7 Nonexistent

The server responds to a *get* with a *nonexistent* message if the client requested a resource that does not exist.

Lines of the message header

1. `nonexistent`
2. The correlation ID of the message to which the server is responding
3. The name of the file on which some operation was requested

Example header

```
nonexistent
c36c6d81-2051-49e5-8e17-1c746aa9533b
sas.txt
```

1.2.8 Bye

The client or server can send a *bye* message as a friendly indicator of intent to close the connection.

Lines of the message header

1. `bye`

Example header

```
bye
```

2 Performance

To test performance, we ran the server and client on two physically separate machines connected to the same gigabit switch. Our results were as indicated in the following table:

	Get time	Get speed	Put time	Put speed
100 kB	72.14 ms	1.386 MB/s	38.68 ms	2.585 MB/s
1 MB	279.7 ms	3.575 MB/s	102.3 ms	9.778 MB/s
10 MB	849.3 ms	11.774 MB/s	526.7 ms	18.986 MB/s
100 MB	4.929 s	20.289 MB/s	4.488 s	22.283 MB/s
1 GB	50.21 s	19.918 MB/s	48.67 s	20.546 MB/s
10 GB	883.2 s	11.322 MB/s	435.5 s	22.961 MB/s

Note that our implementation is capable of handling very large files. It accomplishes this without the need for much memory at all – we run with the default JVM max memory settings – by performing all file operations, including hashing and encryption/decryption, on streams rather than first loading entire files into memory.

The average speed appears to level off above 10 MB. Excluding smaller files, we conclude that our put throughput is approximately 20 MB/s and our get throughput roughly 10-20 MB/s.

Attaching a profiler to the VM during several test transfers revealed the bottleneck to be primarily encryption/decryption and to a lesser degree hashing of the file contents, rather than the network itself or disk I/O.