Distributed Systems

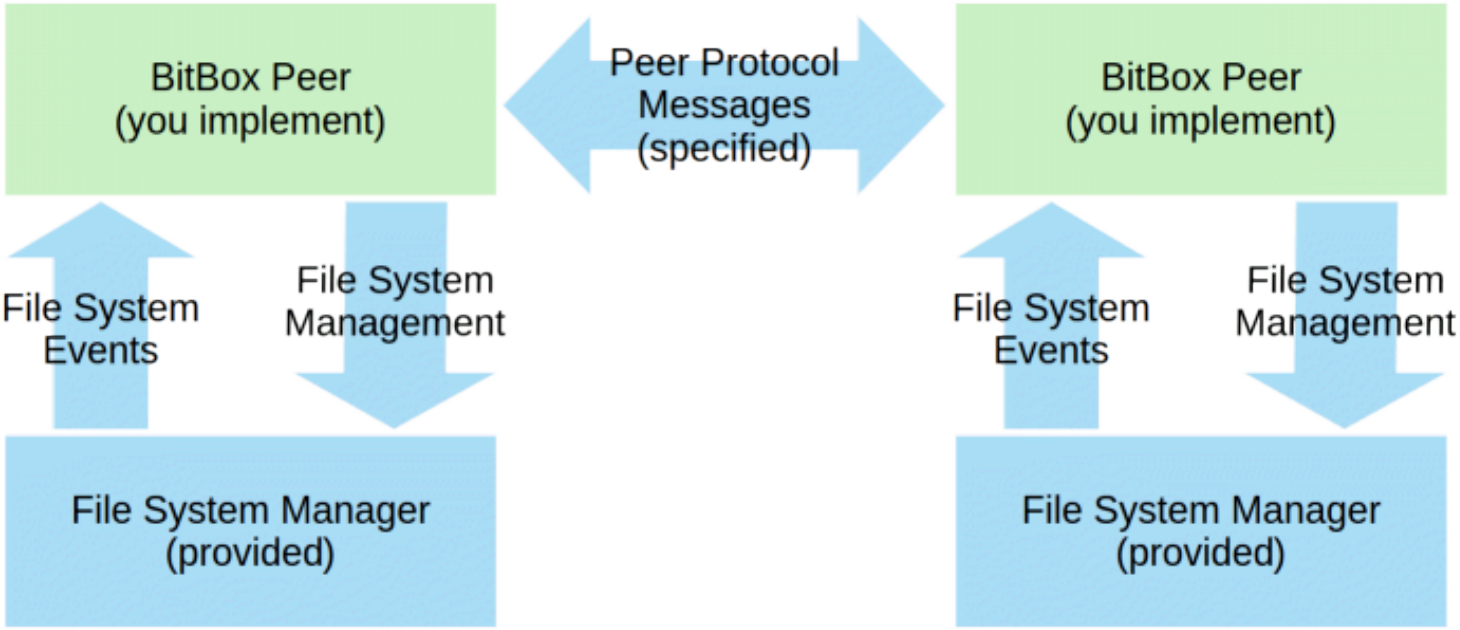
COMP90015 2019 SM1

Project 1 - BitBox

Something like BitTorrent and DropBox

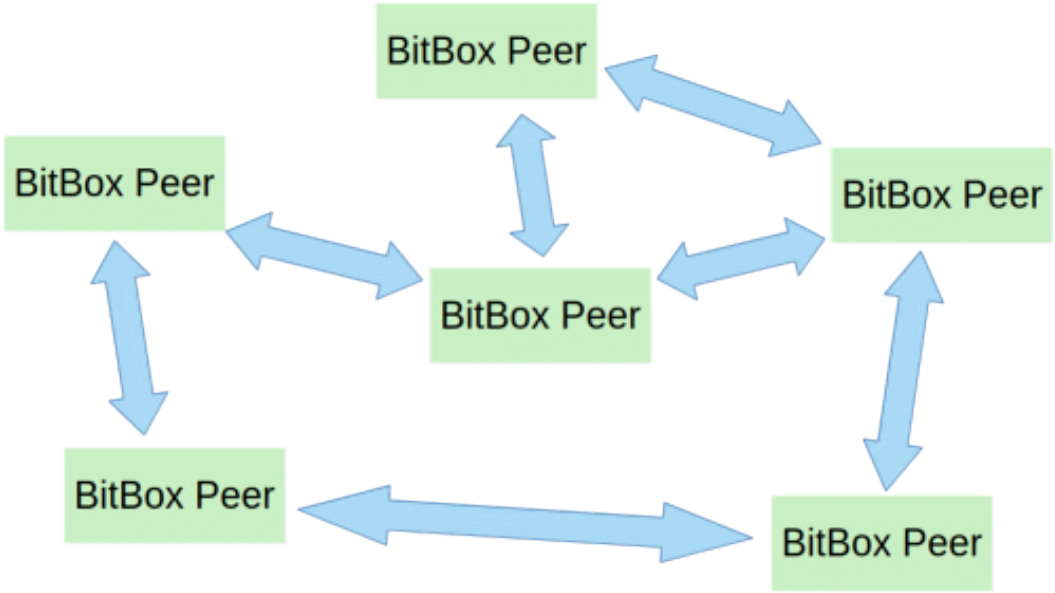
Architecture

The basic components are the File System Manager that monitors a given directory in the file system on a local machine, for changes to files, etc., and a BitBox Peer that can "relay" these changes to another BitBox Peer on a remote machine.


BitBox Peer
(you implement)
Peer Protocol
Messages
(specified)
BitBox Peer
(you implement)
File System
Events
File System
Management
File System
Events
File System
Management
File System Manager
(provided)
File System Manager
(provided)

Unstructured P2P Network

We would like to allow the BitBox Peers to form an unstructured P2P network. By unstructured we mean that the connection pattern has no relevance to the functionality, and is somewhat arbitrary depending on how and when peers come and go.


BitBox Peer
BitBox Peer
BitBox Peer
BitBox Peer
BitBox Peer
BitBox Peer

File System Events

The File System Manager monitors a given directory, called the share directory, recursively, for all changes to directory structure and files. It generates File System Events as it detects changes:

- **FILE_CREATE**
- **FILE_DELETE**
- **FILE_MODIFY**
- **DIRECTORY_CREATE**
- **DIRECTORY_DELETE**

The BitBox Peer is an observer of these events. More documentation about the events can be found in the provided `FileSystemManager.java` file.

File System API

The File System Manager provides an API for the BitBox Peer to safely interact with files and directories inside the share directory:

- cancelFileLoader(String), createFileLoader(String, String)
- checkShortcut(String), checkWriteComplete(String)
- deleteDirectory(String), deleteFile(String, long)
- dirNameExists(String), fileNameExists(String)
- fileNameExists(String, String), generateSyncEvents()
- isSafePathName(String), readFile(String, long, long)
- makeDirectory(String), modifyFileLoader(String, String, long)
- writeFile(String, ByteBuffer, long)

The API is fully documented in the `FileSystemManager.java` file.

Communication

All communication will be via persistent TCP connections between the peers.

All messages will be in JSON format, one JSON message per line, i.e. the JSON object is followed by a new line character.

The text encoding for messages will be *UTF-8*: e.g. `new BufferedWriter(new OutputStreamWriter(socket.getOutputStream(), "UTF8"));`

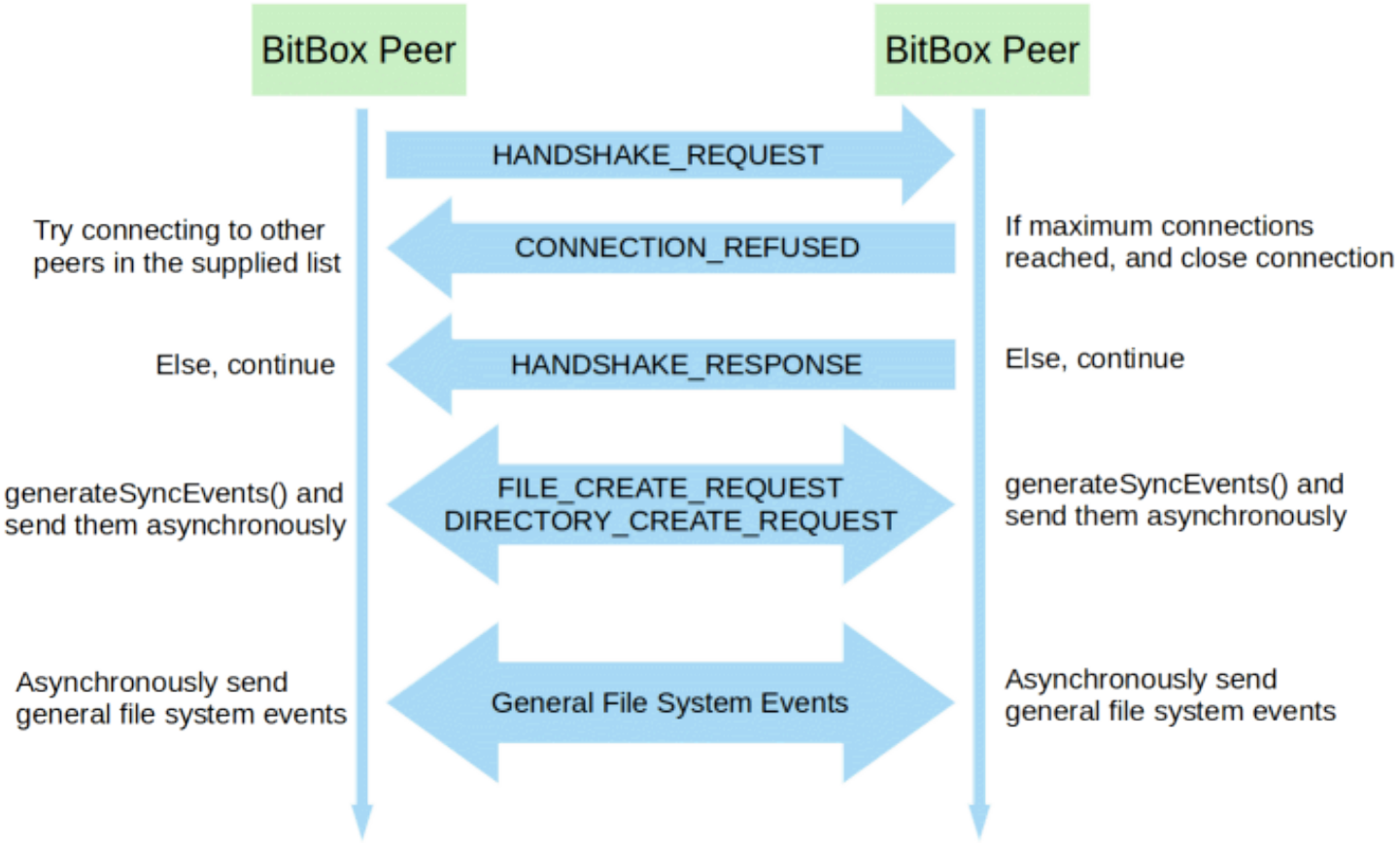
File contents will be transmitted inside JSON using Base64 encoding.

Interactions will in the most part be asynchronous request/reply between peers.

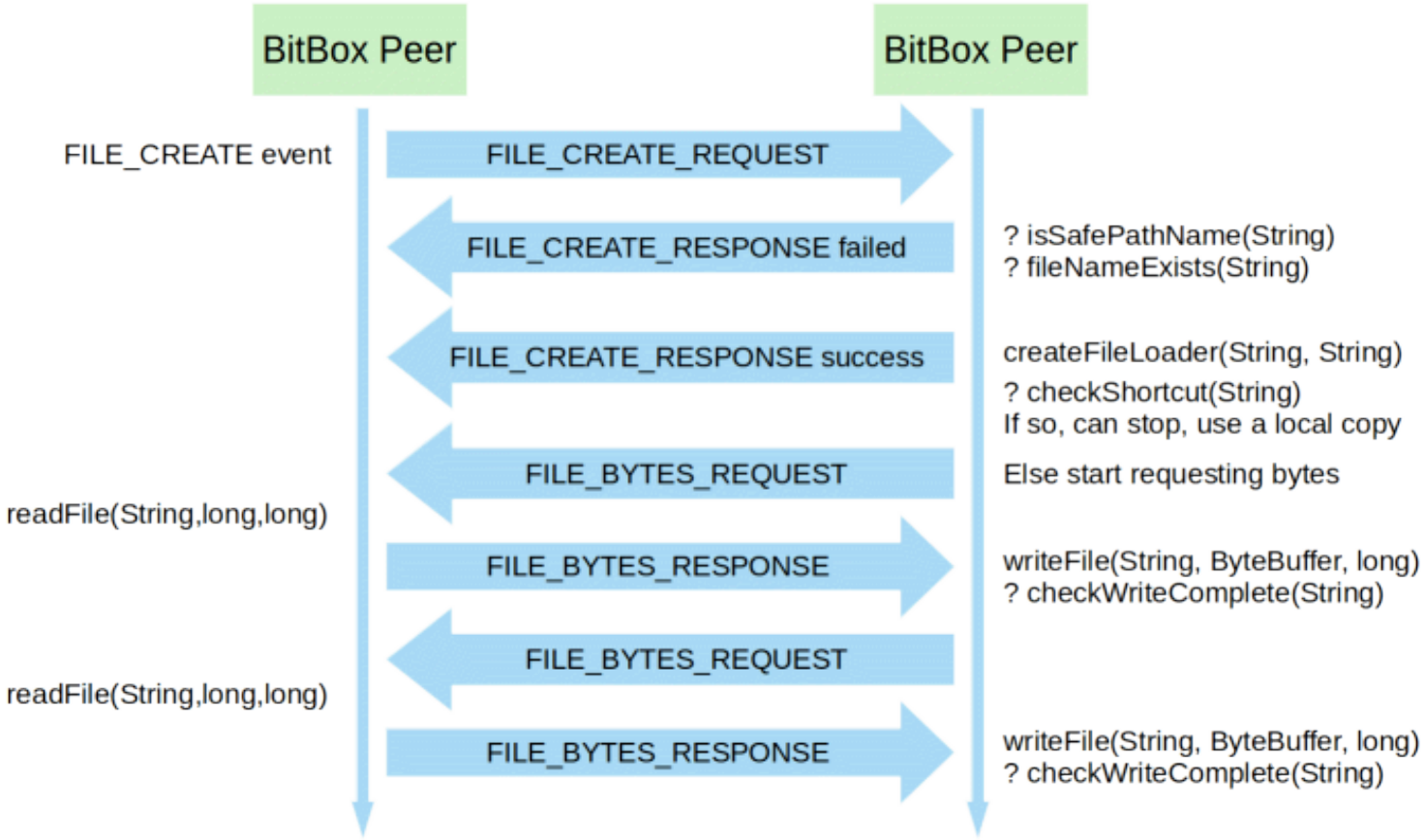
Peer Protocol Messages

- **INVALID_PROTOCOL**
- **CONNECTION_REFUSED**
- **HANDSHAKE_REQUEST**, **HANDSHAKE_RESPONSE**
- **FILE_CREATE_REQUEST**, **FILE_CREATE_RESPONSE**
- **FILE_DELETE_REQUEST**, **FILE_DELETE_RESPONSE**
- **FILE_MODIFY_REQUEST**, **FILE_MODIFY_RESPONSE**
- **DIRECTORY_CREATE_REQUEST**, **DIRECTORY_CREATE_RESPONSE**
- **DIRECTORY_DELETE_REQUEST**, **DIRECTORY_DELETE_RESPONSE**
- **FILE_BYTES_REQUEST**, **FILE_BYTES_RESPONSE**

High Level Protocol



Detailed Protocol Example



INVALID_PROTOCOL

An **INVALID_PROTOCOL** message is sent to a peer when:

- The peer has sent a message which should not be sent, e.g. a **HANDSHAKE_REQUEST** after handshaking has already been completed.
- A message does not contain required fields of the required type.

After sending an **INVALID_PROTOCOL** message to a peer, the connection should be closed immediately.

E.g. if a message is received that does not contain a command field as a string:

```
{
    "command": "INVALID_PROTOCOL",
    "message": "message must contain a command field as string"
}
```

NOTE: your peer should not crash/break due to incorrect messages being received on a connection. You must implement full checking of all messages and attempt to be as robust as possible.

CONNECTION_REFUSED

If a peer receives a new connection from another peer but has already reached its maximum number of incoming connections, `maximumIncommingConnections`, then it responds with a **CONNECTION_REFUSED** message and closes the connection. The `peers` field lists its existing connections.

```
{
    "command": "CONNECTION_REFUSED",
    "message": "connection limit reached"
    "peers": [
        {
            "host" : "sunrise.cis.unimelb.edu.au",
            "port" : 8111
        },
        {
            "host" : "bigdata.cis.unimelb.edu.au",
            "port" : 8500
        }
    ]
}
```

The peer that tried to connect should do a breadth first search of peers in the `peers` list, attempt to make a connection to one of them.

HANDSHAKE_REQUEST

A **HANDSHAKE_REQUEST** message is the first message that a peer will send to another peer upon connecting. It will either receive a **CONNECTION_REFUSED** or a **HANDSHAKE_RESPONSE**, in the later case it can proceed to enter the asynchronous part of the peer protocol. The `hostPort` field gives the address of the peer that is connecting.

```
{
    "command": "HANDSHAKE_REQUEST",
    "hostPort" : {
        "host" : "sunrise.cis.unimelb.edu.au",
        "port" : 8111
    }
}
```

HANDSHAKE_RESPONSE

A **HANDSHAKE_RESPONSE** is sent in response to a **HANDSHAKE_REQUEST** and signals that the peer is ready to go to the asynchronous part of the protocol. The `hostPort` field is in this case the name of the peer that was connected to, i.e. the one responding.

```
{
    "command": "HANDSHAKE_RESPONSE",
    "hostPort" : {
        "host" : "bigdata.cis.unimelb.edu.au",
        "port" : 8500
    }
}
```

The handshake response is the only response that does not contain a `status` and `message` field.

FILE_CREATE_REQUEST

A **FILE_CREATE_REQUEST** is sent to every connected peer when the File System Manager generates a **FILE_CREATE** event. The `fileDescriptor` and `pathName` fields should be as given in the **FILE_CREATE** event.

```
{
    "command": "FILE_CREATE_REQUEST",
    "fileDescriptor" : {
        "md5" : "074195d72c47315efae797b69393e5e5",
        "lastModified" : 1553417607000,
        "fileSize" : 45787
    },
    "pathName" : "test.jpg"
}
```

The receiving peers should call the File System Manager to determine if the file can be created or not. If possible the File System Manager will get ready to receive the file contents, which needs to be requested from the sending peer in blocks of `blockSize`.

FILE_CREATE_RESPONSE

The **FILE_CREATE_RESPONSE** is sent in response to a **FILE_CREATE_REQUEST**. It has all of the same parameters plus, as for all response messages, a `message` and `status` field. The `status` field is true if the request was successful and false otherwise. The `message` is mostly for debugging purposes.

```
{
    "command": "FILE_CREATE_RESPONSE",
    "fileDescriptor" : {
        "md5" : "074195d72c47315efae797b69393e5e5",
        "lastModified" : 1553417607000,
        "fileSize" : 45787
    },
    "pathName" : "test.jpg",
    "message" : "file loader ready",
    "status" : true
}
```

Other response messages, when failing, can include e.g. "unsafe pathname given", "there was a problem creating the file", and "pathname already exists".

FILE_BYTES_REQUEST

The peer that received a **FILE_CREATE_REQUEST**, having called the File System Manager and successfully established a file loader will need to request the bytes of the file from the sender, using a **FILE_BYTES_REQUEST** message. Up to at most `blockSize` bytes (using field `length` in the request message) at a time will be requested, meaning for larger files there will be several request messages.

```
{
    "command": "FILE_BYTES_REQUEST",
    "fileDescriptor" : {
        "md5" : "b1946ac92492d2347c6235b4d2611184",
        "lastModified" : 1553417607000,
        "fileSize" : 6
    },
    "pathName" : "hello.txt",
    "position" : 0,
    "length" : 6
}
```

A peer that receives this message needs to call the File System Manager to read the requested bytes and package them into a response message.

FILE_BYTES_RESPONSE

The **FILE_BYTES_RESPONSE** contains, among other things, the bytes encoded in Base64 format, in the `content` field. These bytes need to be written to the file on the receiving peer.

```
{
    "command": "FILE_BYTES_RESPONSE",
    "fileDescriptor" : {
        "md5" : "b1946ac92492d2347c6235b4d2611184",
        "lastModified" : 1553417607000,
        "fileSize" : 6
    },
    "pathName" : "hello.txt",
    "position" : 0,
    "length" : 6,
    "content" : "aGVsbG8K"
    "message" : "successful read",
    "status" : true

}
```

Other response messages, when failing, can include e.g. "unsuccessful read".

FILE_DELETE_REQUEST

A **FILE_DELETE_REQUEST** is sent to every peer when the File System Manager generates a **FILE_DELETE** event.

```
{
    "command": "FILE_DELETE_REQUEST",
    "fileDescriptor" : {
        "md5" : "074195d72c47315efae797b69393e5e5",
        "lastModified" : 1553417607000,
        "fileSize" : 45787
    },
    "pathName" : "test.jpg"
}
```

The receiving servers should call the File System Manager to determine if the file can be deleted or not.

FILE_DELETE_RESPONSE

In this example of a **FILE_DELETE_RESPONSE**, the peer is responding to say that it could not delete the file because it did not exist.

```
{
    "command": "FILE_DELETE_RESPONSE",
    "fileDescriptor" : {
        "md5" : "074195d72c47315efae797b69393e5e5",
        "lastModified" : 1553417607000,
        "fileSize" : 45787
    },
    "pathName" : "test.jpg",
    "message" : "pathname does not exist",
    "status" : false
}
```

Other response messages, when failing, e.g. can include "unsafe pathname given" and "there was a problem deleting the file". Successful message can be "file deleted".

FILE_MODIFY_REQUEST

The **FILE_MODIFY_REQUEST** is used to modify an existing file and will be sent to all peers when the File System Manager emits a **FILE_MODIFY** event. Transmitting the modification involves the same mechanism as transmitting the contents for a created file.

```
{
    "command": "FILE_MODIFY_REQUEST",
    "fileDescriptor" : {
        "md5" : "d35eab5dd9cb8b0d467c7e742c9b8c4c",
        "lastModified" : 1553417617000,
        "fileSize" : 46787
    },
    "pathName" : "test.jpg"
}
```

FILE_MODIFY_RESPONSE

The **FILE_MODIFY_RESPONSE** merely indicates that the file modification request was successful. Subsequent messages to obtain file bytes, exactly the same as file creation, may or may not be needed.

```
{
    "command": "FILE_MODIFY_RESPONSE",
    "fileDescriptor" : {
        "md5" : "074195d72c47315efae797b69393e5e5",
        "lastModified" : 1553417607000,
        "fileSize" : 45787
    },
    "pathName" : "test.jpg",
    "message" : "file loader ready",
    "status" : true
}
```

Other response messages, when failing, e.g. can include "unsafe pathname given", "file already exists with matching content" (i.e. matching to what the modification would have produced), "there was a problem modifying the file", "pathname does not exist".

DIRECTORY_CREATE_REQUEST

A simple request to create a directory. The File System Manager will always emit events in an order that does not break causality in terms of creation/deletion events of directories and files. So in this example, the directory `dir/subdir` will already exist.

```
{
    "command": "DIRECTORY_CREATE_REQUEST",
    "pathName" : "dir/subdir/etc"
}
```

DIRECTORY_CREATE_RESPONSE

If the directory already exists, still the response status would be false to indicate that it could not be created, otherwise it would be true.

```
{
    "command": "DIRECTORY_CREATE_RESPONSE",
    "pathName" : "dir/subdir/etc",
    "message" : "pathname already exists",
    "status" : false
}
```

Other response messages include "directory created" when successful and "unsafe pathname given", "there was a problem creating the directory" and "pathname already exists" when failing.

DIRECTORY_DELETE_REQUEST

A simple request to delete a directory. In this case the directory being deleted is `etc``1 which is inside`dir/subdir```.

```
{
    "command": "DIRECTORY_DELETE_REQUEST",
    "pathName" : "dir/subdir/etc"
}
```

DIRECTORY_DELETE_RESPONSE

Finally the **DIRECTORY_DELETE_RESPONSE**

```
{
    "command": "DIRECTORY_DELETE_RESPONSE",
    "pathName" : "dir/subdir/etc",
    "message" : "directory deleted",
    "status" : true
}
```

Other response messages include "unsafe pathname given", "there was a problem deleting the directory" and "pathname does not exist" when failing.

Other requirements

- Periodic Synchronization
  - Every `syncInterval` seconds, the BitBox Peer should call `generateSyncEvents()` to do a general synchronization with all neighboring peers.
- Asynchronous protocol
  - Events should not be blocked by other events, e.g. a large file transfer should not prevent other events from being processed until the end of the transfer.

Running the BitBox Peer

The BitBox Peer must be run according to the following:

`java -cp bitbox.jar unimelb.bitbox.Peer`

All of the required libraries must be included (packaged) into the `bitbox.jar`. The peer must read its configuration from a file called `configuration.properties` in the current working directory which follows Java's simple properties file format:

```
path = test
port = 8111
advertisedName = localhost
peers = localhost:8112,localhost:8113
maximumIncommingConnections = 10
blockSize = 1048576
syncInterval = 60
```

`path` is the directory for the File System Monitor to share, `port` is the port number that the peer will bind to, `advertisedName` is the name that the peer will send when handshaking, `peers` is a list of other peers the peer will attempt to connect to when starting up, `maximumIncommingConnections` is the maximum number of incoming connections that the peer will allow (after which it refuses), `blockSize` is the largest number of bytes that will be requested in a file bytes request, and `syncInterval` is the number of seconds between calling `generateSyncEvents()`.

Technical aspects

- **NOTE: be extremely careful when developing on your laptop/computer that contains important data. Incorrect or buggy implementations, or slips of the keyboard when testing, can lead to data loss, including the loss of your development effort. This wont be accepted as a reason for extension to the deadline.**
- Requires Java 1.8.
- Your program must compile using the maven compiler, to produce a bitbox.jar in the target directory.
- Make sure to use a JSON parser/formatter to generate correct JSON messages. A `Document` class has been provided for you to use, that can be very convenient.
- Use the Base64 Java library for encoding/decoding file content.
- Everyone should implement the same protocol, which means that BitBox peers from different groups should interoperate fine.

Your Report

- Use 10pt font, double column, 1 inch margin all around.
- On the first page, clearly show your group's name and the names of all members in the group. Clearly show the login names with university emails as well. The members of the group MUST match the information entered into LMS.
- The report is aimed at addressing a number of questions discussed next. Have one section for in the report for each.
- Figures in the report, including examples of messages and protocol interaction, and any pseudo-code (that you may or may not use), are not counted as part of the word length guidelines.

Introduction

Write roughly 125 words to briefly describe in your own words:

- what the project was about
- what were the technical challenges that you faced building the system
- what outcomes did you achieve

Fault detection/tolerance/recovery

Consider the system's ability, or lack thereof, to recover from faults, such as IO exceptions, both locally at a peer and globally as a distributed file system. In roughly 375 words:

- identify sources of faults that can have the greatest impact on the system, i.e. to disrupt it from working as intended
- describe how the system attempts to detect/tolerate/recover from these faults, if at all
- suggest revisions to the protocol that may overcome these problems

Scalability

There are a number of aspects of the system that present a scalability challenge. In roughly 375 words:

- identify aspects of the system that present problems for scalability
    - be specific with why it is not scalable
- suggest revisions to the protocol that may overcome these problems

Security

The system is designed to allow peer connections, essentially from untrusted third parties. This poses significant security problems since it is unclear whether those third party peers will operate as expected.

In roughly 375 words describe:

- discuss the security issues surrounding the reading and writing of files/directories to a file system in this kind of P2P system,
- how does the current system, including considering the File System Manager implementation and the protocol specification, address these security problems?
- what more could be done to address any outstanding security problems in this regard

Other Distributed System Challenges

Choose a third distributed system challenge that relates to the system and write roughly 375 words explaining how it relates, how the system currently addresses the challenge (if it does at all), and how you might change the system to improve it with respect to the challenge.

Submission

You need to submit the following via LMS:

- Your report in PDF format *only*.
- Your bitbox.jar
- Your source files in a .ZIP or .TAR archive *only*, that can be compiled using maven.

Submissions will be due at the end of Week 8 (does not count the Easter holiday non-teaching period) via a group submission and more details will be posted on LMS nearer to the deadline.