THE UNIVERSITY OF SYDNEY

COMP2017 9017 Assignment 3

Due: 23:59 19 May 2024

This assignment is worth 20% of your final assessment

Assignment 3 - ByteTide - 20%

You are tasked with constructing a P2P File-Transfer program that will allow sending, receiving and detection of anomalous data chunks. The activity that your program will participate in will handle the following activities:

- Loading a configuration
- Loading package files and parsing their format
- Checking the integrity of the file matching to the configuration file's path
- Managing many files that are completed and incomplete
- Complies with a network protocol to communicate with peers
- Informing a client program of the latest information of your peer and the files it manages
- Finalise and check that downloaded files match expected outcome
- Elegantly handle shutdown and disconnections from peers.

To reiterate, the program's aim is to manage files, check integrity of chunks and files, share files and chunks, handle peer connections and requests. Unlike other file-transfer programs, there are no tracker or relay systems in place. A Peer is another program complying with this specification. It will need to implement the configuration, integrity checking, network protocol and object management.

It is advisable that while reading this document that you also refer to the glossary if you do not understand certain terms outlined in this document.

We strongly recommend reading this entire document at least twice. You are encouraged to ask questions on Ed after you have first **searched**, and checked for updates of this document. If the question has not been asked before, make sure your question post is of **"Question"** post type and is under **"Assignment" category** $\rightarrow$ **"A3"** subcategory. Please follow the staff directions for using the question template. As with any assignment, make sure that your work is your own[1], and that you do not share your code or solutions with other students.

It is important that you continually back up your assignment files onto your own machine, flash drives, external hard drives and cloud storage providers (as private). You are encouraged to submit your assignment regularly while you are in the process of completing it.

[1]Not GPT-3/4's, ChatGPT's or copilot's, etc.

1 Part 1 - ByteTide Package Loader & Merkle Tree

To get started, you will need to be able to parse a `.bpkg` file and load it. To assist you with writing your code and complying with the test program, you are advised to complete the `pkgchk.c` file in the `src` directory.

1.1 The Package and its File Format (.bpkg)

In this program, a file is composed of several anomalous data chunks. The chunks are organised in a specific way such that when they are combined, the entire contents of the file can be constructed and presented to the user. A `package` defines the necessary information and resources required to construct the contents of a file. Packages represent a unique file given by an identifier string `ident`.

The package file format is a text format that will need to be parsed by your program. The package file format has the following fields. Please refer to the `hash` and `chunk` parts of the glossary. To also clarify, the package file, can be modelled as a binary tree, the term `h`, refers to the height of the tree in this instance.

- `ident`, hexadecimal string (1024 characters max), the identifier is used within the network to identify the same packages.
- `filename`, string (256 characters max), This is used to help save and locate the file to update when data is sent to it.
- `size`, uint32_t, specifies the size in bytes
- `nhashes`, uint32_t, specifies the number of hashes that are pre-computed from the original file. There must be only 2^(h-1)-1 hashes which will correspond to the hashes of all non-leaf node
- `hashes`, string[2^(h-1) - 1] (64 characters for each string), these correspond to the number hashes in the previous `nhashes` field.
- `nchunks`, uint32_t, specifies the number of chunks. The number of chunks must be a 2^(h-1) value.
- `chunks`, struct[2^(h-1)], each chunk have the fields: hash, offset and size.
  - hash refers to a string (64 characters), corresponding to the datablock hash value
  - offset, uint32_t, is the offset within the file
  - size, uint32_t, is the size of the chunk in bytes

The format below gives an outline to the structure of a `.bpkg` file. Refer to the `resources` folder in the scaffold for a real example.

```
ident: <identifier>
filename: <filename>
size: <size in bytes>
nhashes: <number of hashes that are non-leaf nodes>
hashes:
    "hash value"
    ...
nchunks: <number of chunks, these are all leaf nodes>
chunks:
    "hash value",offset,size
    ...
```

1.2 Package Loading

The focus of this task is to load the `.bpkg` file and also store the details into a merkle tree. Please refer to Section 1.3 for information on a merkle tree.

- Read and load `.bpkg` files that comply with the format outlined in **Section 1.1**
- Once the `.bpkg` has been loaded successfully, it is **advisable** that your program also knows if the file exists or not and has functionality to construct a file of the `size` outlined in the file. Refer to `pkgchk.c:bpkg_file_check` function.
- Implement a merkle tree. Use the data from a `.bpkg` to construct a merkle-tree Refer to **`pkgchk.c:bpkg_get_all_hashes`** and **`pkgchk.c:bpkg_get_all_chunk_hashes_from_hash`** functions, as you should be able to satisfy these operations after implementing a merkle tree **without any IO on the data file**.
- Computing the merkle tree hashes, ensuring that combined hashes match the parents hashes when computed and finding minimum completed hashes. Refer to **`pkgchk.c:bpkg_get_completed_chunks`** and **`pkgchk.c:bpkg_get_min_completed_hashes`** functions. You will need to perform validation on the chunks and discover portions of the file.

  The above verifies chunks against package files and the data's integrity.

1.3 What is a merkle tree?

Binary Tree A merkle tree is a variation on a **binary tree**. A binary tree is tree data structure, where a node is compose of the following.

- It holds a value/data
- Usually implemented to hold a key as well (Key-Value/Map Data Structure)
- Connected to two other nodes that are referred to as `children`. These are referred to as `left` and `right` nodes.

A common structure within C for a binary tree node is as follows.

```
struct bt_node {
        void* key;
        void* value;
        struct bt_node* left;
        struct bt_node* right;
};
```

The above node, holds a `key` that will allow it to be `searchable` with the rule that it must be **unique**. It also holds a `value`, which can be assigned to arbitrary data.

Please Note: When building a tree with a `key` field that allows you to perform a search an efficient tree search, you will need to ensure that your tree is using an appropriate function for the job. **Hint**, if your tree is going to be multi-purpose, consider giving your tree a function pointer to compare the key.

To navigate and/or traverse a tree, you'd be advised to traverse it in `in-order` traversal. Please make sure refer to your tree traversals. Please refer to the following documents to revise on tree-traversals:

- Tree-Traversal - Wikipedia
- Visualgo - BST

Qualities of a merkle tree A merkle tree must is typically a `perfect` or `full and complete` binary tree but it can also be represented as a just a `complete` binary tree (Refer to `Errata, Variations and Notes`).

- Given a depth of `d`, the total number of nodes in your tree will be `2^d - 1`
- All levels are full (necessary for a perfect binary tree).
- A merkle tree will have 2^(d-1) nodes at depth `d`, these will refer to your chunks.
- A merkle tree will have `2^(d-1) -1` non-leaf-nodes.
- All leaves have the same depth (no skewing)

All nodes in a merkle tree have a hash value. Hashes of a leaf node corresponds to a hash value of a data chunk. This value is derived from computing hash value of the data chunk itself.

All other non-leaf nodes derive their hash value by hashing their children's hash values together.

Lets break down the above diagram.

- L1-L4 are data blocks, these refer to `chunks` in a file.
- Your leaf nodes `0-0` to `1-1` use a `hash` function to compute the hash of those data blocks. Given this part already, we have enough information to validate individual blocks.

  Pseudocode Example: `self.hash = Hash(DataBlock[i])`
- Your non-leaf nodes `0`, `1` and root, compute their hashes by combining the hash of their children into a long string and compute the hash of that (Refer: `Errata, Variations and Notes`)

  Pseudocode Example: `self.hash = Hash(left.hash + right.hash)`

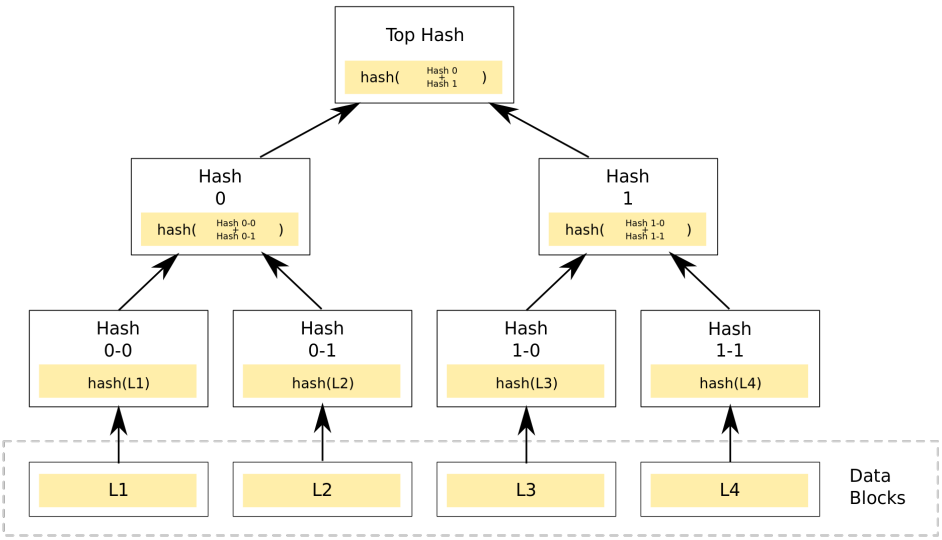

Figure 1: Merkle Tree - Wikipedia

The following is in relation to the `.bpkg` file and your merkle tree's construction. You will have an `expected` hash value stored by your nodes and a `computed` hash value that you can use to 1) compute the hash on datablocks if it is a leaf node, or 2) compute the hash from the concatenation of left and right node hashes if it is a non-leaf node.

The following is an expansion of the operations. We are going through an example of computing the hash of root node of a tree with 7 nodes (similar to the diagram):

Expansion Pseudocode, with steps:

```
We need to compute the hash of the left and right child
1. Hash(root) = Hash(
        Hash(root.left) + Hash(root.right)
    )

Since left and right child are not leaf nodes, we need to do it again
2. Hash(root) = Hash(
            Hash(
                Hash(root.left.left) + Hash(root.left.right)
            )
            +
            Hash(
                Hash(root.right.left) + Hash(root.right.right)
            )
    )
```

```
We have found the leaf nodes
Compute the hash of the data blocks, the size is the chunk size as
outlined in the .bpkg
3. Hash(root) = Hash(
            Hash(
                Hash(DataBlock[0]) + Hash(DataBlock[1])
            )
            +
            Hash(
                Hash(DataBlock[2]) + Hash(DataBlock[3])
            )
     )

We concatenate the leaf children hashes that is assigned to their
`computed` field
4. Hash(root) = Hash(
            Hash(
                root.left.left.computed + root.left.right.computed
            )
            +
            Hash(
                root.right.left.computed + root.right.right.computed
            )
     )
Once again, concatenate the children hashes and compute the hash
of that
5. Hash(root) = Hash(
            root.left.computed + root.right.computed
        )
```

To help you get started, you can use the following struct as well as some helpful scaffold data.

```c
struct merkle_tree_node {
        void* key;
        void* value;
        struct merkle_tree_node* left;
        struct merkle_tree_node* right;
        int is_leaf;
        char expected_hash[64]; //Refer to SHA256 Hexadecimal size
        char computed_hash[64];
};

struct merkle_tree {
        struct merkle_tree_node* root;
        size_t n_nodes;
};
```

Feel free to add and modify the struct above.

Do note You can construct a merkle tree that isn't a perfect binary tree. However, this may make management of your data more difficult. Refer to **Errata, Variations and Notes**.

Do note Please make sure when you compute the hash, you use the **hexadecimal representation**. This is very important for non-leaf nodes that are computing the hash from an ordered concatenation of their children (left + right) hashes.

1.4 Errata, Variations and Notes

- Implementations: It isn't necessary for a merkle tree to be a full or complete binary tree. You could potentially have a merkle tree with more than 2 children Or not all leaf nodes are on the same level

  However, we have made this assumption to help simplify the data structure.

- Same-chunks, different positions: As through experimenting, you may have found that if you have chunks that contain the same data, in this case. Your implementation will need to either assume this will not happen or contain necessary data to differentiate it.

  - Please refer to `REQ` packet, specifically `offset` part to help resolve searches.
  - You can have a bit-field key alongside this similar to the diagram in the previous sections.

- More data than needed: For the most part, the file has been provided with more data than required to help with implementing this data structure but also ensure that other parts aren't restricted if it is in an incomplete state.

- Using hexadecimal hash or byte-hash: The staff implementation uses the hexadecimal hash and while computing with the byte-hash is not-incorrect, it will yield different results to the test cases. Please make sure you comply with this.

1.5 Checklist

- Parse valid `.bpkg` files, ensure you can read each field of them.
- Construct a `merkle tree` from the `bpkg` files after parsing.
- Implement all the functions `pkgchk.c`.
- Run and compile `make pkgchk.o` and that will be able to compile `pkgchk.c` (*Required for test cases*)
- You are free to modify the `Makefile` to refer to your c files you will use in your build targets.
- Run and compile `make pkgchecker`, and compile against `pkgmain.c` to test your program locally.

2 Part 2 - Configuration, Networking and Program

You are now tasked with writing a program that will facilitate P2P file-transfer. Your program will need to complete the following tasks:

- Load a basic configuration file, your program will need to maintain the directory path it will store
- Implement and comply with the protocol to communicate with other peers within the network itself.
- Implement the commands for your program, these will include connecting. Your program will need to connect, disconnect and retrieve peer information. Handle package loading and removing, retrieval of chunks from other peers.

This part deviates a little from part 1 as it is not completely necessary to build a merkle tree to get started on this part, let alone complete it. **However** It is necessary to be able to load a `.bpkg` file, retrieve the `ident`, `filename`, `size`, `nchunks` and `chunks` themselves for this part.

The scaffold has provided the following files for the next sections for you to implement.

- **`src/peer.c`**, - Write peer management code here
- **`src/package.c`** - Write your package management logic here
- **`src/config.c`** - Write your configuration logic here
- **`src/btide.c`**, Contains the main function, starting point of the program.

You are still free to change and alter the contents of the `src` folder how you see fit, however, your `Makefile` still needs to build the required targets.

Make sure your program is able to be build with `make btide`, this should produce an executable.

2.1 Configuration File

Your program will need to parse and load a configuration file that will be used to setup folder that it will either need to create or, if it exists, load existing packages from and refer to an existing file.

Your configuration file will be passed to your program via command line arguments.

```
./btide config.cfg
```

The program's configuration file will use the following information:

- `directory`, string, path local to the system that store `.bpkg` files and the files that are mapped in there. If the directory does not exist, the program should attempt to create it. If the program is unable to create the directory or it is a file, the program should exit with exit code `3`.

- `max_peers`, int, this field is the number of peers the program can be connected to. It is a value between [1, 2048]

  If the `max_peers` value is set to an invalid number, your program should exit, with exit with exit code `4`.

- `port`, `uint16_t`, this field specifies the port the client will be listening on. Acceptable port range (1024, 65535].

  If the `port` value is set to an invalid number, your program should exit, with exit with exit code `5`.

Example of the configuration file:

```
directory:downloads
max_peers:128
port:9000
```

Each field has an action if the constraints of that field are not met as above. If any fields are missing, the configuration should be rejected.

2.2 Network Protocol and Implementation Details

The section below will outline how your program will communicate to other programs on a network. It is advisable that your program also holds data related to peers and packages.

Network Protocol Your program can act as both a server and client to participants among the network. You will need to be able to form a listening socket to accept incoming connections but also have the ability to form new connections.

The network protocol will use 'TCP/IP' data packets to form connections. Your program should use the following packet structure below.

```
union btide_payload {
    uint8_t data[PAYLOAD_MAX];
};

struct btide_packet {
    uint16_t msg_code;
    uint16_t error;
    union btide_payload pl;
};
```

Below are the only `msg_code` values that can be set

```
PKT_MSG_ACK 0x0c
PKT_MSG_ACP 0x02
```

```
PKT_MSG_DSN 0x03
PKT_MSG_REQ 0x06
PKT_MSG_RES 0x07
PKT_MSG_PNG 0xFF
PKT_MSG_POG 0x00
```

Your program can send and receive the following packet types and their byte code value. All packets should be 4096 bytes, and their payload data (if not empty) should follow the order as specified below as the testing system expect data in this format. Padding is not required between different payload parameters.

- `ACP` (`0x02`), when a peer connects to your program, you will need to acknowledge that you have accepted the connection so it can confirm it can add it to its own peer list. If your program connects to peer, you should wait for an `ACP` message back before you add the peer to your own peer list. If the peer does not respond, your program can kill the connection.
- `ACK` (`0x0c`), when your program receives an `ACP` packet after a connect, you will send a message back with an `ACK`. This is to simply acknowledge that you have received the message.
- `DSN` (`0x03`), when a peer wants to disconnect from another peer, it will send a message to the peer, telling that it will be disconnecting. The peer that originated the message will close off its socket and end the connection.

  Your program should detect when a shutdown has occurred by a peer that may not sent `DSN`. Please check `man recv` and `man send` for detecting these cases.
- `REQ` (`0x06`), This packet is sent to a peer to request data for a particular chunk. The request packet will send the `<identifier>` (1024 bytes), `<chunk hash>` (64 bytes) and `<offset>` (4 bytes, `uint32_t`) to another peer in the expectation that the peer will send `<data len>` of data back.

  The order in the packet is as follows: `file_offset`, `data_len`, `chunk_hash` and `identifier`.

  `<data len>`

  If the peer sends you a `REQ` packet but you do not have the expected file or chunk available, you will need to inform the requester of an error in the `RES` packet.
- `RES` (`0x07`), This packet is sent to an originator of a `REQ` packet, it will contain: `<identifier>` (1024 bytes), `<chunk hash>` (64 bytes), `<offset>` (4 bytes, `uint32_t`), `<data len>` (2 bytes, `uint16_t`) and most importantly `<data>` (max 2998 bytes).

  The order in the packet is as follows: `file_offset`, `data`, `data_len`, `chunk_hash` and `identifier`.

  The response packet will send data from the chunk to the requester, since the file `<data len>` component may not match the `<req len>` component of a `REQ` request packet, the peer will need to send multiple `RES` packets to satisfy the length of data requested. This is normal as part of the `REQ-RES` flow.

  `<offset>` refers to the offset of the chunk that `<data>` will be written to.

If the peer does not have the data available, a error byte fields should be set to a number > 0. This will notify the requester that the current peer does not have the data requested.

If it is determined, after a REQ-RES conversation has finished, that the chunk sent is invalid, the chunk should be silently ignored.

- PNG (0xFF), This packet is sent out with the intent to check if a peer is still alive. Nominally, this is usually sent out periodically, however in this implementation we will send it out when PEERS command is called.

  No error handling is necessary for this particular packet.

- POG (0x00), This is a pong message that will be sent to the originator of the PNG(0xFF) message.

The SIGPIPE signal could be raised, in particular with multi-threaded solutions (commonly connection per thread solutions) that may not know the connection has been terminated. Make sure your program handles this signal and also detects errors with your sockets as they arise.

All packets are fixed 4096 byte packets. This results in simple packet handling code within your program.

2.3 Building a CLI

To finish and to test your program, you will need to build a command line interface. There are a handful of commands that need to be implemented which also imply a certain amount of book keeping.

Commands Your program must be able to utilise the following commands. The commands are provided via standard input. Your program will stay alive until QUIT command is inputted.

All commands will have maximum of 5520 characters which can fit the command, identifier and path if ever required.

Do note, it is intended that the commands are case sensitive. You can assume command arguments are delimited by exactly one space [2] and should have no leading and trailing characters.

- CONNECT <ip:port>

This command will attempt to connect to a peer within the network itself. Your program will need to construct a socket and attempt to connect to another peer on the network. man 2 socket for more information. Please refer to ACP and ACK packets in the network protocol section.

If the connect command succeeds, your program must output:
`Connection established with peer.`

If the connect command fails, your program must output:
`Unable to connect to request peer`

[2] except for ADDPACKAGE in which the file name may contain spaces

If the ip and port given, have already been connected to and the connection is alive, your program must output: `Already connected to peer`

If a the `ip` or `port` has not been specified, your program should output
`Missing address and port argument.`

- `DISCONNECT <ip:port>`

This command will disconnect from a peer and remove it from a peer list. Please refer to the `DSN` packet in the network protocol section.

If the peer is connected and in your program's peer list, your program must disconnect with the peer and output `Disconnected from peer.`

If the peer does not exist in your program's peer list, your program must output: `Unknown peer, not connected`

If a the `ip` or `port` has not been specified, your program should output `Missing address and port argument.`

- `ADDPACKAGE <file>`

This command will add a package to manage.

If a the `file` has not been specified, your program should output `Missing file argument.`

If the `file` does not exist or you do not have permission to use it, Your program must output: `Cannot open file`

If the `file` is not a valid `bpkg` file, your program must output `Unable to parse bpkg file.`

- `REMPACKAGE <ident, 20 char matching>`

where ident is identifier or partial identifier.

This command will remove a package that is being maintained by the program.

If a the `ident` has not been specified, your program should output
`Missing identifier argument, please specify whole 1024 character or at least 20 characters.`

If the `ident` is not managed by your program, your program should output:
`Identifier provided does not match managed packages`

On success, the program will output `Package has been removed`

- `PACKAGES`

Your program should report on the status of the packages loaded. Your program will have need to maintain a list of packages that have been added in working memory.

If your program is not managing any packages, your program will output
`No packages managed.`

If your program is managing 1 or more packages, your program will output in the following format:

```
[1..N]. <32 char identifier>, <filename> : (INCOMPLETE | COMPLETED)
```

Example:

```
1. 0c4d036a2161aa6525743d44725e6212, song5.mp3 :  INCOMPLETE
2. 13d608773eb8842426fddb8131d5c184, song6.ogg :  COMPLETED
    ...
```

- `PEERS`

Lists all connected peers. This will also trigger a `PNG` packet to be sent to all peers you are connected to on the network.

If your program is not connected to any peers, it will output: `Not connected to any peers`

If your program is connected to 1 or more peers, your program will output in the following format:

```
Connected to:

[1..N]. <ip>:<port>
```

Example:

```
Connected to:

1. 192.168.1.1:9001
2. 192.168.2.120:1723
    ...
```

- `FETCH <ip:port> <identifier> <hash> (<offset>)`

Requests chunks related to the hash given. Please refer to `REQ` and `RES` packets in the network protocol section. If an offset is specified, it will use this additional info to narrow down a hash at a particular offset of the file. The offset will need to match the start of that chunk and must be a number greater than or equal to 0.

If the number of arguments provided does not match 3, program will output:
`Missing arguments from command`

If the ip and port is missing from the peer list, your program will output: `Unable to request chunk, peer not in list`

If the identifier is missing from the package list, your program will output: `Unable to request chunk, package is not managed`

If the hash does not exist in the package, your program will output
`Unable to request chunk, chunk hash does not belong to package`

If the arguments specified are correct, a `REQ` packet will be sent to the peer.

- `QUIT`

The program quits, no error should be outputted from this command.

Any erroneous commands will require the program to output `Invalid Input`.

2.4 Checklist

- Implement all the packet types in the Network Protocol Section
- Implement a data structure to add, remove and retrieve peer information.
- Implement a data structure to add, remove and retrieve package information that your peer is managing.
- Implement the commands for your client program.
- Ensure that your packets are compatible between clients, test your program in class or with friends at uni.
- Implement a tests for merkle tree based on A3 tests and any new tests you have derived since.
- Implement a tests for networking based on A3 tests and any new tests you have derived since.

2.5 Implementation Help

This section here is to help give you hints to implement your networking application.

- A simple networking technique is to use thread-per-connection. This means that, when a connection is accepted, it is split off into another thread.

  However, you will need to identify and manage when a connection has been terminated and detect when a thread has finished.

- When managing peers, use a `dynamic array` or refer to the `max_peers` property. However, you will need to keep track of the number of peers that are currently connected

- The struct packet given provides a good hint as to where to add specific packet data information. Consider why a union would be best suited there. In additional, it makes it easy to just use only a single type to handle packets that are sent to your program.

- Handling packets is similar to handling commands via standard input. As long as you got the message, you just need to make the decisions based on the type of packet.

2.6 Resources

You have been supplied additional resources to help with your assessment. These resources will include command line tools and utility functions that you will need to use during the development of your program. Use them to help with testing different components and constructing files test cases.

- SHA256 Implementation (`crypt/sha256.c/.h`), used for hashing data for the merkle tree, Also comes with `sha256` utility program.

- Package Make (`./pkgmake --help`), used for constructing a package file.

- Packet Validator (`./pktval --help`), used for checking basics of packets sent across the network.

- Getting started with networking (`./getting_started/`), folder with basic networking example code.

- Example Packages (`/packages/`), a folder with a variety of example packages and files to test and inspect. You are encouraged to use `xxd` to inspect the files or even break up the files to verify different parts.
- `split` command (`./split --help`), use this command to break apart a file and run it against the `sha256` program included with the assessment.

3 Assumptions

This assessment makes a few reasonable assumptions around how packets and data will be encoded without explicitly outlining it in each section. It is also reasonable to make it clear how communication is to be afforded if between all peers.

For sake of simplicity, file and network binary data is sent or saved as little endian. For singular bytes this does not have any serious bearings however, for integers this is significant to outline as the order of bytes may be different between a BE and LE system with this software.

4 High performance Merkle tree

For the high mark, you would have a working implementation for all parts. If you cannot pass most test cases, this section will not be graded.

You are to optimise your merkle tree implementation to improve one of the following areas of your choosing:

- minimising the time required for the I/O bound problem of large volumes of data to load and hash using multiple threads
- minimising the time required for the CPU bound problem of calculating all hashes of the merkle tree using multiple threads for insertions
- building a merkle tree from a .bpkg file in parallel

For this mark, you will need:

- a benchmark method that can be executed by the grader,
- testing data (ideally procedurally generated from the benchmark),
- a report of 500-1000 words describing which of the above optimisations you are aiming for, and where you applied these in the code.

Ideally, you achieve a speedup proportional to the number of threads and/or size of input. A graph of this behaviour would complement your report.

5 Marking and notes

This assignment will be subjected to manual marking and auto marking. The assessment has been broken up into 3 parts, with one external part that you are able to access.

- Part 1 - Automatic tests 8%
- Part 1 - Own merkle tree tests - Manual 2%
- Part 2 - Automatic tests 6%
- Part 2 - Own networking tests - Manual 1%
- Code Style & Comments - Manual 1%
- You are required to construct a README.md documentation to describe the organisation of your software, where test data is located and how they are run (300 - 1500 words) - Manual 1%
- Part 1 & 4 - Performance benchmark and reporting - Manual 1%

Deductions apply when:

- There exists a significant mismatch between A3tests and submitted tests. Only in the case where the number of tests described in the A3tests report are far *greater* than what is being tested in this final submission. For example, if there are 50 tests described in A3tests and only 10 are implemented in the final, this would be a problem and deductions would apply.
  Up to 5/20 A3 marks may be deducted.
- Final `git` repository is UNCLEAN.
  Up to 2/20 A3 marks will be deducted when the final `git` repository is unclean.
  Essentially, you are submitting a `git` repository to EdStem. Make sure the final `git` repository (e.g., all `git commits` of this repository) submitted contains **ONLY** source files, header files, test files, documentation in text files. **Ask** on EdStem if you need to commit other files. Suggestions: (1) Execute `git status` frequently, before each `git add` and `git commit`. (2) Never `git add .` or `git add -A`. (3) Include a proper `.gitignore` file. (4) Read previous guides and discussions on EdStem.
- Memory errors and memory leaks occur.
  1/20 A3 marks will be deducted for EACH occurrence of memory errors or memory leaks. This deduction will be capped at 5/20 A3 marks. Memory errors and leaks are determined by `Valgrind`.
  Can be detected with `Valgrind` and `ASAN`. Guides are available on EdStem. **Ask** on EdStem if you cannot find the guides or need help.
- Dynamic memory and shared memory are not utilised in implementations. E.g., When **ONLY** file IO is utilised, while `mmap` and heap memory are not used.
  20/20 A3 marks may be deducted.
- VLAs (Variable-length array) are used. Add `-Wall -Wvla` to compilation arguments.
  1/20 A3 marks will be deducted for EACH VLA occurrence (VLA definition). This deduction will be capped at 5/20 A3 marks.
- There exists non-English comments, or notes, presented in any part of the submission.
  1/20 A3 marks will be deducted for EACH line. This deduction will be capped at 5/20 A3 marks.
- There exists the use any external libraries, other than those in `glibc`.
  20/20 A3 marks may be deducted. **Ask** on EdStem before using any external libraries.

Other restricted functions may come at a later date.

Academic Declaration

By submitting this assignment you declare the following: I declare that I have read and understood the University of Sydney Student Plagiarism: Coursework Policy and Procedure, and except where specifically acknowledged, the work contained in this assignment/project is my own work, and has not been copied from other sources or been previously submitted for award or assessment.

I understand that failure to comply with the Student Plagiarism: Coursework Policy and Procedure can lead to severe penalties as outlined under Chapter 8 of the University of Sydney By-Law 1999 (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the Internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

I realise that I may be asked to identify those portions of the work contributed by me and required to demonstrate my knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

I acknowledge that the School of Computer Science, in assessing this assignment, may reproduce it entirely, may provide a copy to another member of faculty, and/or communicate a copy of this assignment to a plagiarism checking service or in-house computer program, and that a copy of the assignment may be maintained by the service or the School of Computer Science for the purpose of future plagiarism checking.

Changes

- 2024-04-29
  - Minor typographical errors
  - DISCONNECT contained notion of not removing peer from peer list
  - Added optional offset parameter to FETCH command to help with specificity.