HIGH LEVEL DESIGN

1. **Document Information**	
Document Name	High Level Design document
Service	
Author	Sreenath Kumar Mopuri
Contributors	Sreenath Kumar Mopuri
Issue Date	20/09/2024

2. **Document History**

Version	**Date**	**Summary of change**	**Reference ID**
1.0	20/09/2024	Initial draft	

Contents

Problem Statement:

We have a file with millions of records, each containing either an IPv4 address, an IPv6 address, or an invalid string. We need to process the file to:

1. Count total valid IPv4 addresses.
2. Count total valid IPv6 addresses.
3. Count unique IPv4 addresses.
4. Count unique IPv6 addresses.
5. Count invalid IP address strings.

1 Functional Requirements

1.1 Input

A file containing millions of lines, each representing an IP address (IPv4, IPv6) or an invalid string.

1.2 Output:

- Total IPv4 count.
- Total IPv6 count.
- Unique IPv4 count.
- Unique IPv6 count.
- Invalid IP count.

2 Non-Functional Requirements

2.1 Efficiency

As the file contains millions of records, efficient file handling and memory management are critical.

2.2 Scalability

The solution should scale for even larger files.

2.3 Concurrency

Optionally, multi-threading can be used to improve performance.

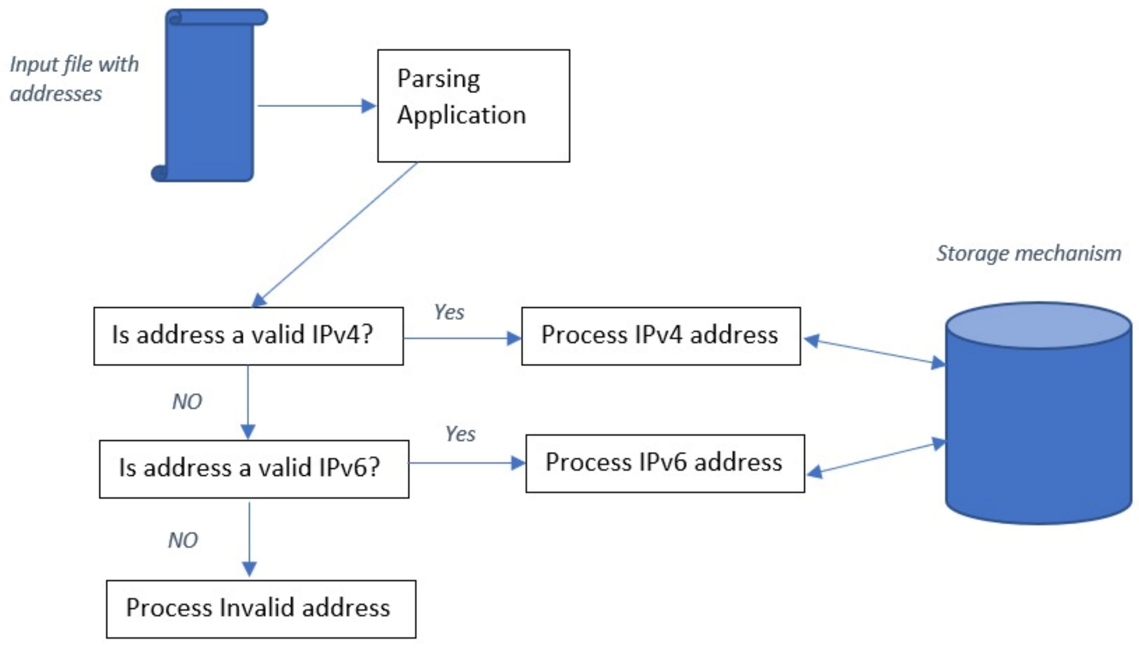
2.4 Memory Usage

Efficient handling of large datasets with mechanisms to prevent excessive memory consumption.

2.5 Error Handling

Invalid strings should be properly handled without breaking the program.

3 High Level Design in C++



3.1 Parallel Processing

To optimize the processing of millions of IP addresses, we can leverage **multi-threading** in C++. The idea is to split the file into chunks and assign each chunk to a separate thread, allowing for concurrent processing of the file. Each thread will process its assigned portion of the file and accumulate the results (IPv4, IPv6, invalid counts, and unique IPs). After all threads have finished processing, the results are merged.

3.2 High-Level Design for Parallel Processing

1. **File Splitting**:
   - Divide the file into chunks of roughly equal size.
   - Each thread will process a chunk of the file by reading its assigned portion.
2. **Thread Management**:
   - Use C++ threads (std::thread) to process each chunk in parallel.
3. **Shared Data**:
   - Each thread will maintain local sets for unique IPv4 and IPv6 addresses to avoid contention.
   - At the end of the process, merge these local sets into global sets for final unique counts.
4. **Synchronization**:
   - Use mutexes to synchronize access to shared data (e.g., total counters for IPv4, IPv6, and invalid IPs) to avoid race conditions.

4 Flame Graph

A **Flame Graph** is a visualization tool used to analyse and understand the performance of an application, especially when dealing with complex, multi-threaded, or CPU-bound code. It helps in identifying where the CPU time is being spent, which functions are most resource-intensive, and whether there are any performance bottlenecks in the code.

4.1 Key Concepts of a Flame Graph:

1. **Hierarchical Stack Visualization**:
    - A flame graph represents a **call stack** in a hierarchical, color-coded format.
    - The **x-axis** represents the **stack profile** (which shows different function calls), and the **width** of each box (flame) represents how much **CPU time** that function or stack frame consumes.
    - The **y-axis** represents the **depth of the call stack**. Functions closer to the bottom of the graph are higher in the call stack (like the main function), and functions higher up in the graph are lower-level (deeper) function calls.
2. **CPU Time and Width of Blocks**:
    - The wider the block, the more time the CPU spent in that function.
    - If a function at the top of the graph has a wide block, it indicates that the function itself consumes significant time. Conversely, if a function lower down the stack is wide, it implies that most of the time is spent in that function and its descendants (i.e., the functions it calls).
3. **Aggregated View**:
    - A flame graph is based on **sampling the stack traces** of all threads over time. This sampling gives an aggregated view of which functions were active during the profiling period and how much time was spent in each.
4. **Interactive Navigation**:
    - Flame graphs are often interactive (in a browser or viewer). By hovering over or clicking on individual frames, you can see details about the function (like its name, total time spent, and location in the source code).

4.2 Structure of a Flame Graph:

- **Functions** are represented by rectangular boxes (flames).
- **Colors** of the boxes don't represent any particular meaning (they're typically random or gradient). However, they help distinguish between different functions.
- **Base of the graph**: The main entry point (e.g., main() function) is located at the bottom.
- **Higher levels**: Functions that are called by other functions are stacked above their callers.

- **Flat or Wide areas**: Areas of the graph where the blocks are wide indicate functions that take a lot of time.

4.3 Use Cases of Flame Graphs:

1. **Performance Tuning**: Flame graphs are useful for finding functions that are consuming excessive CPU or taking a long time to execute. By identifying "hot" functions (functions where most time is spent), developers can target those for optimization.
2. **Bottleneck Identification**: You can use flame graphs to identify slow parts of a program, including inefficient loops, recursive function calls, or excessive locking and synchronization.
3. **Profiling Multi-Threaded Programs**: Flame graphs work well in multi-threaded programs by aggregating the performance data across threads, making it easy to spot which threads or parts of the code are consuming more CPU resources.
4. **Understanding Call Patterns**: Flame graphs help visualize the relationships between function calls, providing insight into how the program is structured and executed.
5. **How Flame Graphs Are Created:**

Flame graphs are typically generated through a process that includes:

1. **Profiling**: The program is run with a profiling tool that periodically samples stack traces. This sampling helps track which functions are being executed at each point in time.
2. **Aggregating Samples**: The collected samples are then aggregated to count how many times each function appears in the call stack during the profiling period.
3. **Rendering**: The aggregated stack data is rendered into a hierarchical, interactive flame graph using tools like **FlameGraph.pl** (a Perl script by Brendan Gregg, who pioneered the concept), or other visual tools.