Media Processing Application

Technical Specification Document

Project Overview

This document specifies the requirements and implementation details for a distributed media processing application that monitors directories for media files, processes them according to specific rules, and maintains a queue system with statistics. The application will be built using Python with a Tkinter GUI interface and packaged as a standalone executable.

Core Functionality

1. **File Monitoring**
   - Monitor specified directories for new media files
   - Identify file types and apply appropriate processing rules
   - Support distributed monitoring across multiple machines
2. **Media Analysis**
   - Analyze media files to identify video codecs, audio tracks, and subtitles
   - Detect Dolby Vision HDR content
   - Identify audio track languages and channel configurations
   - Check subtitle attributes
3. **Metadata Integration**
   - Interface with Sonarr/Radarr to fetch metadata using tt{idnumber}
   - Connect to OMDB API to determine native audio language
   - Use metadata to make processing decisions
4. **Audio Processing**
   - Select audio tracks based on language preferences (native language + eng,dut/nld,tur,und)
   - Remove commentary tracks
   - Convert audio to Opus with bitrate decisions based on channel configuration:
     - 1.0 → 1.0 Opus 32kbps
     - 2.0 → 2.0 Opus 64kbps
     - 2.1, 3.0, 4.0 → 2.0 Opus 64kbps
     - 5.1 (128-384kbps) → 5.1 Opus 128kbps
     - 5.1 (384-640kbps) → 5.1 Opus 256kbps
     - 5.1 (>640kbps) → 5.1 Opus 320kbps
     - 7.1/9.1 (128-384kbps) → 5.1 Opus 128kbps

    - 7.1/9.1 (384-640kbps) → 5.1 Opus 256kbps
    - 7.1/9.1 (>640kbps) → 5.1 Opus 320kbps

5. **Subtitle Management**
    - Keep subtitles in eng, dut/nld, tur, und languages
    - Remove SDH and commentary subtitles
    - Reorder subtitle tracks
6. **Container Management**
    - Run mkvpropedit on the new container
    - Reorder streams (video, audio by language priority and channel count, subtitles by language priority)
7. **Distributed Processing**
    - Inter-instance communication between applications running on different machines
    - Status sharing and coordination
    - Work distribution
8. **Queue Management**
    - Queuing system for pending jobs
    - Currently processing view
    - Completed jobs history with statistics
    - Failed jobs tracking
    - Configurable parallel processing limits per instance
9. **Customizable Commands**
    - User-definable command templates for different content types:
        - Dolby Vision HDR content processing
        - Normal content processing
        - Downscaling content (for files with downscale tag)
10. **User Interface**
    - Start/stop/pause controls
    - Queue management interface
    - Processing status and progress
    - Configuration interface
    - Log viewer
    - Statistics display
11. **Logging & Statistics**

  - Detailed logging of all operations
  - HTML log file generation
  - Processing statistics tracking and persistence
  - Job history maintenance
  - Restart-persistent application state

Technical Implementation

Programming Language & Framework

- Python 3.8+ with Tkinter GUI

Required Python Libraries

- **Built-in Libraries**
  - `tkinter` - GUI framework
  - `threading` - Background processing
  - `queue` - Thread-safe queue implementation
  - `sqlite3` - Local database for settings and statistics
  - `json` - Configuration and state serialization
  - `logging` - Application logging
  - `os`, `sys`, `shutil` - File system operations
  - `socket` - Basic networking
  - `subprocess` - Executing external commands
  - `string` - String templating for commands
- **External Libraries (pip installable)**
  - `watchdog` - File system monitoring
  - `pymediainfo` - MediaInfo wrapper for media file analysis
  - `ffmpeg-python` - FFmpeg wrapper for audio conversion
  - `requests` - HTTP client for API calls
  - `websockets` - Inter-instance communication (alternative to raw sockets)
  - `pyinstaller` - Executable creation

External Dependencies

- FFmpeg - Media processing toolkit
- MediaInfo - Media file analysis

- qsvenc - Video encoding
- mkvpropedit - MKV container manipulation

Application Architecture

1. **Main Application (main.py)**
    - Entry point
    - UI initialization
    - Background services startup
2. **User Interface (ui.py)**
    - Main window setup
    - Tabbed interface (Queue, Statistics, Logs, Settings)
    - Control panel
    - Status updates
3. **Configuration Manager (config.py)**
    - Settings loading/saving
    - Command templates management
    - UI preferences
4. **File Monitor (monitor.py)**
    - Directory watching
    - New file detection
    - File type identification
5. **Media Analyzer (analyzer.py)**
    - Media file inspection
    - Stream identification
    - Language detection
    - API integration (Sonarr/Radarr, OMDB)
6. **Processor (processor.py)**
    - Command execution
    - Audio conversion
    - Container manipulation
    - Progress tracking
7. **Queue Manager (queue_manager.py)**
    - Job scheduling
    - Parallel execution control

    - State persistence

8. **Network Communication (network.py)**
    - Inter-instance messaging
    - Status broadcasting
    - Work distribution
9. **Statistics Manager (statistics.py)**
    - Job history tracking
    - Performance metrics
    - Data visualization
10. **Logger (logger.py)**
    - Console logging
    - File logging
    - HTML log generation

Database Schema

Settings Table

```
CREATE TABLE settings (
    key TEXT PRIMARY KEY,
    value TEXT
);
```

Jobs Table

```
CREATE TABLE jobs (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    filename TEXT,
    status TEXT,  -- 'queued', 'processing', 'completed', 'failed'
    start_time TIMESTAMP,
    end_time TIMESTAMP,
    original_size INTEGER,
    processed_size INTEGER,
    command_used TEXT,
    error_message TEXT,
    instance_id TEXT
);
```

Statistics Table

```
CREATE TABLE statistics (
    date TEXT,
    jobs_completed INTEGER,
    jobs_failed INTEGER,
    total_size_processed INTEGER,
    total_size_saved INTEGER,
    average_processing_time REAL
);
```

User Interface Design

Main Window Layout

```
Main Window
├── Menu Bar (File, Settings, Help)
├── Control Panel (Start/Stop/Pause buttons)
├── Instance Mode Indicator (Master/Slave/Standalone)
├── Tab View
│   ├── Queue Tab
│   │   ├── Currently Processing Section
│   │   ├── Pending Jobs Section
│   │   └── Completed Jobs Section
│   ├── Statistics Tab
│   │   ├── Processing History
│   │   └── File Statistics Charts
│   ├── Network Tab (visible in Master mode)
│   │   ├── Connected Slaves List
│   │   ├── Instance Status Overview
│   │   └── Work Distribution Stats
│   ├── Logs Tab
│   │   └── Real-time Log Viewer
│   └── Settings Tab
│       ├── Folder Monitoring Settings
│       ├── Processing Rules Settings
│       ├── Network Settings
│       ├── Command Configuration
│       │   ├── Dolby Vision HDR Commands
│       │   ├── Standard Content Commands
│       │   └── Downscaling Commands
│       └── Output Settings
└── Status Bar (Processing status, instance info, connection status)
```

Settings Tab Structure

```
Settings Tab
├── General
│   ├── Instance Name
│   ├── Max Parallel Jobs
│   └── Default Output Directory
├── Monitoring
│   ├── Add/Remove Watched Directories
│   └── File Pattern Filters
├── API Integration
│   ├── Sonarr/Radarr API URL & Key
│   └── OMDB API Key
├── Audio Processing
│   ├── Language Priority List
│   └── Channel Configuration Rules
├── Command Templates
│   ├── Dolby Vision HDR Content Command
│   ├── Normal Content Command
│   └── Downscaling Content Command
└── Network
    ├── Instance Discovery Mode
    ├── Port Settings
    └── Known Instances
```

Inter-Instance Communication Protocol

Network Architecture

- **Master Mode**: Central coordinator that distributes work and tracks status
- **Slave Mode**: Connects to a master instance for job coordination

Message Types

- `HELLO` - Instance discovery and capabilities announcement
- `STATUS` - Instance status update
- `JOB_REQUEST` - Request for job information
- `JOB_STATUS` - Job status update
- `JOB_COMPLETE` - Job completion notification with stats
- `SYSTEM_STATUS` - Overall system status report
- `REGISTER_SLAVE` - Slave instance registering with master
- `MASTER_COMMAND` - Command from master to slave
- `SLAVE_RESPONSE` - Response from slave to master

Message Format

```json
{
  "type": "MESSAGE_TYPE",
  "instance_id": "unique-instance-id",
  "instance_role": "master|slave",
  "timestamp": 1620000000,
  "data": {
    // Message-specific payload
  }
}
```

Master-Slave Connection Flow

1. Slave starts up and checks configuration
2. Slave attempts to connect to configured master IP/port
3. Slave sends `REGISTER_SLAVE` message with capabilities
4. Master acknowledges and adds slave to instance pool
5. Regular heartbeat messages maintain connection
6. Master coordinates job distribution across connected slaves

Implementation Plan

1. **Phase 1: Core Application Framework**
   - Set up basic Tkinter application structure
   - Implement settings management
   - Create basic UI layout

2. **Phase 2: File Monitoring & Analysis**
   - Implement directory watching
   - Build media file analysis module
   - Create file type detection logic

3. **Phase 3: Processing Engine**
   - Implement command template system
   - Build audio conversion logic
   - Create subtitle handling

4. **Phase 4: Queue Management**
   - Build job queuing system
   - Implement parallel processing
   - Create job status tracking

5. **Phase 5: Network Communication**
   - Implement instance discovery
   - Build message passing system
   - Create work distribution logic

6. **Phase 6: Statistics & Logging**
   - Implement detailed logging
   - Build statistics collection
   - Create HTML log generator

7. **Phase 7: UI Finalization**
   - Refine user interface
   - Add data visualization
   - Implement all configuration options

8. **Phase 8: Packaging & Testing**
   - Package application as executable
   - Perform comprehensive testing
   - Fix identified issues

Executable Creation

The application will be packaged using PyInstaller:

~~~
```bash
# Basic package
pyinstaller --onefile --windowed --icon=app_icon.ico --name=MediaProcessor main.py

# With additional data files
pyinstaller --onefile --windowed --icon=app_icon.ico --add-data "assets/*:assets" --name=MediaProcessor main.py
~~~

System Requirements

- Python 3.8+ (development)
- Windows 7/10/11 (target deployment)
- FFmpeg, MediaInfo, qsvenc, and mkvpropedit installed
- Network connectivity for distributed mode
- Sufficient storage for media processing

Additional Notes

1. **Error Handling**
   - All external tool calls should have robust error handling
   - Failed jobs should be logged with detailed error information
   - Application should gracefully handle unexpected shutdowns
   - Network disconnections should be detected and recovered from
2. **Performance Considerations**
   - File analysis should be done efficiently to minimize startup time
   - Background processing should be properly threaded to keep UI responsive
   - Database operations should be batched where possible
   - Network communication should be optimized to reduce overhead
3. **Security**
   - API keys should be stored securely
   - Network communication should validate source instances
   - File paths should be sanitized before use in commands
   - Master-slave connections should implement basic authentication
4. **Extensibility**
   - Command template system allows for future expansion
   - Plugin architecture could be considered for future versions
   - Configuration system should be designed for easy extension

5. **Master-Slave Architecture**
    - Master instance coordinates work distribution
    - Slave instances report capabilities and status to master
    - Job allocation considers slave capabilities and current load
    - System handles instances joining and leaving dynamically
    - Master failure should be detected by slaves

This specification document provides a comprehensive overview of the Media Processing Application requirements, architecture, and implementation plan. It serves as a guide for development and can be expanded upon as needed during the implementation process.