Video Processing Algorithms

June 19, 2018

Azam Abidjanov
Jared Hofer
Connor Koch
Johnny Zeng

Table of Contents

1. Introduction
    1.1. Client Description
    1.2. Product Description
2. Requirements
    2.1. Functional Requirements
    2.2. Non-Functional Requirements
3. System Architecture
    3.1. Architecture Diagram
    3.2. Reader/Writer
    3.3. InvertColor
    3.4. Face Recognition
    3.5. Motion Detection
    3.6. ImageChip
4. Technical Design
    4.1. Google Protocol Buffers
    4.2. MPCore
    4.3. Maven
5. Design Decisions
    5.1. Coding Language
    5.2. Computer Vision Library
    5.3. Other Decisions
6. Results
    6.1. Features Implemented
    6.2. Features Not Implemented
    6.3. Performance Testing Results
    6.4. Future Work
    6.5. Lessons Learned
7. Appendices
    7.1. Installation Instructions
    7.2. Code Sources

1. Introduction

1.1 Client Description

Northrop Grumman is a global security company specializing in defense and aerospace technology. It is the fifth largest defense contractor in the world and provides products and solutions in the form of autonomous systems, surveillance and intelligence, and communications, to name a few. Northrop Grumman products play a large role in the evolving defense and space climate of the 21st century. Because of this, they are a key part of the United States' national security.

1.2 Product Description

We were tasked with creating two or more image or video processing applications to use in the Northrop Grumman (NG) proprietary program, Mission Processing Core (MPCore), which is a highly configurable architecture that allows for managing and running data processing workflows. MPCore is used to control data processing for the space-based missile warning and defense missions. The applications must utilize Google Protocol Buffer to be used in the workflow created by MPCore. In general, the video processing applications we create must be able to send and/or receive data from other apps to perform different processes on a video. The applications can be arranged in a workflow or swapped out with others to change the processing of the video. The video must then be displayed to the user after processing has occurred.

2. Requirements

2.1 Functional Requirements

- Create app(s) that are able to read and process video files, filtering and highlighting specific information (moving objects, background etc.).
- The app must take in a video/image and display a video/image
- The programs need to be able to keep up with the flow of data
- On top of this, MPCore requires a CSV wrapper to accompany the applications
- These applications should be able to link together in MPCore, ideally in any order
- Utilize Google Protocol Buffer to send and receive data
- The applications should be developed with a makefile (C++) or Maven (Java)

2.2 Non-Functional Requirements

- The applications should efficiently manage memory being written to the disk.
- The applications should run in a way that show that the applications are processed in real time.
- It's recommended the applications be written in Java or C++.
- Our data types should be defined in Google Protocol Buffer.

3. System Architecture

3.1 Architecture Design

The structure of the projects involves the running of separate apps that can be combined to process and display videos. Each app can be run separately, however, almost all apps stay idle until there is data being sent to them by the Reader app (Figure 1). As seen in Figure 1, the core of the project involves inputting a video and outputting a processed video. The processing involves stringing a series of apps that must compose of a Reader, Writer, and some sort of processing app. The processing apps are interchangeable and can be switched out with different processing apps depending on the data they send and receive from Google Protocol Buffer (explained in 4.1). While the apps work with each other, they also work with the MPCore client through APIs that are implemented within the applications.

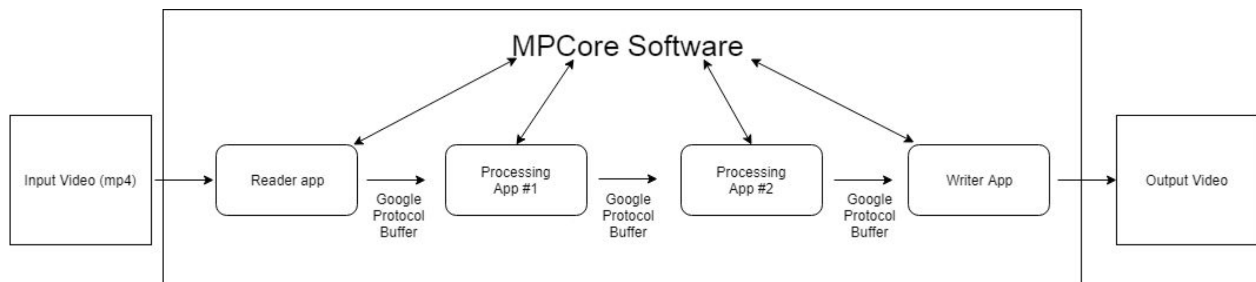

Figure 1: Outline of MPCore Workflow

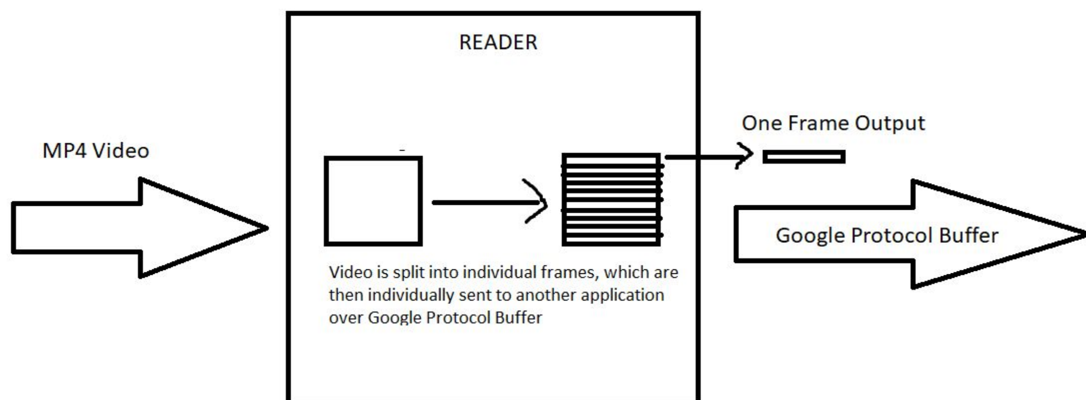

Figure 2: Detailed Outline of Reader App

3.2 Reader/Writer

The Reader is always the starting app in the workflow. It reads in the video given by the user, parses it into individual frames, and outputs the frames to Google Protocol Buffer to be processed by other apps. As shown in Figure 2 above, each pass of main in Reader parses one frame and sends it over the data stream.

The Writer is an app that transforms the frames sent from other apps into a video that is played in real time. The Writer uses JFrame to display one frame and updates the frames with a new image when the image is received instead of creating a video.

3.3 InvertColor

The InvertColor app utilizes OpenCV, a computer vision library, to invert the RGB values of each pixel in the frame.

3.4 Face Recognition

The Face Recognition app makes use JavaCV along with a prebuilt HaarClassifier from the OpenCV.org github repository. The app detects any faces found within a frame and places a hat on top of their head. This app has two versions: one which outputs ImageChips (small, cropped photos focused on faces) of the faces detected, and the other outputs just the modified version of the original image.

3.5 Motion Detection

One of the apps performs motion detection by processing the difference in pixel data between consecutive frames. After the removal of isolated pixels, remaining groups of connected pixels are classified as homogeneous moving objects. The algorithm finds the contour center for these groups of pixels and draws a rectangle around them. Currently, the motion detection app requires a static background for proper operation.

3.6 ImageChip

The ImageChip is a seperate writer app that uses a different format to display images than the Writer app. The ImageChip Writer uses a grid layout that displays the images that it receives into the slotted grids. Once the grid is full, the oldest frame is deleted and the newest frame is put in. Another function that the ImageChip does is that it can output multiple frames if needed while the Writer is programmed to only retrieve one at a time. Currently, the ImageChip Writer only functions with the Face Recognition application since the input of the Writer differs from the outputs of the other apps.

.

4. Technical Design

4.1 Google Protocol Buffers

Google Protocol Buffer is a language and platform neutral automated mechanism for serializing structured data. It is compatible with multiple languages such as C++, Java, and Python. To utilize Google protocol Buffer, a proto file must first be created (Figure 3). This protobuf file contains messages that are similar to C++ structs in that they contain declared variables of primitive data types or of other message types. These variables are either optional or required to be set in the source code of the accessing application. Once a proto file is made it needs to be compiled with a protobuf compiler. This action then creates a Java class that contains functions and APIs useful to any program that wishes to access them. It can then be utilized in data streams. The format of a Google Protocol Buffer can be seen in Figure 3.

```
syntax = "proto2";

package Protobuff;

message byteMap {
    required bytes image = 1;
    optional int32 height = 2;
    optional int32 width = 3;
}
```

Figure 3: Message from proto file

Our applications use Google Protocol Buffers to serialize the frames, and other information about the frames, into one single message that can be sent between applications.

Each frame that is processed is converted to the “bytes” type (essentially a bytestring representation of the frame) and is then stored in the message byteMap. This message can also be used to hold information such as the height and width of the frame, which may be required by an application to turn the bytestring back into a frame. Sending the height and width of the frames prevents any issues that can come with hard coding in the height and width of the frames into the code. The message is then serialized to be retrieved and unpacked by other apps.

4.2 MPCore

MPCore is a “highly dynamic and configurable architecture that permits creating, managing, and running domain agnostic data processing workflows”. It allows a user to start, stop, and configure applications that are registered with the client via a CSV, which contains information pertaining to the location of the executable or JAR, the arguments that need to be passed to the app, and other pertinent app data. MPCore has a graphical interface that allows for easy use (Figure 4).

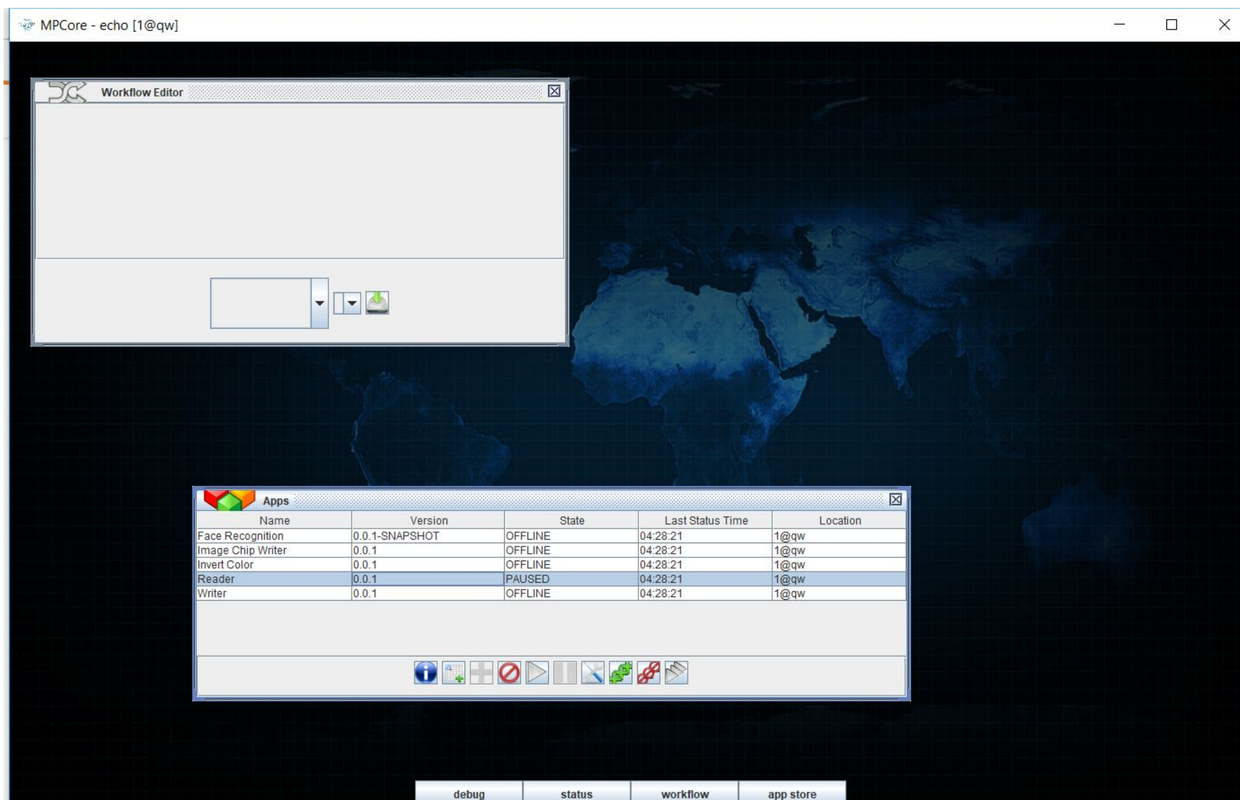

Figure 4: MPCore GUI showing app hub

Once an app's CSV is in the MPCore directory, it shows up in the Apps tab and can be interacted with (bottom center tab in Figure 4). The Workflow Editor (top left Figure 4, Figure 5) specifies the order that the apps run in, as well as the protobuf message type that is sent between the apps. Once in the Apps tab, applications can be added to the Workflow Editor based on the protobuf types they send and receive. The first app in the workflow can then be run and the others that are linked to it will follow suit.

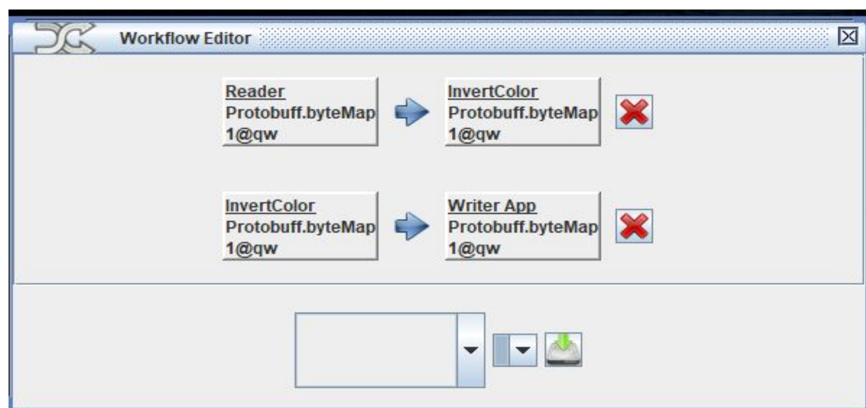

Figure 5: Workflow Editor showing flow of protobuf data through apps

4.3 Maven

Apache Maven is a build tool usually used for Java projects. A user is able to specify how a project should be built and what files it is dependent on through a Project Object Model or POM file and is usually packaged as a JAR or a WAR file depending on the user specification. The dependency management feature is especially useful when packaging up a project. Dependency management allows for a user to identify artifacts such as libraries that it needs. These artifacts also contain a POM file that tells Maven its dependency and packaging needs. This allows Maven to install all cascading dependencies in the way their developers intended without the user being required to track all the dependencies themselves saving developers lots of time when packaging an app to be used in a live or test environment.

At times the dependency management can have some drawbacks if others set up their dependencies poorly. This happened to our group when using the JavaCV libraries. The original developer had added every library that could be used with JavaCV instead of breaking up his artifacts to make them more modular. This resulted in a 500MB JAR file due to a large number of libraries in which a majority of those would not be used in our project. This resulted in having to tell Maven to exclude a large number of files to prevent it from downloading excess files.

For MPCore, we made liberal use of the Maven assembly plugin in particular. This can been seen below in Figure 6. This plugin allows for the packaging of all dependencies, documentation, and other files into a single executable JAR file. This was particularly helpful for our use as it minimized issues with build paths and not being able to find the specified dependencies at runtime.

```
<plugin>
    <artifactId>maven-assembly-plugin</artifactId> <!-- plugin id -->
    <configuration>
        <archive>
            <manifest>
                <mainClass>test.Reader</mainClass> <!-- location of main class
                necessary for a executable -->
            </manifest>
        </archive>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef> <!-- creates
            a JAR packaged with all necessary dependencies -->
        </descriptorRefs>
    </configuration>
    <executions>
        <execution>
            <id>make-assembly</id>
            <phase>package</phase> <!-- binds to package phase -->
            <goals>
                <goal>single</goal> <!-- tells Maven to use the descriptorRef -->
            </goals>
        </execution>
    </executions>
</plugin>
```

Figure 6: Maven Assembly Plugin Commented

5. Decisions

5.1 Coding Language

Because the project is required to utilize Google Protocol Buffer and MPCore APIs, the languages that were most suited for the role were Java and C++. We chose to program in Java since we were more familiar with Java, and we also wanted to learn to use Maven which is mainly used in Java environments. Throughout the project, we continued to use Java to code our project and did not attempt to use other languages.

5.2 Computer Vision Library

Computer vision is a field that deals with processing digital images and videos and automating what a human can do with the processed data. Our project revolves heavily around doing just this, so we knew we had to utilize Java computer vision libraries. We chose to use OpenCV due to the abundance of tutorials that were available to us and how well developed the library was to use. We also enjoyed the built-in methods that handled many complex algorithms in an efficient manner. Later, though, we had difficulty getting this to work without installation of .so files for Linux with CMake and the packaging of the ffmpeg library which OpenCV was dependent on. This was a particularly large issue as the generated .so file from CMake seems to be different for every computer we installed it on. As a result, an attempt was made to use JavaCV instead as we were able to package that into JAR files quite well without prior installation when we wanted to run on linux. This led to the development of the Face Recognition app that was made using JavaCV as a trial. Unfortunately, JavaCV was very difficult to use and no longer received good developer support. Therefore, no other apps were converted to JavaCV and we decided to require an installation of OpenCV prior to running the app.

5.3 Other Decisions

- We used the org.apache.commons.io library to unpack our libraries from the JARinto a temporary directory at run time as we wanted our JAR files to be a standalone executable without the need for the user to place the libraries in a specific location or specify a path to those libraries
- Our applications implements image processing instead of video processing since Google Protocol Buffer was only able store frames of a video instead of the video itself. It processes the video into frames and uses Google Protocol Buffer to serialize the frames to be read by another application.
- We decided to create a Google Protocol Buffer message that holds a bytestring. The bytestring is converted to a byte array that is then converted to a BufferedImage. The image can be modified and is then converted back to a byte array and to a bytestring to be serialized using Google Protocol Buffer.
- Instead of compiling processed frames into a video and writing it to a disk, we decided to sequentially display them through one of our applications to avoid memory issues that come with a long video.

- We got our processing algorithms from other users on github who had left tutorials on how to use computer vision libraries as the algorithms themselves were not the focus of the project. It was important to get processed video to display at 24 FPS.

6. Results

6.1 Features Implemented

Six applications were implemented in the final product. These include:

- Reader
- Writer
- ImageChip Writer
- Image/Video Processing Applications
    - Face Recognition
    - InvertColor
    - Motion Tracking

All of these work in MPCore in both Windows and Linux and is able to process video in real time. See section 3 for details on the applications.

6.2 Features Not Implemented

The applications were dependent on some outside sources and cannot be run by themselves. Therefore, it is necessary to have some dependencies already installed on the computer before being able to run the apps properly. With additional time, we would design our JAR file so it would unpack all libraries it is dependent on at runtime so that this installation step would be unnecessary or some other way to package these external sources more efficiently.

6.3 Performance Testing Results

Currently, the software runs in MPCore on both Windows and Linux performing image processing on a video at a very slow frame rate (approx. 5-7 fps). Additional work will have to be done to get this to run at a higher fps. Currently, we have not identified the cause of the poor performance. Since we were testing the apps in VirtualBox with limited RAM and CPU power, it might be the case that Virtual Box's resources were just not enough to handle the applications and, as a result, VB is being forced to run garbage collection more often. MPCore was designed to be used on server grade architecture and with each app being ran on a seperate computer. This too may be tied to the cause of the slow down.

6.4 Future Work

- The apps should in the future be able to run properly without any outside dependencies.

- The image processing apps (excluding the Reader and Writer) can be modified with config files that can switch between the image processing algorithms. This allows the apps to be condensed into one app.

6.5 Lessons Learned

- Maven is a powerful tool for packaging dependencies but can take some time to figure out
- Protobuf is a powerful tool for exchanging data between programs that aren't linked at runtime
- OpenCV is probably better used in C++ than in Java
- When incorporating new methods to the code from a 3rd party library, looking up the documentation is very helpful. Even if the code does not produce compilation errors, you still might be missing some arguments in function calls to get them running properly.
- Having everyone working in the same room and communicating what we are doing increases productivity

7. Appendices

7.1 Installation Instructions

Assumes MPCore is already installed.

Our apps require specific library files from OpenCV to run. We package these files into a JAR and then load them during runtime. All files that require packaging/loading during runtime should be placed in:

(project_base_directory)/src/main/resources

OpenCV provides pre-built windows packages that contain two .dll files necessary for our apps. They are opencv_ffmpeg341_64.dll & opencv_java341.dll. Loading these two files allow us to run apps on any windows machine. For linux platforms (each), you will have to build OpenCV from source code that will generate libopencv_java341.so file. You can then place that file in the resources folder (for all apps except the Writer) mentioned above and modify the code to access it. (Unless there is a better way to do it)

That should allow you to run the apps.

So, to build OpenCV on linux:

1. Install Java Development Kit. Configure JAVA_HOME / PATH environment variables.

- Note: Verify java -version & javac -version commands are working.

2. Install Apache Ant: https://ant.apache.org/manual/install.html

3. Install CMake & CMake Gui:

- "sudo apt-get install cmake" (Mint)
- "sudo yum install cmake" (CentOS)
- "sudo apt-get install cmake-gui" (Mint)
- "sudo yum install cmake-gui" (CentOS)

4. Install additional tools

- (CentOS)
- yum groupinstall "Development Tools" -y
- yum install gcc gtk2-devel numpy pkconfig -y
- (Mint)
- sudo apt-get install build-essential
- sudo apt-get install git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev

5. Build OpenCV:

You can refer to the following instructions: Remember to use version 3.4.1 when downloading the source pack.

- (CentOS, Stop at step 4) https://www.vultr.com/docs/how-to-install-opencv-on-centos-7
- (Mint, Stop at Java sample with Ant Section) https://docs.opencv.org/3.4/d9/d52/tutorial_java_dev_intro.html

The summary of the steps are outlined below:

1. Download OpenCV 3.4.1 source pack https://github.com/opencv/opencv/archive/3.4.1.zip
2. Unzip the file
3. Open the terminal:
    a. cd into the unzipped directory (opencv-3.4.1)
    b. mkdir build
    c. cd build
    d. cmake-gui
4. In "Where is the source code" specify the path to unzipped opencv pack (/opencv-3.4.1)
5. In "Where to build the binaries" specify the path to build directory you made (/opencv-3.4.1/build)
6. Press Configure. Use the default compilers for Unix MakeFiles
7. Unselect BUILD_SHARED_LIBRARY
8. Press Configure twice
9. Examine the output of Cmake and verify "java" is listed in the "To be built" line under "OpenCV modules"
    a. If not listed, verify JDK and Ant installation

10. Verify FFMPEG is set to YES
    a. If set to NO, ensure WITH_FFMPEG is selected before pressing configure again
11. If all is set correctly, press generate. Close the cmake-gui after generation is finished.

12. In the terminal (still in build directory):
    a. sudo make
    b. sudo make install

At this point you should have the libopencv_java341.so file generated in "opencv-3.4.1/build/lib" directory. You can now put this file in the (project_base_directory)/src/main/resources directory for every app (except Writer).

Modify 2-3 lines of code in the MpcStandalone.java (see lines 33-81 for similar examples) to add that file for packaging and loading during runtime. It is important to delete other function calls that try to load .so files that were not generated through the steps outlined above. Reinstall the app using "mvn install" command. You should be able to run the app in the current system now.

- Note: Before running all of the MPCore compatible apps, you can try running the standalone java app (StandTest)

You can run StandTest app from the terminal with a "java -jar Appname.jar vid.mp4" command (given that vid.mp4 is in the same directory as the JAR file). You will still need to put your libopencv_java341.so generated file into src/main/resources folder, modify code and reinstall the app.

If the app does not run, try installing ffmpeg and run the app again:

- (CentOS) https://www.vultr.com/docs/how-to-install-ffmpeg-on-centos
- (Mint) https://www.ostechnix.com/install-ffmpeg-linux/

7.2 Code Sources

Face Detection source:

From the JavaCV github Demo made by the JavaCV development team

https://github.com/bytedeco/javacv

Motion Detection:

From a Spanish tutorial found on Wordpress (Unknown author)

https://ratiler.wordpress.com/2014/09/08/detection-de-mouvement-avec-javacv/