MPD SDK and Sample Code

1 Overview

This package contains the MPD transport SDK (in folder `mpd/`), a sample program (in folder `demo/`), scripts for local test environment setup (in folder `mininet/`), and log analysis tools (in folder `tools/`). This package can only be used for participating the 2023 MMSys MPD grand challenge and related research activities. Any commercial use of this package is strictly prohibited.

You are encouraged to read the challenge introduction web page to get an overview of how MPD system works before digging into the code and this document. Feel free to send any questions to wanghaiping.paloma@bytedance.com if you have trouble with any steps below.

The MPD SDK included in this package was specifically customized for this grand challenge event only. It differs significantly from the commercial version used in ByteDance services. Please understand the code here has **NOT** been fully tested to meet the quality bar of ByteDance products. Although we are trying our best to deliver high quality code, you may still encounter bugs or even crashes, in which cases please let us know via emails so that we can help identify the problems and improve the code robustness.

2 Local Test Environment

For local test, you need to install mininet first. Please refer to the instructions listed on this page (we also recommend option 1).

With an Ubuntu VM with mininet set up, you can copy the directory `mininet/` to VM and run the script to start the test. The folder should look like this:

```
./bin/servertest
./config/upnode_mn0.json
./config/upnode_mn1.json
./config/downnode_mn.json
./twohosts_twoswitches.py
./MPDtest
```

Please be very careful when you copy files to the VM. Make sure the data node program `./bin/servertest` maintains its execution permission. Otherwise you

will not be able to run any test. Note that `MPDtest` is not included in this package. You need to build on your own platform first following the instructions in the next section and copy the binary file here. To start the test, run the following command:

```
$ sudo python twohosts_twoswitches.py
```

The script `twohosts_twoswitches.py` automatically sets up the network topology and deploys the data node instances. Then it swtiches to the command line and you need to manually start `MPDtest` using the following command:

```
mininet> client {absolute_path}/MPDtest {absolute_path}/downnode_mn.json
```

The provided script only sets up a very basic topology and all data nodes are deployed in the same LAN. You are encouraged to create more complex topologies, add various network constraints to test your algorithm performance after you get familiar with basic concepts. In the case you modify the network topology, you also need to modify the configuration files. One configuration file is required for each data node deployed. It simulates the data node assignment service in the real MPD system that dynamically searches for the best available data nodes upon user's video streaming requests. You need to change IP address in the configure file or add new configure files for additional data nodes following the examples of existing sample configuration files if you modify the mininet topology.

In normal cases, after the downloading program exits, you can simply exit the mininet command line and stop all other network components.

```
mininet> exit
```

However, if your `MPDtest` crashes, it may also crash the mininet as well. In this case, you may need to manually clean the network topology and data node instances using the following commands:

```
$ sudo mn -c
$ sudo sudo pgrep -a servertest
$ sudo pkill -f <pid output by previous command>
```

3 Compile and Run the Demo

Please use Ubuntu 20.04 to compile and run the demo program. We are not able to provide support to the compilation issues on other platforms for this challenge.

Please install the depending packages first with the following commands:

```
$ sudo apt update
$ sudo apt install libboost-all-dev openssl libspdlog-dev
```

We have provided `CMakeList.txt` to compile and build the demo program. You can run the following command in the package home folder:

```
$ cmake .
$ make
```

to generate the Makefile and then build the program. You are expected to copy the generated executable file to the mininet VM you have set up in the previous step (if you choose to compile in a different place) and put it in the same folder with the mininet script.

Use the mininet script to start your program with other test components as described above.

Each run stops after the whole video file has been downloaded or any error happens. You can find the detailed information in the program log.

4 Trace Analysis

Running the demo program generates a trace file `MPDTrace.txt` and it contains the information to evaluate the transport performance. We provide the trace analysis tool in the `tools/` folder, which is a python script `get_score.py`, to help you with the analysis. To run the script, you need to first install the required libraries:

```
$ pip3 install -r requirements.txt
```

Run the python script:

```
$ python3 get_score.py <path_to_trace_file>
```

You need to provide the correct path to the `MPDTrace.txt` file generated after running the demo program. Note that the demo program may overwrite the existing trace file. You are expected to see the script output as follows:

```
$ python3 get_score.py MPDTrace.txt

File size: 10 MB, video duration: 80.0 s, download duration: 19.430
657 s, total data downloaded: 20.0 MB, total data downloaded (no du
plicate): 10.0 MB

Video duration: 10 s, amount of data required: 1292.7802734375 KB,
download duration: 2.753581 s, amount of pieces downloaded: 1293
Video duration: 20 s, amount of data required: 2572.7607421875 KB,
download duration: 5.358802 s, amount of pieces downloaded: 2573
Video duration: 30 s, amount of data required: 3852.7412109375 KB,
download duration: 7.539469 s, amount of pieces downloaded: 3853
Video duration: 40 s, amount of data required: 5132.7216796875 KB,
download duration: 10.210041 s, amount of pieces downloaded: 5133
Video duration: 50 s, amount of data required: 6412.7021484375 KB,
download duration: 12.370609 s, amount of pieces downloaded: 6413
Video duration: 60 s, amount of data required: 7692.6826171875 KB,
```

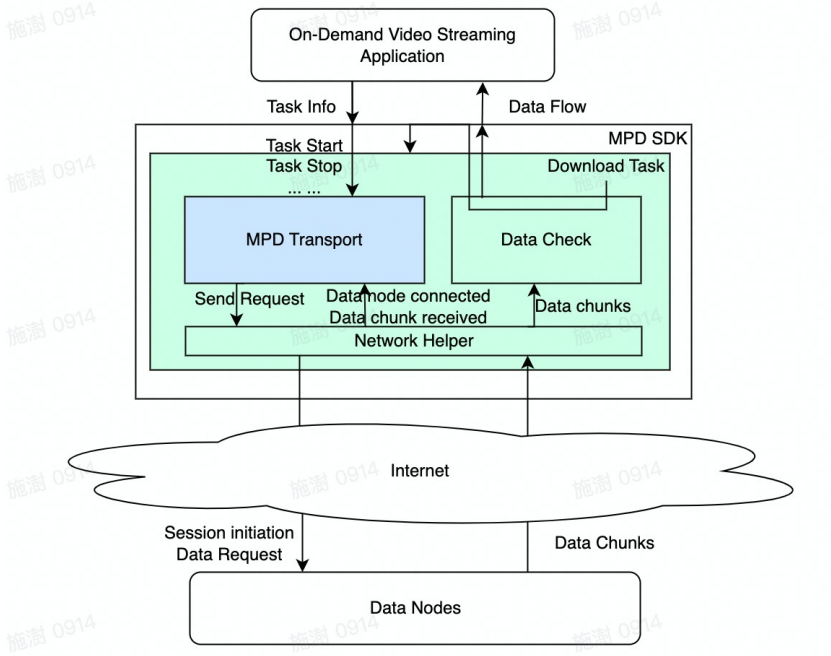

Figure 1: MPD SDK Framework

```
download duration: 15.042011 s, amount of pieces downloaded: 7693
Video duration: 70 s, amount of data required: 8972.6630859375 KB,
download duration: 17.215413 s, amount of pieces downloaded: 8973
Video duration: 80 s, amount of data required: 10239.9990234375 KB,
download duration: 19.430657 s, amount of pieces downloaded: 10240

alpha: 1.5, belta: 0.0, Score: 351.3348347751 KBps
```

It lists basic information of the whole downloading process and calculates the score, which is defined on the challenge website. It should be noted that if the program fails to download the entire file within the video duration, the final score will be set to 0.

Other than the trace file, you can also record the screen print out logs to help you debug the program. You can also adjust log level in the source code to get more or less information. Refer to the documentation of `spdlog` for more details.

5 Understand the SDK

The Figure. 1 shows an illustration of how every module interacts with each other. The MPD SDK as a whole provides the capability to download video

content using MPD transport for on-demand video streaming applications. The application provides the task information, which includes the ID, length and bitrate of the video content (see the struct `TransportDownloadTaskInfo`) and the SDK returns the sequential data flow as downloading starts.

Inside the SDK, there are three major modules: Network helper, Data check, and MPD transport. Network helper module performs all network communications, including querying the list of available data nodes for a given downloading task, initiating the data connection with data nodes, sending data requests, and receiving data chunks. In our system, video files are divided into small chunks (also called **data pieces** in the source code and comments) of 1024 bytes. All data nodes simply respond to each data request by sending back the requested data chunks without any delivery guarantee or congestion control. The MPD SDK manages all the received data chunks in the data check module. This module restores the data chunk order and returns to the streaming application. We have implemented these two modules so that you can focus on the design and implementation of the key MPD transport module.

MPD transport interacts with other modules mainly through callback functions. We provide a virtual class `MPDTransportController` to define all callback interfaces. Upon starting each downloading task, the module is notified with `OnDownloadTaskStart()`. After the network helper module starts data node connection, MPD transport receives `OnSessionCreate()` for each successful connection set up with a single data node. Note that MPD transport cannot query the information of how many data nodes are available on the list or still being attempted to connect. It only gets the notification once the connection is set up. Once the data node session is created, MPD transport can start to download data chunks by calling `DoSendDataRequest()`, an interface provided by network helper through `MPDTransCtlHandler`, to send out data requests to the connected data nodes. MPD transport receives the notification `OnDataSent()` and `OnDataPiecesReceived()`, when the data request is successfully sent out and the data chunk is received, respectively. MPD transport should repeat sending data requests until all data chunks are received, which will trigger `OnDownloadTaskStop()`. You can find more details, including the definition of function parameters, in the comments added for each function defined in `mpd/download/transportcontroller/transportcontroller.hpp`.

MPD transport needs to call `DoRequestDatapiecesTask()`, an interface defined in `MPDTransCtlHander`, in order to get the actual chunk IDs the SDK needs to download. Upon the request, the SDK puts the requested number of chunk IDs in a vector container and returns to MPD transport using `OnPieceTaskAdding()` call back asynchronously. There are two reasons why we design the interface this way. First, in the actual MPD system, a downloading task may only need to download a small portion of the video file. Second, this method adds a safety measure to limit the maximum downloading speed. In this challenge, we have modified the implementation of this interface to disable the speed limitation. However, we still ask the participant to use this interface to obtain all chunks ID to download rather than calculating based on the file length to be compatible with the online deployment in the future test and

evaluation.

You can choose to call `DoRequestDatapiecesTask()` frequently and each time you only request a small number of chunk IDs, like the demo application. You can also request a large enough number (e.g., calculate the most possible chunk number based on the file length) in the beginning to get all chunk IDs. Please note that each request call does not guarantee a responding callback. The SDK may not respond at all if it does not have the piece information ready yet. You may want to repeat the request if you do not receive callback responses and you believe there are more data chunks to download.

The SDK also provides a timer for your convenience. You get a periodic callback `OnLossDetectionAlarm()` at a fixed time interval. You may use this timer to check possible data chunk loss or anything else. The default interval is 100 ms and you can modify the interval value in `TransportModuleSettings`.

6 Develop New Transport Algorithms

In order to develop a new transport algorithm for MDP transport, you need to create a new transport controller class that inherits `MPDTransportController`, and add logic to all the callback functions listed above. You also need to provide a `TransportControllerFactory` class to SDK, which will be used to create your controller at the proper time. If you want to pass additional parameters to your controller, you can use `TransPortControllerConfig`, a struct that will be passed to your `TransportFactory` class to create your controller. The last step is to register your algorithm with MDP SDK through `TransportModuleSettings`.

You can refer to the demo implementation we have included in the `demo/` folder to get a better idea of how to put all the pieces together.

Regarding the strategy, you need to consider the following issues to enhance the downloading performance:

- Analyze the performance of available data nodes
- Adjust the download workloads assigned to different data nodes
- Apply certain flow control/congestion control methods to avoid mass data loss
- Recover missing data chunks in time to maintain adequate downloading speed

For example, in the provided demo,

- Sort the data nodes based on their smoothed RTT
- Assign each data node a window, the size of which is dynamically adjusted based on the congestion control algorithm. We guarantee the number of in flight packets does not exceed the window size.

- Apply a reno-like congestion control algorithm for each data node connection
- Set an timeout to detect packet loss

Last but not the least, the network helper module uses Boost.Asio library to send and receive UDP packets. All the operations inside the transport controller will be called in a single thread (See ASIO library's website), the same thread runs Network IO handlers. Please do not post any synchronous IO operation or any time consuming task inside the functions, which will slow down the IO handle speed. These functions are assumed to return quickly. If you do have such heavy duty operations (in very rare case), consider creating a new thread.