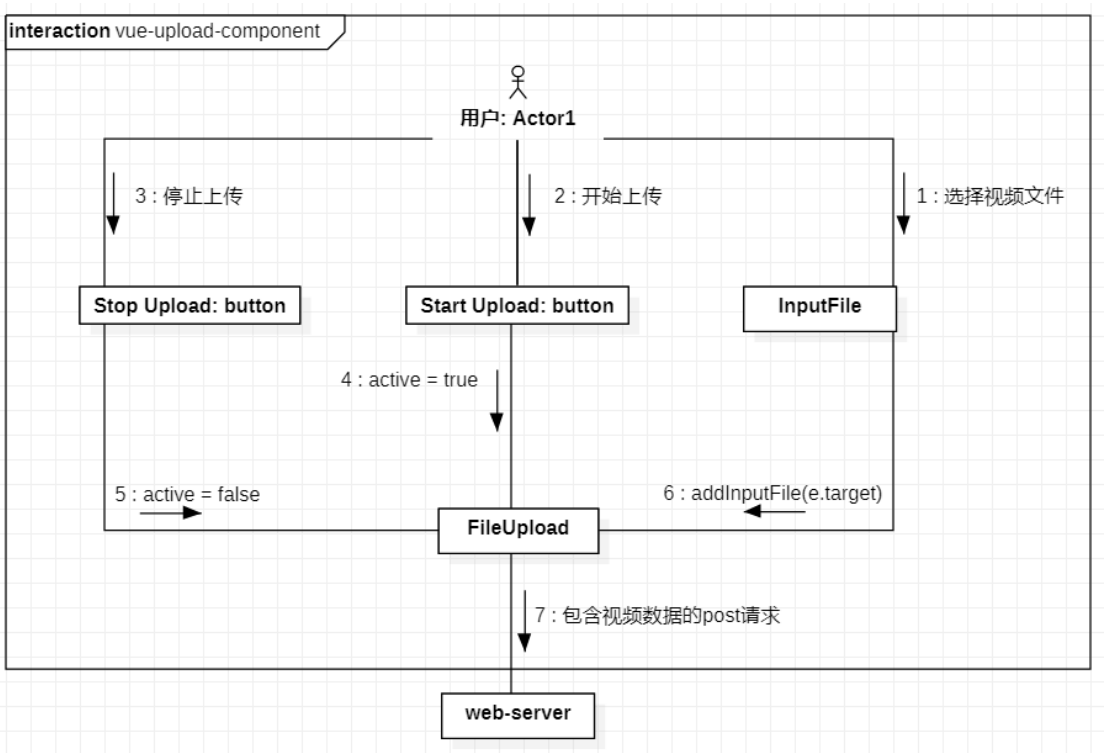
技术报告

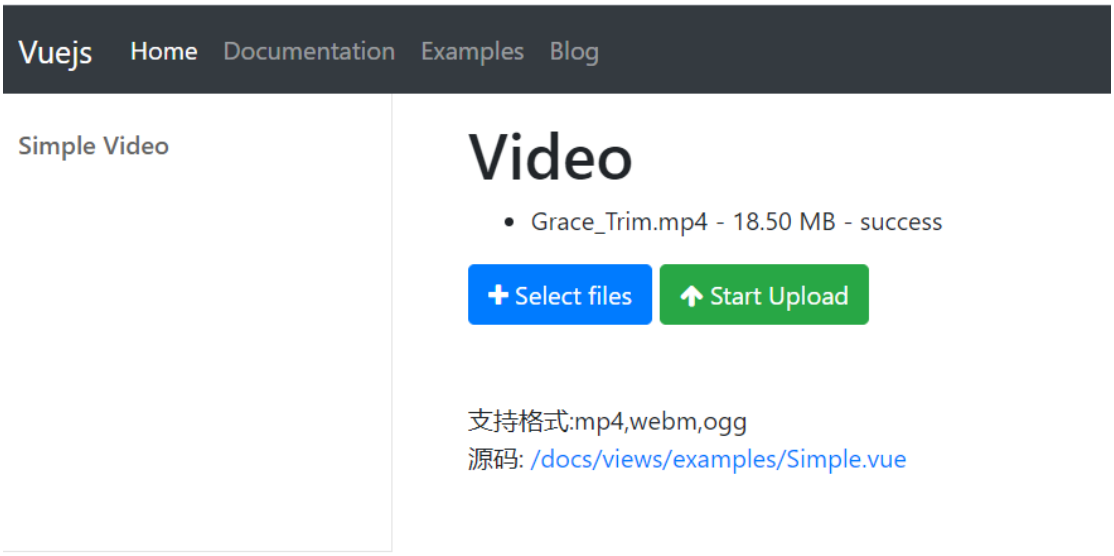
一、项目设计与分析

1. 功能性需求

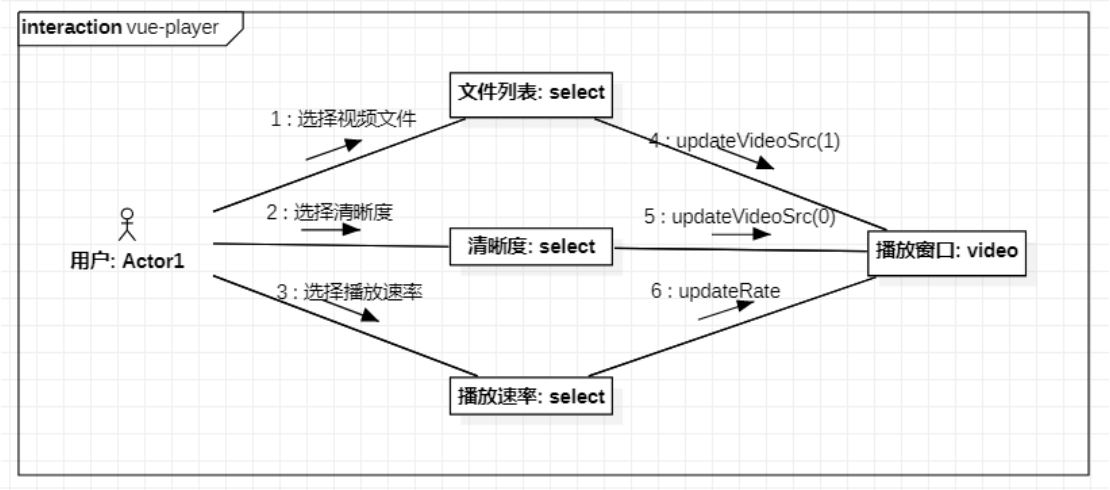
1.1 vue-upload-component——让用户上传视频文件的前端应用



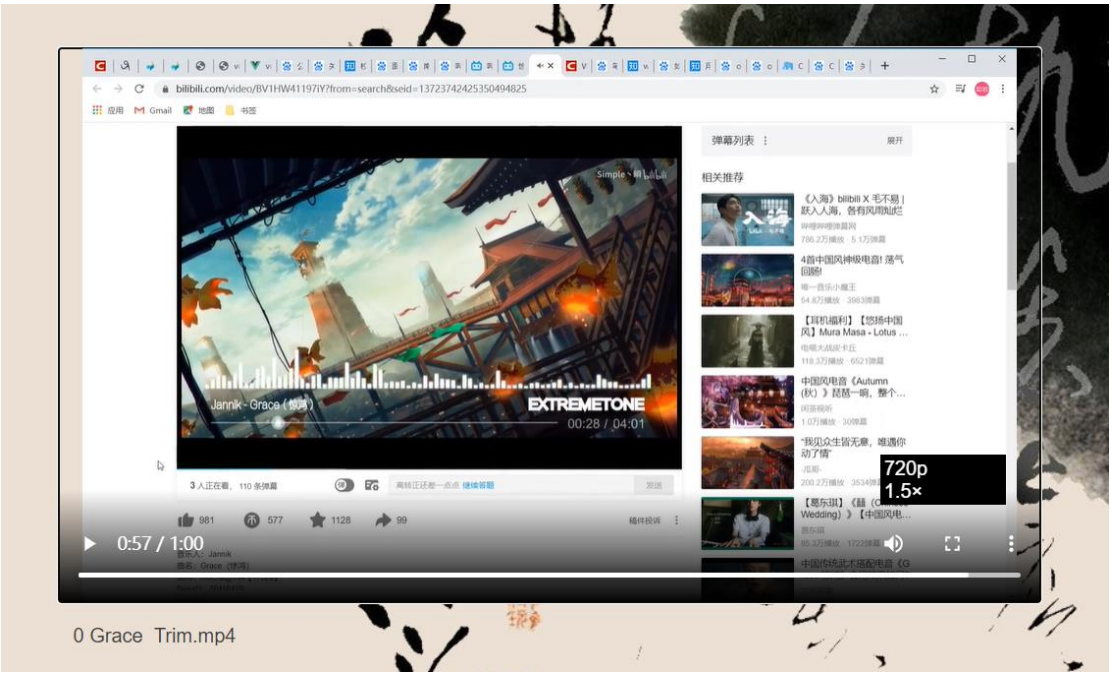
用户点击 Select files 选择文件，然后点击 Start Upload 上传视频文件。



1.2 vue-player——用户播放视频的前端应用



用户点击文件列表，选择文件，然后在播放窗口右键，会出现一个设置界面，该界面可以通过设置清晰度和放映速率来播放视频，再次在播放窗口右键，会关闭设置界面。

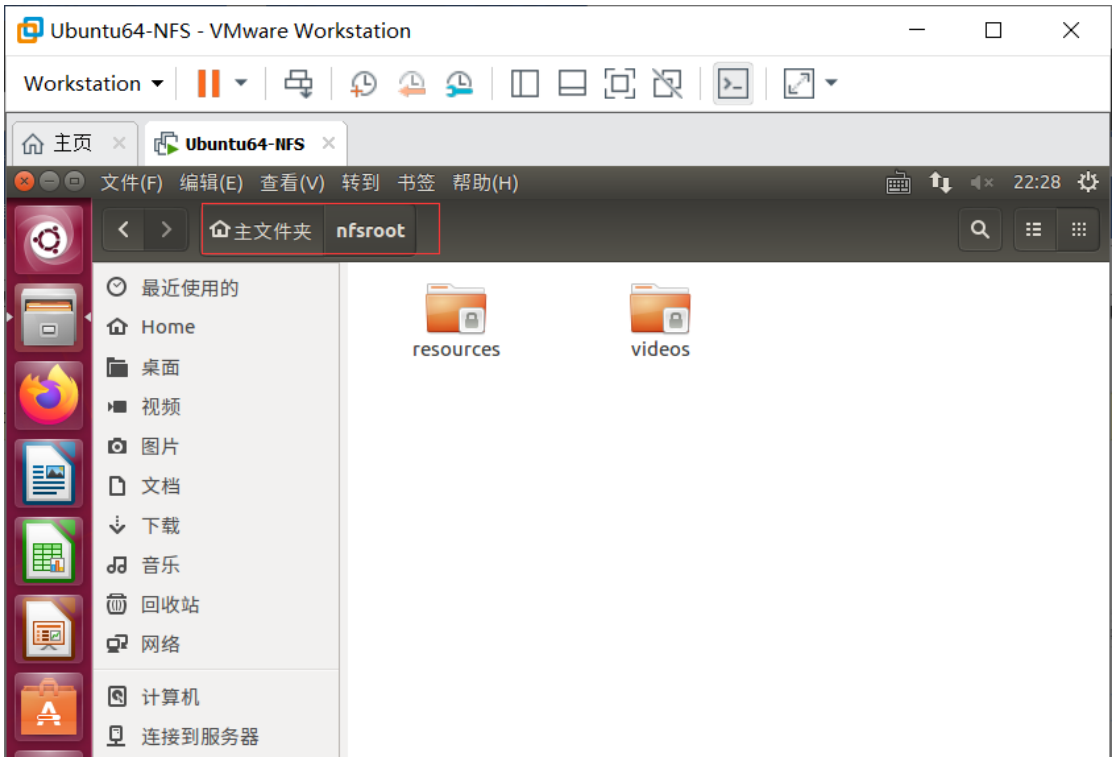


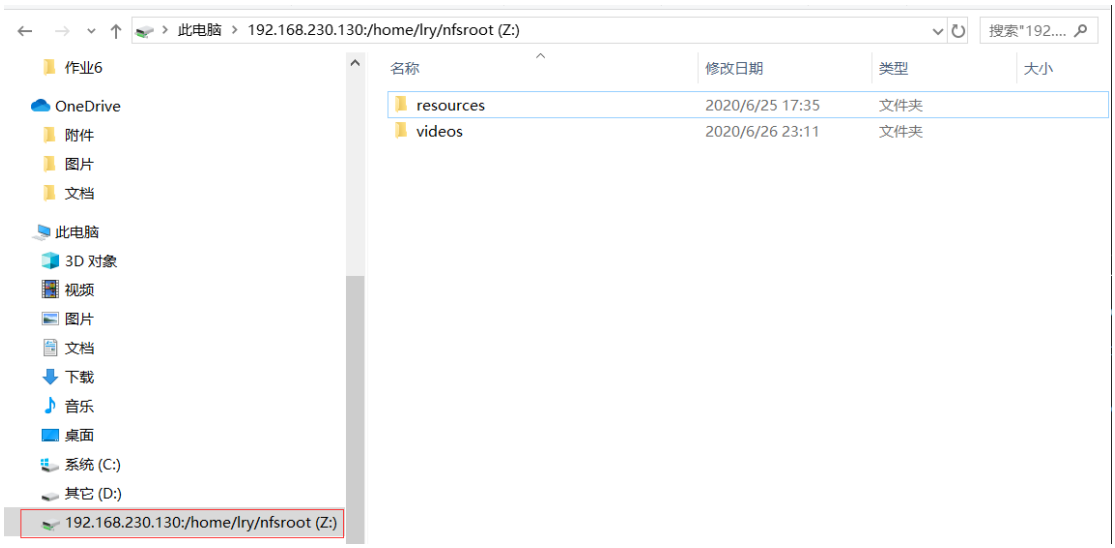
1.3 web-server——处理 http 请求的后端服务器

服务器 web-server 监听到“/upload”请求后，会接收前端发过来的视频，然后驱动一个转码系统来压缩视频；监听到“/infolist”请求后，会返回一个 json 数据，其中包含了服务器中的视频文件名列表和它们的清晰度；监听到“/files/视频文件名称”请求后，会返回该视频文件的数据流。

1.4 NFS——保存原始视频文件的独立的存储系统

通过 Ubuntu 来做 NFS 服务器，用 win10 系统作为 NFS 客户端





1.5 rabbitmq——保存任务的消息队列

RabbitMQ

Overview | Connections | Channels | Exchanges | Queues | Admin

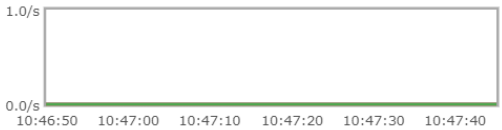
Overview

Totals

Queued messages (chart: last minute) (?)

Ready	0
Unacked	0
Total	0

Message rates (chart: last minute) (?)



Publish	0.00/s	Publish (Out)	0.00/s
Confirm	0.00/s	Deliver	0.00/s
Publish (In)	0.00/s	Redelivered	0.00/s

Global counts (?)

Connections: 2 | Channels: 2 | Exchanges: 8 | Queues: 1 | Consumers: 1

Overview | Connections | Channels | Exchanges | Queues | Admin

Connections

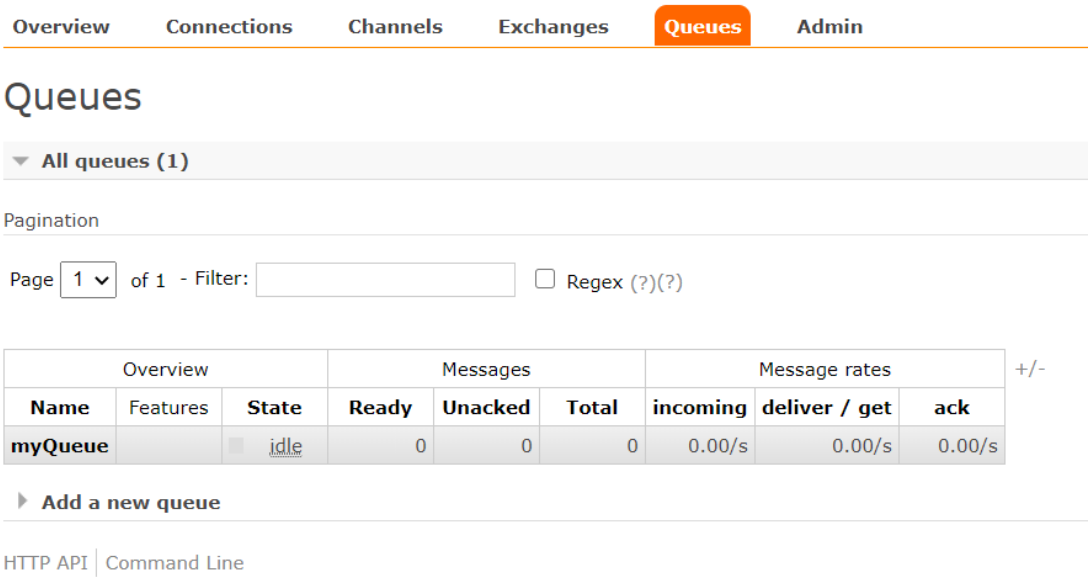
All connections (2)

Pagination

Page 1 of 1 - Filter: Regex (?)(?)

Overview			Details			Network		+/-
Name	**User name**	**State**	**SSL / TLS**	**Protocol**	**Channels**	**From client**	**To client**	
172.17.0.1:46818 rabbitConnectionFactory#7ae492a4:0	guest	running	○	AMQP 0-9-1	1	0B/s	0B/s	
192.168.99.1:13210 rabbitConnectionFactory#4cbab070:0	guest	running	○	AMQP 0-9-1	1	0B/s	0B/s	

HTTP API | Command Line



1.6 amqp-compress——独立的编码子系统

该编码子系统监听 myQueue 队列，获得一个消息后启动压缩视频的线程 CompressThread，然后将该线程插入 threads 中，最后确认消息，若 threads 中的线程数目大于或等于最大线程数 MAX_TASK_NUM，就让该消息等待，直到 threads 中的线程数目小于 MAX_TASK_NUM。压缩结束后将产生的不同清晰度的视频存放到 NFS 网络文件系统中。CompressThread 会调用 win10 下的 HandBrakeCLI，如果您使用的 MacOS，请将其替换成 MacOs 下的 HandBrakeCLI

```java
// 配置监听
@Bean
public SimpleMessageListenerContainer myQueueLister(ConnectionFactory connectionFactory) {
    System.out.println("simple message listener********");
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer(connectionFactory);
    // 设置监听的队列
    container.setQueues(new Queue(MY_QUEUE_NAME, NON_DURABLE));

    // 指定要创建的并发使用者数。
    container.setConcurrentConsumers(1);
    // 设置消费者数量的上限
    container.setMaxConcurrentConsumers(1);

    // 设置是否自动签收消费 为保证消费被成功消费，建议手工签收
    container.setAcknowledgeMode(AcknowledgeMode.MANUAL);

    container.setMessageListener(new ChannelAwareMessageListener() {
        @Override
        public void onMessage(Message message, Channel channel) throws Exception {
            // channel
            channel.basicQos(MAX_MSG_NUM, false);
            // 可以在这个地方得到消息额外属性
            MessageProperties properties = message.getMessageProperties();
            String str = new String(message.getBody());
            System.out.println(MY_QUEUE_NAME + "收到消息:" + str);
```

```java
            while(threads.size()>=MAX_TASK_NUM){
                for(int i=threads.size()-1;i>=0;i--){
                    if(!threads.get(i).isAlive()){
                        System.out.println("remove thread");
                        threads.remove(i);
                    }
                }
                Thread.sleep(100);
            }
            Thread thr=new CompressThread(message, channel);
            thr.start();
            threads.add(thr);
            /*
             * DeliveryTag 是一个单调递增的整数 第二个参数 代表是否一次签收多条，如果设置为 true,则所有
             * DeliveryTag 的消息都会被签收
             */
            channel.basicAck(properties.getDeliveryTag(), false);
            System.out.println("ack**********");
        }
    });
    return container;
}
```

2. 非功能性需求

2.1 为支持用户并发访问，web-server 应用可通过 docker 扩展。我在系统中自行加入负载均衡器（haproxy）和监控组件（prometheus）进行性能指标监控，在指标低于某个阈值时触发警报，给出提示信息让管理人员手工进行子系统扩展；

haproxy.cfg：

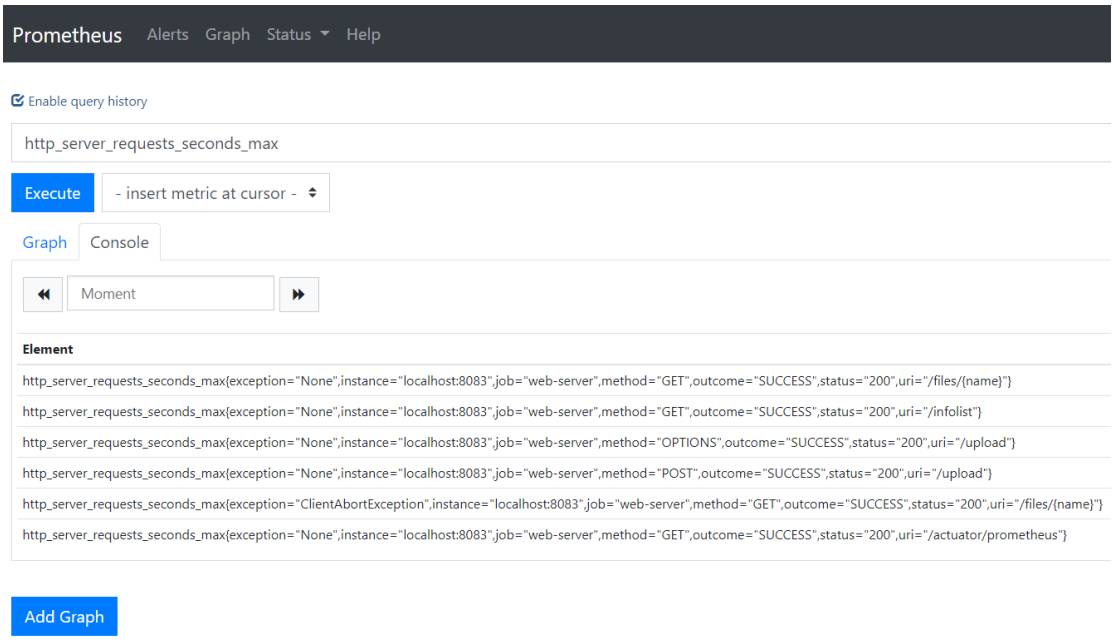
```
global
    log 127.0.0.1 local2
    maxconn 3200
    pidfile D:\Program\haproxy-windows\haproxy.pid
defaults
    log global
    mode tcp
    timeout connect 5000ms
    timeout client 10000ms
    timeout server 50000ms
    timeout http-request 20000ms

frontend http_request
    bind *:8083
    default_backend servers
backend servers
    balance roundrobin
    server server0 192.168.99.100:18080 check inter 1000
    server server1 192.168.99.100:18081 check inter 1000
    # server server2 192.168.99.100:18082 check inter 1000
```

```
listen admin_stats
    mode http #http的7层模式
    log 127.0.0.1 local0 err #错误日志记录
    bind *:1080
    option httplog
    stats refresh 30s
    stats uri /stats
    stats auth admin:admin
    stats admin if TRUE
    stats hide-version
```

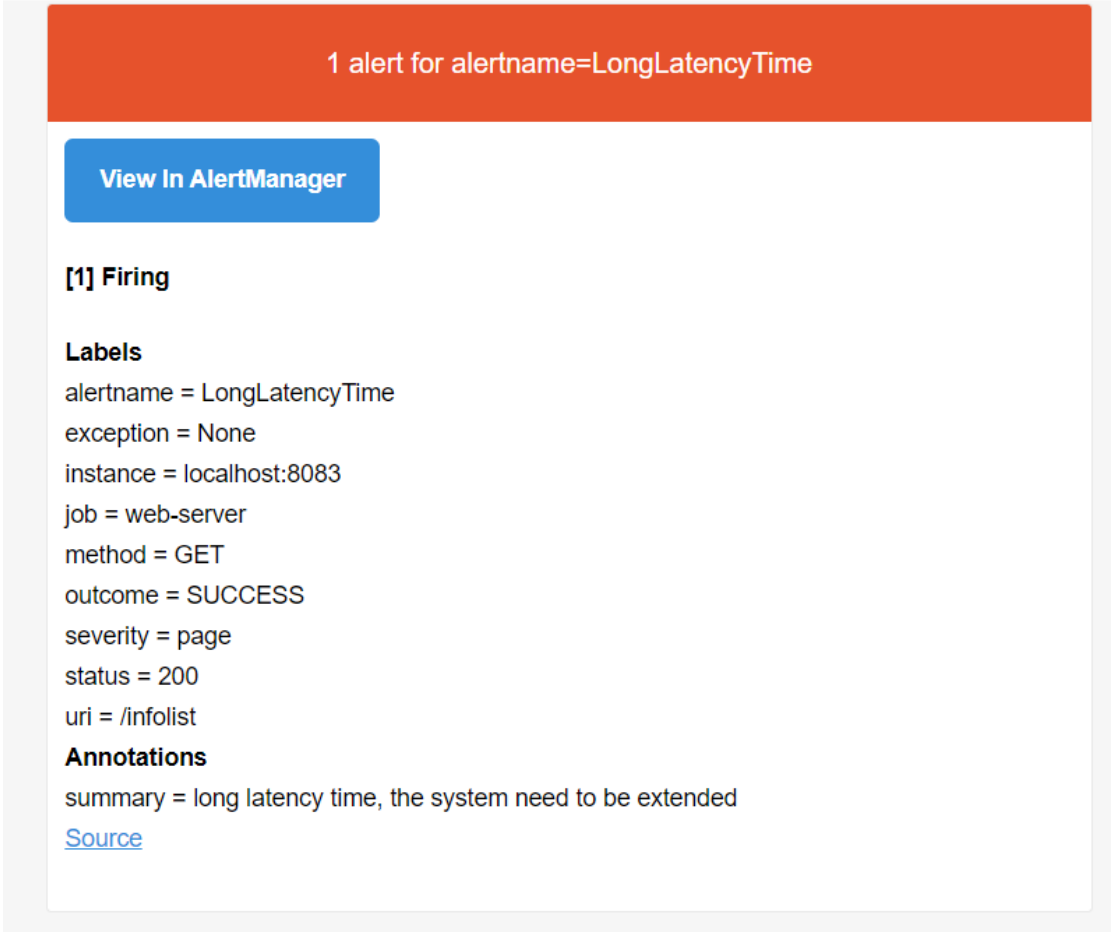
prometheus.yml：

```
# A scrape configuration containing exactly one endpoint to scrape:
# Here it's Prometheus itself.
scrape_configs:
  # The job name is added as a label `job=<job_name>` to any timeseries
  - job_name: 'prometheus'

    # metrics_path defaults to '/metrics'
    # scheme defaults to 'http'.

    static_configs:
    - targets: ['localhost:9090']

  - job_name: rest-server
    metrics_path: /actuator/prometheus
    static_configs:
    - targets: ['localhost:8083']
```

prometheus 监控界面：



邮箱中收到的提醒：



2.2 为提高系统响应性，编码子系统应在有限条件下进行水平扩展。因编码任务是计算密集型任务，假设系统资源只够 N 个编码任务同时运行，因此上

述系统中设计了任务队列，用于缓冲待编码任务。如果当前编码任务数小于 N，且任务队列中存在等待任务，则直接启动新的编码进程处理任务，否则让任务处于等待。详情见 1.6 的 amqp-compress 分析。

3. 优点与缺点

3.1 优点

3.1.1

项目的各个部分相互独立，仅仅通过接口联系在一起，提高了内聚性，降低了耦合度。

3.1.2

项目的各个部分互不干扰，可以各自独立开发，便于将任务分配给不同部门来同时开发，提高了开发效率。

3.1.3

使用 NFS 网络文件系统有很多好处：

- 节省本地存储空间，将常用的数据存放在一台 NFS 服务器上且可以通过网络访问，那么本地终端将可以减少自身存储空间的使用。
- 用户不需要在网络中的每个机器上都建有 Home 目录,Home 目录可以放在 NFS 服务器上且可以在网络上被访问使用。
- 一些存储设备如软驱、CDROM 和 Zip(一种高储存密度的磁盘驱动器与磁盘)等都可以在网络上被别的机器使用。这可以减少整个网络上可移动介质设备的数量。

3.1.4

使用 rabbitmq 有很多好处：

- 同步变异步
  - 可以使用线程池解决，但是缺点很明显：要自己实现线程池，并且强耦合，大多数是使用消息队列来解决

- 低内聚高耦合：解耦，减少强依赖。
- 流量削峰
  - 通过消息队列设置请求最大值,超过阀值的抛弃或者转到错误界面
- rabbitmq 采用信道通信，不采用 tcp 直接通信。
  - tcp 的创建和销毁开销大，创建 3 次握手，销毁 4 四次分手。
  - 高峰时成千上万条的链接会造成资源的巨大浪费，而且操作系统每秒处理 tcp 的数量也是有数量限制的，必定造成性能瓶颈。
  - 一条线程一条信道，多条线程多条信道，公用一个 tcp 连接。一条 tcp 连接可以容纳无限条信道（硬盘容量足够的话），不会造成性能瓶颈。

3.2 缺点

3.2.1

项目划分的模块较多，不便于统筹管理，开发各个模块的小组之间需要良好的沟通与协调。

3.2.2

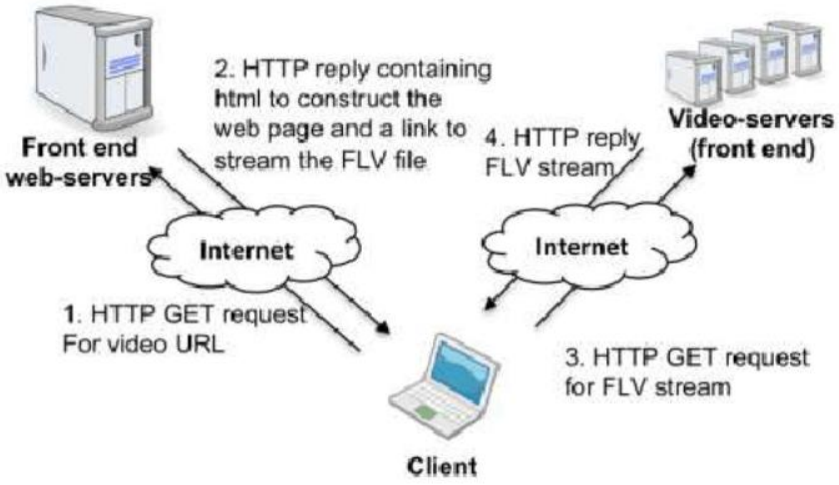
功能要求中给出的设计缺少一个处理前端 http 请求的服务器，如果不添加该服务器，上传视频的前端应用就需要添加一个发送 rabbitmq 消息的组件，导致前端逻辑体积越发庞大，越来越“臃肿”，同时存储系统也要添加处理上传视频请求和获取视频请求的服务端，导致存储系统功能不再专一。

二、YouTube 的系统设计

1. 架构图解

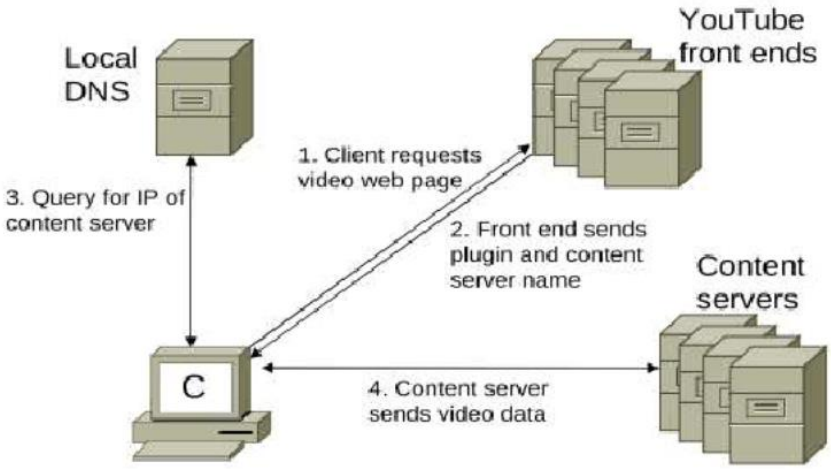
YouTube 被 Google 收购前的架构：

1. YouTube video delivery framework



YouTube 被 Google 收购后的架构：

1. High level sequence of steps to retrieve content.



2. 平台

Apache

Python

Linux(SuSe)

MySQL

psyco，一个动态的 Python 到 C 的编译器

lighttpd 代替 Apache 做视频查看

3. 状态

支持每天超过 1 亿的视频点击量

成立于 2005 年 2 月

于 2006 年 3 月达到每天 3 千万的视频点击量

于 2006 年 7 月达到每天 1 亿的视频点击量

2 个系统管理员，2 个伸缩性软件架构师

2 个软件开发工程师，2 个网络工程师，1 个 DBA

4. Web 服务器

4.1，NetScaler 用于负载均衡和静态内容缓存

4.2，使用 mod_fast_cgi 运行 Apache

4.3，使用一个 Python 应用服务器来处理请求的路由

4.4，应用服务器与多个数据库和其他信息源交互来获取数据和格式化 html 页面

4.5，一般可以通过添加更多的机器来在 Web 层提高伸缩性

4.6，Python 的 Web 层代码通常不是性能瓶颈，大部分时间阻塞在 RPC

4.7，Python 允许快速而灵活的开发和部署

4.8，通常每个页面服务少于 100 毫秒的时间

4.9，使用 psyco(一个类似于 JIT 编译器的动态的 Python 到 C 的编译器)来优

化内部循环

4.10，对于像加密等密集型 CPU 活动，使用 C 扩展

4.11，对于一些开销昂贵的块使用预先生成并缓存的 html

4.12，数据库里使用行级缓存

4.13，缓存完整的 Python 对象

4.14，有些数据被计算出来并发送给各个程序，所以这些值缓存在本地内存中。这是个使用不当的策略。应用服务器里最快的缓存将预先计算的值发送给所有服务器也花不了多少时间。只需弄一个代理来监听更改，预计算，然后发送。

5. 视频服务

5.1，花费包括带宽，硬件和能源消耗

5.2，每个视频由一个迷你集群来 host，每个视频被超过一台机器持有

5.3，使用一个集群意味着：

- -更多的硬盘来持有内容意味着更快的速度
- -failover。如果一台机器出故障了，另外的机器可以继续服务
- -在线备份

5.4，使用 lighttpd 作为 Web 服务器来提供视频服务：

- -Apache 开销太大
- -使用 epoll 来等待多个 fds
- -从单进程配置转变为多进程配置来处理更多的连接

5.5，大部分流行的内容移到 CDN（content delivery network）：

- -CDN 在多个地方备份内容，这样内容离用户更近的机会就会更高
- -CDN 机器经常内存不足，因为内容太流行以致很少有内容进出内存的

颠簸

5.6，不太流行的内容(每天 1-20 浏览次数)在许多 colo 站点使用 YouTube 服务器

- -长尾效应。一个视频可以有多个播放，但是许多视频正在播放。随机硬盘块被访问，在这种情况下缓存不会很好，所以花钱在更多的缓存上可能没太大意义。
- -调节 RAID 控制并注意其他低级问题
- -调节每台机器上的内存，不要太多也不要太少

6. 视频服务关键点

6.1，保持简单和廉价

6.2，保持简单网络路径，在内容和用户间不要有太多设备

6.3，使用常用硬件，昂贵的硬件很难找到帮助文档

6.4，使用简单而常见的工具，使用构建在 Linux 里或之上的大部分工具

6.5，很好的处理随机查找(SATA，tweaks)

7. 缩略图服务

7.1，做到高效令人惊奇的难

7.2，每个视频大概 4 张缩略图，所以缩略图比视频多很多

7.3，缩略图仅仅 host 在几个机器上

7.4，持有一些小东西所遇到的问题：

- -OS 级别的大量的硬盘查找和 inode 和页面缓存问题
- -单目录文件限制，特别是 Ext3，后来移到多分层的结构。内核 2.6 的最近改进可能让 Ext3 允许大目录，但在一个文件系统里存储大量文件不是

个好主意

- -每秒大量的请求，因为 Web 页面可能在页面上显示 60 个缩略图
- -在这种高负载下 Apache 表现的非常糟糕
- -在 Apache 前端使用 squid，这种方式工作了一段时间，但是由于负载继续增加而以失败告终。它让每秒 300 个请求变为 20 个
- -尝试使用 lighttpd 但是由于使用单线程它陷于困境。遇到多进程的问题，因为它们各自保持自己单独的缓存
- -如此多的图片以致一台新机器只能接管 24 小时
- -重启机器需要 6-10 小时来缓存

7.5，为了解决所有这些问题 YouTube 开始使用 Google 的 BigTable，一个分布式数据存储：
-避免小文件问题，因为它将文件收集到一起

-快，错误容忍

-更低的延迟，因为它使用分布式多级缓存，该缓存与多个不同 collocation 站点工作
-更多信息参考 Google Architecture，GoogleTalk Architecture 和 BigTable

8. 数据库

8.1，早期

- -使用 MySQL 来存储元数据，如用户，tags 和描述
- -使用一整个 10 硬盘的 RAID 10 来存储数据
- -依赖于信用卡所以 YouTube 租用硬件
- -YouTube 经过一个常见的革命：单服务器，然后单 master 和多 read

slaves，然后数据库分区，然后 sharding 方式

- -痛苦与备份延迟。master 数据库是多线程的并且运行在一个大机器上所以它可以处理许多工作，slaves 是单线程的并且通常运行在小一些的服务器上并且备份是异步的，所以 slaves 会远远落后于 master
- -更新引起缓存失效，硬盘的慢 I/O 导致慢备份
- -使用备份架构需要花费大量的 money 来获得增加的写性能
- -YouTube 的一个解决方案是通过把数据分成两个集群来将传输分出优先次序：一个视频查看池和一个一般的集群

8.2，后期

- -数据库分区
- -分成 shards，不同的用户指定到不同的 shards
- -扩散读写
- -更好的缓存位置意味着更少的 IO
- -导致硬件减少 30%
- -备份延迟降低到 0
- -现在可以任意提升数据库的伸缩性

9. 数据中心策略

9.1，依赖于信用卡，所以最初只能使用受管主机提供商

9.2，受管主机提供商不能提供伸缩性，不能控制硬件或使用良好的网络协议

9.3，YouTube 改为使用 colocation arrangement。现在 YouTube 可以自定义所有东西并且协定自己的契约

9.4，使用 5 到 6 个数据中心加 CDN

9.5，视频来自任意的数据中心，不是最近的匹配或其他什么。如果一个视频足够流行则移到 CDN

9.6，依赖于视频带宽而不是真正的延迟。可以来自任何 colo

9.7，图片延迟很严重，特别是当一个页面有 60 张图片时

9.8，使用 BigTable 将图片备份到不同的数据中心，代码查看谁是最近的