基于 UDP 实现多媒体即时通信机制

GS201603596TG034 项目需要设计一套高性能的即时通信框架，其中涉及到传输图片、视频、大块文字的需求。为了实现高性能的同时减少带宽占用，项目组参考了 QQ 通信协议以及 Google Protocol Buffers 的精简设计理念设计了一套基于 UDP 的多媒体即时通信机制，达到低带宽、高性能的效果。

QQ 通信协议

QQ 在即时通信应用中并没有采用 XMPP、HTTP 等上层的协议进行消息通信，主要是基于效率的考虑：XMPP、HTTP 等上层的传输协议为了保证易用性、通用性，包含了太多对应用无用的数据，从带宽、性能方面而言并不是特别优秀。QQ 通信的数据结构和 Protocol Buffers 的机制类型，通过对数据进行序列化反序列化直接进行传输。但是 QQ 的通信仍然是基于 TCP 的，而本项目的场景要求对流量使用有着变态的要求，因此项目组尝试基于 UDP 的方式底层机制来实现。

目标与问题

项目要求传输的数据至少包括如下数据类型：文字、语音、图片、视频、文档、其他等。

文字	语音	图片	视频片段	文档	表情	其他

由于 UDP 协议中每个包传输的数据大小有限并且 UDP 是不可靠的通信传输方式，会导致出现粘包和丢包的情况，设计不当的话其效率将比 TCP/HTTP 这类传统传输协议低效得多，因此在实现中既要处理好粘包的情况，也要处理好丢包的情况。

数据结构设计

在协议字段上，我们参考了 QQ 的 TCP 传输协议，将数据进行分片打包的方式进行流式处理。

序号	字段	类型	描述
1	tag	byte[2]	传输协议包头固定值 0xbebe，用于区分是否有效数据包、过滤部分恶意攻击

2	counter	int	数据包唯一 ID，单调递增以帮助接收端识别丢包
3	cmdType	byte	消息类型，不同的消息 ID 对应不同的数据内容（如文字、语音、图片、视频、文档等等）
4	sliceID	short int	分片序号，第一片数据序号要加上 0x7FFF 用于区别首包数据，分片序号采用降序递减的方式发送(避免接收方区分 EOF)；不分片传输时分片序号为 0xffff （节省一个 byte）。注意：本设计的最大传输文件大小为 0x7FFF*MTU(如 MTU 为 1500，则最大传输数据为 48MB)
5	sliceLength	short int	待传输数据的整体长度或分片数据长度，取决于是否是首片数据。首片数据时传整体长度是为了接收方预准备缓存以避免多次数据拷贝操作
6	body	byte[]	消息体具体内容。分片传输时首片数据的 body 提供配置信息，依次为：MTU 值、丢包处理、超时时间，对应字段为：short int, byte, short int

代码如下：

```
public abstract class AbstractPacket {

        /** 协议包头固定值 0xbebe，用于区分是否有效数据包，防止恶意攻击 **/

        protected byte[] tag;

        /**数据包唯一 ID，不断递增，帮助识别丢包 */

        protected int counter;

        /**消息类型，不同的消息 ID 对应不同的数据内容（如文字、语音、图片、视频、文档等）*/

        protected byte cmdType;      /**消息分片序号，第一片数据的序号要加上 0xff 用于区别首包数据，序号采用降序递减的方式发送(避免区分 EOF) */
```

```
        protected short int sliceID;        /**待传输数据的整体长度
或分片数据的长度 */
        protected short int sliceLength;
        /**消息体*/
        protected byte[] body;

}
```

以上数据结构是经过验证的较优结果，能够以有限的字段传递足够的信息（数据本身、丢包、超时、配置信息等）。

数据传输

数据传输有两种情形：

1、普通数据传输

要发送的数据较小时，不需要分片就可直接发送，这是最简单的一种情形。例如要传输一串文本（”QQ”）时，那么数据包的设置将为：tag=0xbebe, counter=<递增 ID>,cmdType=<文字消息 ID>，sliceID=0xffff, sliceLength=2, body=”QQ’。

2、大数据传输

如果要发送的数据很大，一次 UDP 发送无法发送完毕，那就需要对数据进行分片。例如要发送的一段大小为 10KB 的语音数据，那么

首片数据的设置将为：tag=0xbebe, counter=<递增 ID>,cmdType=<语音消息 ID>，sliceID=0xff+10KB/1024+1=0x10A, sliceLength=5, body=”MTU=1024,Relay=1,Timeout=120″，

第二片数据的设置为：tag=0xbebe, counter=<递增 ID+1>,cmdType=<语音消息 ID>，sliceID=10KB/1024=0xA, sliceLength=1024, body=”<语音消息前 1024 个字节>”，

最后一片数据的设置为：tag=0xbebe, counter=<递增 ID+11>,cmdType=<语音消息 ID>，sliceID=1, sliceLength=1024, body=”<语音消息最后 1024 个字节>”

丢包处理

由于 UDP 传输存在不可靠、乱序等问题，数据传输的接收方需要有效的识别丢包并反馈给发送方以请求重传。丢包处理有两种情况：第一种情况，针对不需要分片的数据，只要过了超时时间即可要求发送方重传。

第二种情况是针对分片的大数据，一种最简单的策略（Relay=1）是一旦发现分片数据丢失即要求所有分片数据重新传输，这种方式对接收方而言最简单，但是效率很低。实测发现其效率远低于使用 TCP 等传统的传输方式。另一种优化的策略（Relay=2）是只要求发送方重传丢失的分片数据，已收到的分片数据不影响，这种策略也是大多数丢包处理方案所采用的的方式。两种方式进行结合也是可以的，主要看分片数量的大小：如果分片数量很少，第一种策略效果没问题；如果分片数量很大，则应采用第二种。

除了以上两种情况，丢包处理还可以进一步进行优化，具体的实现方法可以参考 TCP 的 sliding window 的重传方式，但由于实现相对复杂，并且项目时间紧张而暂未实现。

在数据收取上，我们采用了 bitmap 的方式记录已收和未收的包，最大程度对空间进行压缩。

命令回复

数据传输的回复结果理论上可以复用数据发送的流程，但由于命令的回复内容都较短，一般不会出现需要分片的情景，只需要处理丢包的情景，因此可以针对数据发送的流程进行优化来实现命令回复。具体如下所示：

序号	字段	类型	描述
1	tag	byte[2]	传输协议包头固定值 0xbeef，用于区分是否有效数据包、过滤部分恶意攻击
2	counter	int	回复数据包唯一 ID，单调递增以帮助识别丢包
3	cmdType	byte	消息类型，不同的消息 ID 对应不同的数据内容（如文字、语音、图片、视频、文档等等）
4	result	short int	信息传输的返回码，具体可根据需要随便定义

5	body	byte[]	信息传输的成功反馈或错误解释

确认 MTU 的值

选好 MTU 的值对于传输效率的影响十分巨大，因此在协商过程中应该把 MTU 的值设为多少很有讲究。MTU 选得过大，则丢包率严重；MTU 选得过小，则传输次数太多，影响传输速度（因为重传机制存在）。在我们大量的测试验证中发现，MTU 在 500 字节左右是最合适的。如果网络环境较差，丢包率较高，可以适当减少 MTU 的值，但不要低于 200；如果网络环境较好、丢包率低，可以适当增大 MTU，但不要大于 1500。

方案效果

虽然我们花了一些时间另辟蹊径地在 UDP 的基础上设计了这个多媒体传输方案，期望得到相对更优的方案。项目上线后发现整体优化结果不是非常明显，甚至在一些情况下效率反而很低，可能是丢包重传机制没有优化的原因。在网络环境很差的场景中，本方案的效率比较低，但网络环境一般甚至良好的情况下其效果较好。