Dragonite Protocol (v2)

协议文档

简介

Dragonite 是一个基于 UDP 的快速可靠数据传输协议。UDP 自身是一种无连接，以报文为单位，提供数据的不可靠传递的协议。Dragonite 在 UDP 的基础上主要实现了数据拆分、丢失重传、按序到达、速率控制等用于进行可靠数据传输的特性。Dragonite 通过对 UDP 传输的封装能够提供类似 TCP 协议的功能与接口，但在协议设计、重传机制、速率控制、连接管理上与 TCP 有较大不同。Dragonite 的设计使用场景是在已知带宽上限、丢包率较高的不稳定网络环境中保持稳定与快速的传输性能。

规范与概念

协议中所有多字节数字均采用网络字节序（大端序）。

连接管理

为了在无连接的 UDP 协议中实现有状态的连接机制，连接接受方使用一个固定 UDP 端口用于监听接受连接，连接发起方的每个连接使用一个独立的 UDP 端口，接受方根据发起方 IP:PORT 的组合区分不同连接。在原先连接已经关闭后（显式关闭或超时等），同样的地址可复用于新的连接。

消息

Dragonite 协议传输数据的单位为"消息"(Message)，每个消息有着不同类型与结构，例

如携带数据，表示连接的发起/关闭，声明数据收到等。

消息分为两大类型，普通消息(Message)与可靠消息(Reliable Message)。普通消息不需要保证消息的成功到达，类似于 UDP 包的直接发送；可靠消息需要确保其按序、成功到达。

消息序号

消息序号(Sequence)是每个可靠消息包含的一个 4 字节整数字段。可靠传输机制需要以此保证消息的按序可靠传输。每个连接建立后，双方发送的每个可靠消息的消息序号将各自从 0 开始依次递增。接受方应将一个未建立连接的地址发来的 0 号可靠消息视为连接发起请求。

由于 4 字节整数的表示范围是有限的，当一个连接传输的可靠消息数量超过了其最大表示范围后，序号应从最小值重新开始。

消息类型

每个 Dragonite 协议的消息包均有以下结构：

VERSION 协议版本 1 signed byte	MSG TYPE 消息类型 1 signed byte
PAYLOAD 内容	

内容消息

类型编号	可靠消息	长度	作用
0	是	不定	携带数据

内容消息是用于携带用户发送内容数据的消息类型。

VERSION 协议版本 1 signed byte	MSG TYPE 消息类型 1 signed byte
SEQUENCE 消息序号 4 signed bytes	
DATA LENGTH 内容长度 2 unsigned bytes	
DATA 内容	

连接关闭消息

类型编号	可靠消息	长度	作用
1	是	8	声明连接关闭

关闭消息用于声明一条连接的关闭。

VERSION 协议版本 1 signed byte	MSG TYPE 消息类型 1 signed byte
SEQUENCE 消息序号 4 signed bytes	
STATUS CODE 状态代码 2 signed bytes	

状态代码用于表示连接关闭的原因。正常情况下应设置为 0。如果需要表示一个特殊状态，也可设置为上层应用需要的任意值。

连接建立与关闭的详细处理流程请见下文对应部分。

回应消息(ACK)

类型编号	可靠消息	长度	作用
2	否	不固定	声明收到消息

回应消息用于连接中的一方在接收到对方发送的可靠消息后发送，表示消息已收到。此消息也含有接收方目前上层应用对数据的读取情况，发送方可用于判断是否需要暂停发

送，等待接收方的处理。

此消息可以包含 0 到多个消息编号(Sequence)，批量表示一段时间内收到的多个消息。

关于批量 ACK 的推荐处理方式请见下文对应部分。

VERSION 协议版本 1 signed byte	MSG TYPE 消息类型 1 signed byte
CONSUMED SEQUENCE 上层读取序号 4 signed bytes	
SEQUENCE COUNT 编号个数 2 unsigned bytes	
SEQUENCE 1 消息序号 1 4 signed bytes	
SEQUENCE 2 消息序号 2 4 signed bytes	
SEQUENCE N 消息序号 N 4 signed bytes	
.....	

心跳消息

类型编号	可靠消息	长度	作用
3	是	6	保持 NAT 映射 检测连接存活

心跳是连接建立后每隔一段时间都会发送的一个用于保持 NAT 与检测连接是否存活的消息。

接收方在收到心跳消息后按照正常处理可靠消息的方式返回 ACK 即可。发送方根据是否在超时时间内收到接收方的 ACK 判断连接是否仍然存活。

推荐的实现细节请见下文对应部分。

VERSION 协议版本 1 signed byte	MSG TYPE 消息类型 1 signed byte
SEQUENCE 消息序号 4 signed bytes	

功能实现

重要概念

发送序号(Sent sequence)：发送方目前发送过的可靠消息中的最大序号

接收序号(Received sequence)：接收方目前接收到的可靠消息中的最大序号

接收连续序号(Received consecutive sequence)：接收方目前接收到的最大可组成完整数据的序号

上层读取序号(Consumed sequence)：接收方上层应用目前读取过的数据中的最大序号

例如，A 发送 1 2 3 4 5 6；B 目前实际收到了 1 2 3 5。

则 A 的发送序号为 6；B 的接收序号为 5；B 的接收连续序号为 3；若上层应用实际只调用了两次 read 读走了 1 与 2，上层读取序号为 2。

连接建立与关闭

一个未建立连接的地址发来的 0 号可靠消息即为 Dragonite 中建立连接的标志。下面是一个连接建立的标准流程：

	Dragonite A		Dragonite B
1.	CLOSED		LISTEN
2.	CONNECTING	→ Reliable Message 0 →	ESTABLISHED

3. ESTABLISHED ← ACK 0 ← ESTABLISHED

当连接一方主动发出了关闭连接的消息后，该方将立即进入停止发送(Sender closed)状态，无法再发送任何数据，但在收到另一方返回的对应关闭消息的 ACK（或超时）前仍可继续接收数据，在接收到对应 ACK 或等待超时后彻底关闭；收到关闭连接消息的一方会立即进入停止发送状态，但仍将继续接收对方可能已发送但未到达的数据，目前在接收缓冲区内的数据也仍可继续读取，当本方的上层读取序号到达了该关闭消息的序号后彻底关闭。

批量 ACK 回应

为避免在高速传输数据时产生过多极小的 ACK 消息增加开销，Dragonite 的 ACK 消息可包含多个 Sequence 序号用于一次性表示多个可靠消息的接收成功。在实际实现中推荐使用每 5-20ms 的间隔发送一次批量回应。

重传机制

发送方应根据从一个可靠消息发出到收到对应 ACK 消息所经过的时间判断网络连接的延迟大小，再根据由此计算出的正常延迟判断可靠消息是否在传输中丢失需要重新发送。在一般情况下，应只利用没有经过重发的消息的 ACK 计算延迟，以避免错误判断出过高延迟。延迟的计算可通过$newRTT = oldRTT * 0.875 + currentRTT * 0.125$以实现渐进式的变化，减小偶然性网络波动的影响。延迟波动变量的计算为$newDevRTT = oldDevRTT * 0.75 + |RTT - currentRTT| * 0.25$。

由于在特殊网络环境下可能存在延迟突然大幅增长的情况，导致所有数据包若根据此前 RTT 进行重发判断，则至少重发一次，无法再仅依据没有经过重发的数据包刷新 RTT 数

值。因此应加入延迟矫正机制，即若两秒之内收到的所有 ACK 消息均来自重发过的消息，则用经过重发的消息计算出的延迟进行一次 RTT 更新。加入此机制后，可保证在两秒或数个两秒后将 RTT 纠正到实际大小。

重传机制应在等待至少$RTT + devRTT$的时长后仍未收到一个可靠消息的对应 ACK 后再认定为丢失，进行重新发送。注意当 dev RTT 小于一次批量 ACK 的时间间隔时，应以 ACK 间隔为准，以避免因远端 ACK 的延迟发送而提前判断为数据包已经丢失。

心跳消息

存在 NAT 的不同网络中 UDP 映射的超时时间并不固定，长至 24 小时短至 30 秒均有可能。Dragonite 推荐的实现为在没有任何数据收发后的每 5-30 秒发送一次心跳消息。

超时时间

由于存在心跳消息用于保证即使连接在没有数据收发时也能保持状态，双方可根据长时间没有收到消息判断对方已经意外断开。至少需要大于心跳消息的发送频率，推荐时间设置为其频率的两倍。

发送限制

发生丢包后，接收方在等待重新传输能够组成完整数据，被上层应用取走前需要将数据全部存在缓存中。另一方面，上层逻辑也有可能长时间不调用读取方法，导致数据堆积在缓存中。由于缓存大小有限，发送方不能在出现这些情况后依然持续发送大量数据。发送方需要根据接收方返回的 ACK 获取接收连续序号与上层读取序号，对目前是否还能继续发送进行判断。推荐的计算方法为根据发送速度对应的每秒消息数量 pps 与延迟毫秒数 RTT 得出$pps \cdot RTT$，这个值是在延迟完全稳定且不丢包的理想情况下正好达到

目标速率所需要的最小窗口大小。但由于实际的网络情况，其需要进行一定程度倍数放大。常见的实现通常默认将发送方发送序号与接收方上层读取序号之间的差大小限制在 2-4 倍最小窗口大小之间。

速度限制

发送方的速率控制方式没有规范。在实际应用中可以直接采用类似令牌桶算法(Token bucket)的方式根据当前网络已知的最大速率进行控制，也可自行实现类似 TCP 拥塞控制算法的机制。