CubeFS卷级别流控方案

1. 背景

CubeFS是一个多租户的分布式文件系统，大集群多租户模式在实际运营中能够更好保障集群资源利用率，从而降低成本，但对稳定性要求也带来了挑战。流量QoS保障是其中较为重要的一个方面，某一个卷的流量突增会冲击存储系统或者整个机房的服务：

- 多个卷的数据分区和元数据分区可能会放置在一台机器，甚至一个磁盘，突增IOPS也会影响到节点和磁盘上的其它卷的IO能力。
- 对存储进程的负载能力也会有冲击，影响存储卷的整体请求质量。
- 影响到进程的其它方面的服务，如网络通讯，进而影响到集群对服务进程的整体状态判断，导致集群管控行为，例如迁移数据，造成了整个集群的抖动。
- 冲击集群上游交换机，不仅影响存储卷，还会影响其它内部的非存储业务。

2. 架构设计

1）管控面需要一个控制单元，收集并管理所有的卷的使用带宽，监控卷流量，工具可调整卷带宽的上限，内部可将数据上报卷metrics带宽告警。

2）要具备均衡客户端的写入能力，例如在限流的情况下，多个线程的写入客户端，要高于单个线程写入的进程。多个线程内的平均写入size大的实际写入流量要高于平均size小的线程。

3）结合当前CubeFS架构，BlobStore作为独立的存储子模块，卷和BlobStore的bucket无法对应，也可能后续存在其它的外部存储系统也无法对应，因此，流控能力不能依赖于每一个子存储系统的流控能力和配合，需要在副本卷和纠删码卷之上维护对于客户端流量的控制，所以将流控实际控制放置在客户端。

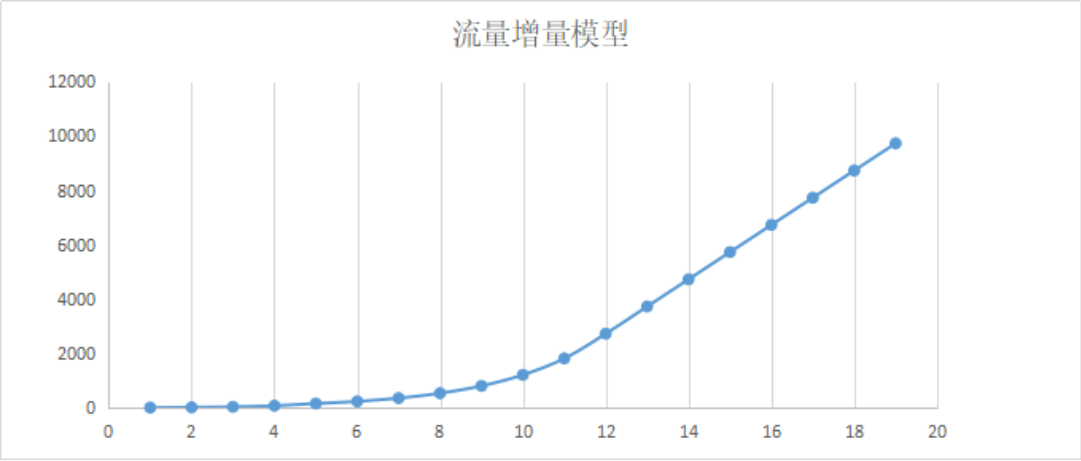
具体模块分工：

- sdk模块提供了client多个客户端（fuse模块、s3模块、hdfs模块）的公共调用的接口，可以集成端侧的流量申请、限流逻辑。
- Master做控制中心管理卷流量和客户流量，不增加额外流控server，减少运维维护压力。
- Master均衡客户端流量，保证volume整体流量调控下平稳，减少流量波动。
- sdk和Master配合，在流量在资源充足的场景下，可以快速增长，尽量避免流控逻辑对客户端的影响。

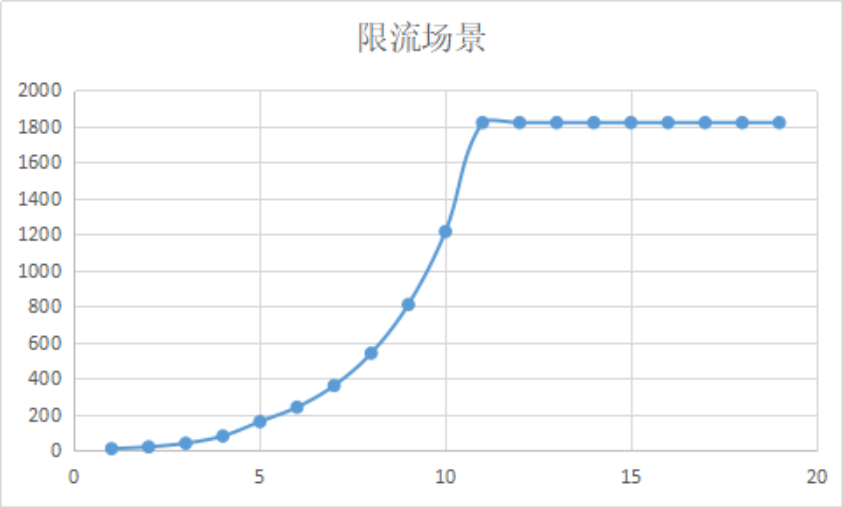
3. 目标模型

3.1 场景

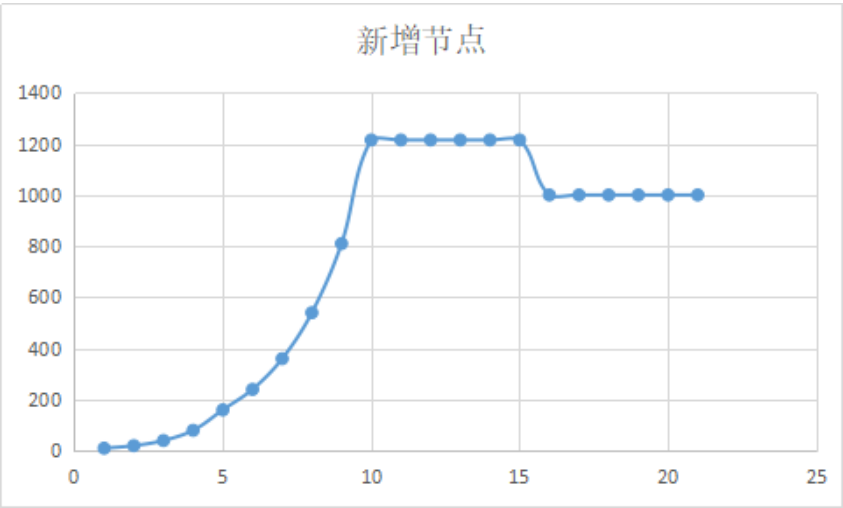
1）client流量持续增长



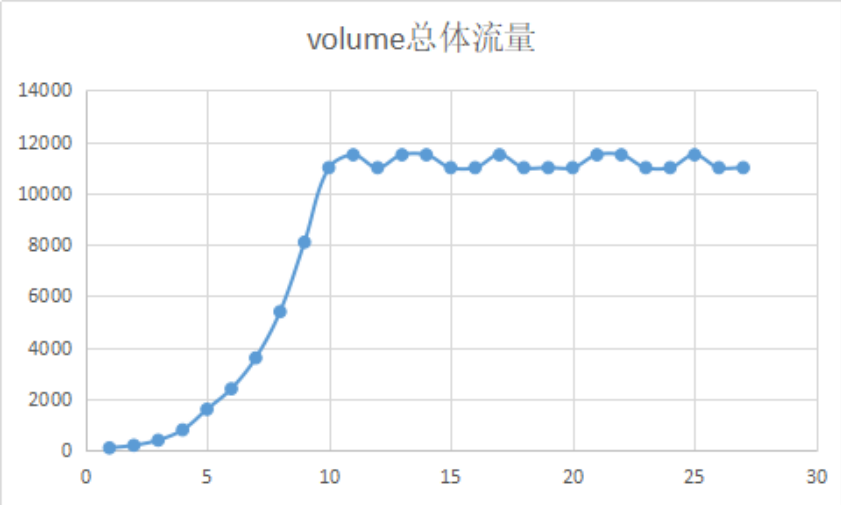
2）client 流量需求超过限制，持平部分达到限流



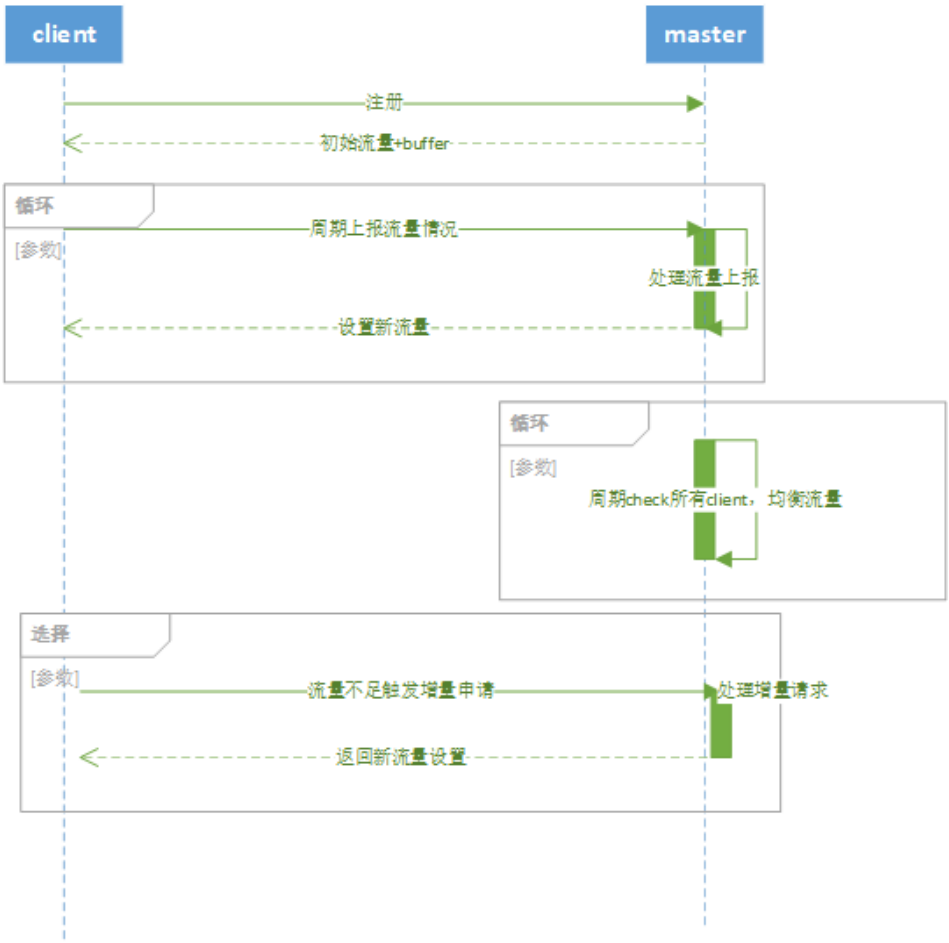
3）客户端之间流量相互影响，其他客户端需求流量增加或者新加入节点后流量开始下降，均衡后流量维持平稳。



4）volume总体流量增长后，在限流水平上下小幅度抖动。fuse场景，客户端数量较少（个位数），总流量限制较少的情况下（200MB以下），抖动略明显，会在5%左右。



3.2 交互



- 初始化，资源充足的情况下，master默认按初始客户端数量，均分卷的流量，但设置上限
- client上报master资源使用情况，获得master的资源配额，在固定时间内内切分多个窗口，当有P%窗口达到限流，则继续请求master再次扩大资源额度。
- master收到增量请求，周期性重新计算全量流量。再次计算预分配buffer并回复给client。
- 流量充足的情况下，client从初始值为100Mb，达到1Gb，最快12秒，10Gb，需要24秒。
- 流量达到设置的卷流量上限的情况下，周期触发Master根据限流比，调整当前上报的client的预分配流量上限。

3.3 master负载

- 活跃客户端上报周期为5s

- 1000个客户端，200/s
- 5000个客户端，1k/s
- 10000个客户端，预估2K/s的请求上限

非活跃客户端，没有读写，没有上报

- 增量申请上报

限制1s一次，服务端限制总量3k/s的限流。

4. 流量分配规则设计

分布式场景下，为减少客户端和master的通讯，一次申请尽量获得最多的流量，流量宽松时但也要避免浪费，为后续新加入的客户端保留空间，流量紧张或者限流时要均衡分配使用流量，对低流量客户端给予相对于自身高比例的流量预期，对于高流量的客户度给予相对于自身低比例的流量预期。

4.1 流量宽松场景

master需要根据客户端使用情况，预分配流量和卷的整体流量使用情况，除了分配客户端当前需要的流量，也可以在buffer中拿出流量（预分配流量buffer），预分配更多的流量，作为一种预测，满足客户端的流量过快上涨需求。

4.2 流量紧张场景

所剩余的buffer已经不能如4.1超额分配，但会最大化来分配剩余buffer池中的流量，在现有buffer中，按目前客户端的使用占比，分配剩余的buffer。此种情况下，流量最差的情况是buffer接近0，只有满足客户端申请的流量，基本没有buffer额度可分配。

4.3 限流场景

已经没有预分配流量buffer，总的实际需求大于配额，现有的流量不能满足所有的需求，不能分配给部分用户，剩余用户没有流量可用。因此，需要计算客户端的实际需求情况（实际需求大于了后面的分配的流量）占比在所有需求的的比例，用此比例来分得自己的流量配额。

4.4 限流极端场景

分得流量不低于128KB，保留客户端最低的流量使用带宽，可配置。

4.5 客户端启动场景

低流量到高流量，相对于自身流量预期会从高到底，采用幂函数方式来支持。

5. 名词解释

- client 预分配流量上限：client计划使用流量
- client 预分配流量buffer：client计划外可用流量，减少申请，减少对master的请求量
- client限流流量：client端被限制的未能使用的流量, client可以通过时间单元消耗流量得出实际流量
- client 使用流量 <= client 预分配流量上限 + client 预分配流量buffer
- client实际流量需求 = client 使用流量 + client限流流量

- volume待分配流量 = 总流量限制 – sum(client的预分配流量上限) – sum(client 预分配流量buffer)
- 限流比 = 限流流量 / 实际流量需求，参考6.4的解释
- volume限流比 = sum(client限流流量) / sum(client实际流量需求)
- client目标流量 = client实际流量需求 * （1 – volume限流比）

6. Client实现

6.1 模块

- 抽象出来公共模块，供副本卷和纠删码使用

```
type LimitManager struct {
    ID                  uint64
    limitMap            map[uint32]*LimitFactor
    enable              bool
    simpleClient        wrapper.SimpleClientInfo
    exitCh              chan struct{}
    WrapperUpdate       UploadFlowInfoFunc
    ReqPeriod           uint32
    HitTriggerCnt       uint8
    lastReqTime         time.Time
    lastTimeOfSetLimit time.Time
    isLastReqValid      bool
    once                sync.Once
}
type LimitFactor struct {
    factorType          uint32
    gridList            *list.List
    waitList            *list.List
    gidHitLimitCnt      uint8
    mgr                 *LimitManager
    gridId              uint64
    magnify             uint32
    winBuffer           uint64
……
}

```

6.2 流程

限流算法

本文采用滑动窗口算法的简化形式，流量是格子内等值的方式来划分时间和流量

格子默认值默认值

```
girdCntOneSecond     = 3 //每秒3个
gridWindowTimeScope  = 10 //维护时间长度
```

代码中grid保留3秒，每秒10个格子，如果最近30个格子，有gidHitLimitCnt个格子达到限流上限，则增量请求master，请求master限制最多1s内一次（可配置）。

支持接口调整gidHitLimitCnt, 可以降低敏感度。

```
func (factor *LimitFactor) CheckGrid() {
……
    grid := factor.gridList.Back().Value.(*GridElement)
    newGrid := &GridElement{
    ……
    }
    factor.gridId++
    factor.gridList.PushBack(newGrid)
    for factor.gridList.Len() > gridWindowTimeScope*girdCntOneSecond {
……
        factor.gridList.Remove(factor.gridList.Front())
    }
    factor.TryReleaseWaitList()


```

限流入口

- 获取master限流，按滑动窗口处理，对于用户请求不会透支流量，也不会拒绝，持续累计以达到触发新流量获取的条件。
- 会触发异步增量申请资源，master不回包或者阻塞延迟回包，则继续维持当前流量配额

```
func (factor *LimitFactor) alloc(allocCnt uint32) (ret uint8, future *util.F
    atomic.AddUint64(&factor.valAllocApply, uint64(allocCnt))
……

    type activeSt struct {
        activeUpdate bool
        needWait     bool
    }
    activeState := &activeSt{}
……
    grid := factor.gridList.Back().Value.(*GridElement)
    if factor.mgr.enable && (factor.waitList.Len() > 0 || atomic.LoadUint64(
        activeState.needWait = true
        future = util.NewFuture()
        factor.waitList.PushBack(&AllocElement{
            used:    allocCnt,
            future:  future,
            magnify: factor.magnify,
        })

        if grid.hitLimit == false {
            factor.gidHitLimitCnt++
```

```
            if factor.gidHitLimitCnt >= factor.mgr.HitTriggerCnt {
                tmpTime := time.Now()
                if factor.mgr.lastReqTime.Add(time.Duration(factor.mgr.ReqPe
                    factor.mgr.lastReqTime = tmpTime
                    ……
                    activeState.activeUpdate = true
                    go factor.mgr.WrapperUpdate(factor.mgr.simpleClient)
                }
            }
        }
        grid.hitLimit = true
        return runLater, future
    }
    return runNow, future
}

```

- 普通上报或者增量请求的回包，如果volume资源用尽，收到master的buffer会为0，维持最基本的流量128KB
- 普通上报回包，更新grid限流额度

幂函数计算预分配流量

- 根据实际使用情况，预计可能的增量流量。
- 以300MB为分界线，300MB以下相对快速增长，300MB以上相对缓慢增长。

```
func (limitManager *LimitManager) CalcNeedByPow(limitFactor *LimitFactor, us
    if limitFactor.waitList.Len() == 0 {
        return 0
    }
    if limitFactor.factorType == proto.FlowReadType || limitFactor.factorTyp
        used += limitFactor.GetWaitTotalSize()
        if used < 128*util.KB {
            need = 128 * util.KB
            return
        }
        need = uint64(300 * util.MB * math.Pow(float64(used)/float64(300*ut
    } else {
        if used == 0 {
            used = uint64(limitFactor.waitList.Len())
        }
        need = uint64(300 * math.Pow(float64(used)/float64(300), 0.8))
    }

    return
}
```

流量上报

- 周期性上报和增量上报的入口。
- 计算出使用流量的和预分配流量，上报到客户端。
- 如果用户长时间没有流量上报，则不会周期上报给master，减少通讯，master也会删除掉该客户端的流量维护。直到新的有效读写时才上报master。

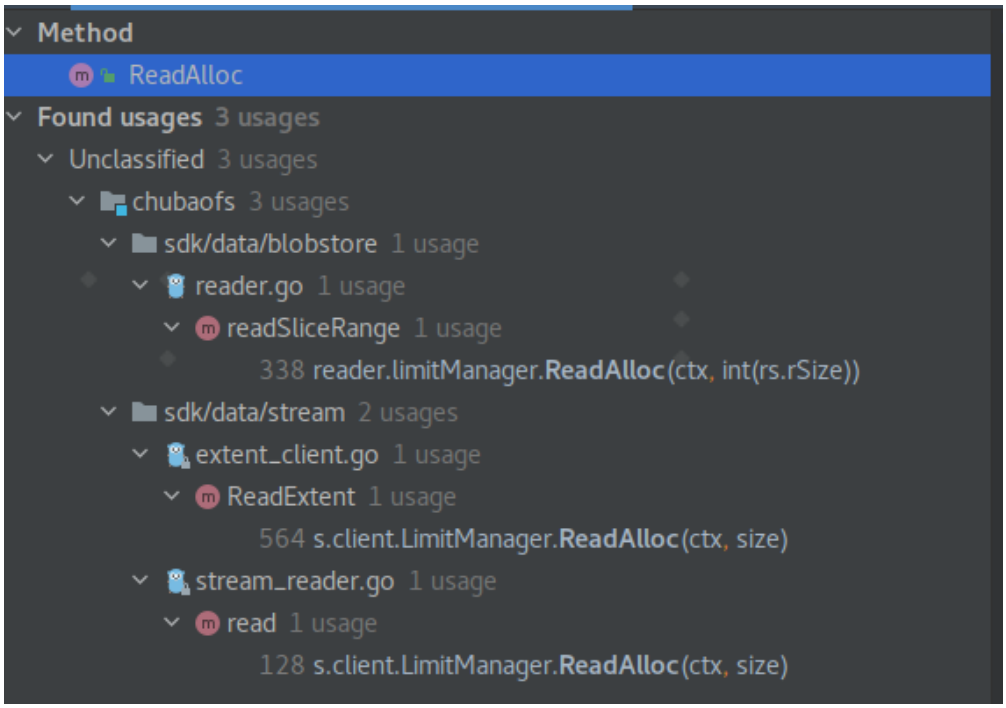
```
func (limitManager *LimitManager) GetFlowInfo() (*proto.ClientReportLimitIni
    ……
       if griCnt > 0 {
            timeElapse := uint64(time.Second) * uint64(griCnt) / girdCntOneS
            if timeElapse < uint64(qosReportMinGap) {
                log.LogWarnf("action[GetFlowInfo] type [%v] timeElapse [%v]
                    proto.QosTypeString(limitFactor.factorType), timeElapse)
                timeElapse = uint64(qosReportMinGap) // time of interval get
            }
            reqUsed = uint64(float64(reqUsed) / (float64(timeElapse) / float
        }

        factor := &proto.ClientLimitInfo{
            Used:       reqUsed,
            Need:       limitManager.CalcNeedByPow(limitFactor, reqUsed),
            UsedLimit:  limitFactor.gridList.Back().Value.(*GridElement).lim
            UsedBuffer: limitFactor.gridList.Back().Value.(*GridElement).buf
        }
……
}
```

6.3 接口支持

1. 抽象接口

```
type SimpleClientInfo interface {
    GetFlowInfo() (*proto.ClientReportLimitInfo, bool)
    UpdateFlowInfo(limit *proto.LimitRsp2Client)
    SetClientID(id uint64) error
}
```

2. 流量check调用



3. blobstore cache

因为blobstore存储，可能启用副本组cache，所以路径上要针对处理。

纠删码启用blobstore的流量控制，调用公共的流量控制模块

1）路径：

写：写blobstore –> 写副本组

读：读副本组 –> 读blobstore–> 写副本组

2）cache命中的情况下：

不计入限流。仅考虑blobstore的读写限流

7. Master实现

7.1 流量初始化

1. client启动后，master给client初始default流量

```
func (qosManager *QosCtrlManager) initClientQosInfo(clientID uint64, host st
    clientInitInfo := proto.NewClientReportLimitInfo()
    cliCnt := qosManager.defaultClientCnt
    if cliCnt <= proto.QosDefaultClientCnt {
        cliCnt = proto.QosDefaultClientCnt
    }
    if len(qosManager.cliInfoMgrMap) > int(cliCnt) {
        cliCnt = uint32(len(qosManager.cliInfoMgrMap))
    }
```

7.2 处理上报

1. 查看系统给client端准备的流量，根据当前上报的需求流量，如果有buffer，调整反馈流量。
2. 客户端流量不足会有触发再次主动上报申请，也会调用该接口。

```
// handle client request and rsp with much more if buffer is enough according rules of allocate
func (serverLimit *ServerFactorLimit) updateLimitFactor(req interface{}) {
    rsp2Client.UsedLimit = assignInfo.UsedLimit
    rsp2Client.UsedBuffer = assignInfo.UsedBuffer

    // flow limit and buffer not enough,client need more
    if (clientReq.Need + clientReq.Used) > (assignInfo.UsedLimit + assignInfo.UsedBuffer) {
        log.QosWriteDebugf("action[updateLimitFactor] vol [%v] clientID [%v] type [%v], need [%v] used [%v], used limit [%v]",
            serverLimit.qosManager.vol.Name, clientID, proto.QosTypeString(factorType), clientReq.Need, clientReq.Used, clientReq.UsedLimit)

        dstLimit := serverLimit.getDstLimit(factorType, clientReq.Used, clientReq.Need)
……
}
```

3. 保护master

单独针上报接口增加了限流逻辑

```
    m.cluster.QosAcceptLimit.WaitN(ctx, 1)
```

7.3 周期性计算客户端流量

volume限流比 = sum（client限流流量） / sum(client实际流量需求)

client目标流量 = client实际流量需求 * （1 – volume限流比）

```
func (qosManager *QosCtrlManager) assignClientsNewQos(factorType uint32) {
    qosManager.RLock()
    if !qosManager.qosEnable {
        return
    }
    serverLimit := qosManager.serverFactorLimitMap[factorType]
    var bufferAllocated uint64

    // recalculate client Assign limit and buffer
    for _, cliInfoMgr := range qosManager.cliInfoMgrMap {
        cliInfo := cliInfoMgr.Cli.FactorMap[factorType]
        assignInfo := cliInfoMgr.Assign.FactorMap[factorType]
```

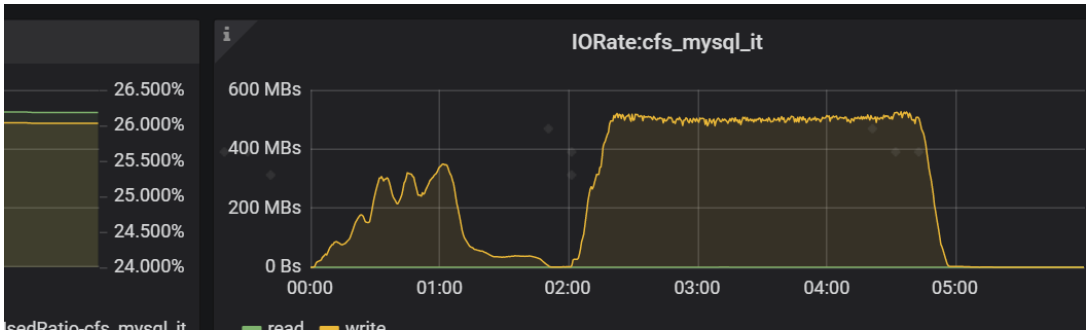
7.4 动态调整限流比

限流情况下，根据客户端的需求，服务端计算流量，并根据前面提到的限流比来分配流量。但实际情况是，因为业务行为未必连贯，导致实际分配的流量在客户端部分浪费，没有被使用起来，例如限流1GB，分配给所有客户端流量为1GB，但因为时间差等因素，实际客户端真实反馈的使用流量为0.9GB。因此，需要在服务端适当调整限流比，放大客户端的需求流量，例如供给到1.1GB，最终实际使用1GB，让客户端得以达到限流流量。

```
func (qosManager *QosCtrlManager) updateServerLimitByClientsInfo(factorType uint32) {
……
    if serverLimit.CliUsed < serverLimit.Total {
            if serverLimit.LimitRate > -10.0 && serverLimit.LastMagnify < serverLimit.Total*10 {
                serverLimit.LastMagnify += uint64(float64(serverLimit.Total-serverLimit.CliUsed) * 0.1)
            }
        } else {
            if serverLimit.LastMagnify > 0 {
                var magnify uint64
                if serverLimit.LastMagnify > (serverLimit.CliUsed - serverLimit.Total) {
                    magnify = serverLimit.CliUsed - serverLimit.Total
                } else {
                    magnify = serverLimit.LastMagnify
                }
                serverLimit.LastMagnify -= uint64(float32(magnify) * 0.1)
            }
        }
        serverLimit.LimitRate = serverLimit.LimitRate * float32(1-float64(serverLimit.LastMagnify)/float64(serverLimit.Allocated+serverLimit.NeedAfterAlloc))
```

8. 效果

这是一个真实示例，带宽在达到500MB后保持平稳。



9. 使用说明

文中为方便讲解，以流量为例。实际支持流量和IOPS的读和写的四种类型。

- 创建卷时启用QOS：

```
curl -v "http://192.168.0.11:17010/admin/createVol?name=volName&capacity=100
```

启用qos，写流量设置为10000MB

- 获取卷的流量情况

```
curl  "http://192.168.0.11:17010/qos/getStatus?name=ltptest"
```

- 获取客户端数据

```
curl  "http://192.168.0.11:17010/qos/getClientsInfo?name=ltptest”
```

- 更新服务端参数，关闭、启用流控，调节读写流量值

```
curl  "http://192.168.0.11:17010/qos/update?name=ltptest&qosEnable=true&flow
```

10.后续

- 目前配置化和工具化依然不够完善，需要加强。
- 因为上报反馈的方式的过程是基于自身数据的变化的调整，多个条件例如流量和IOPS同时达到限流标准，会导致误判为自身流量调整引起，叠加后引起波动，需要继续优化。
- 部分代码需要整理，例如部分magnify相关的代码已经废止。
- Zone维度，对datanode限流逻辑本文未介绍，但该逻辑还较为简单，需要优化。