The design of gonet

instructor: 傅理
http://github.com/xtaci

CHALLENGE

Build a single logical server, holding millions of players, and the players will interacts with each other.

basically, we need...

1. scalable game server
2. scalable database backend
3. a mechanism for IPC and RPC

后端的现有方案分析(mysql)：

尝试采用mysql *读写分离*:

一个 writable instance，将数据库日志replicate到若干个 readonly instance，实现数据同步。每个instance 都有相同的数据副本

缺点:

replication traffic(WAL)随着 readonly instance 节点数增加而增加, 千兆交换的极限, 大概不会超过20个.

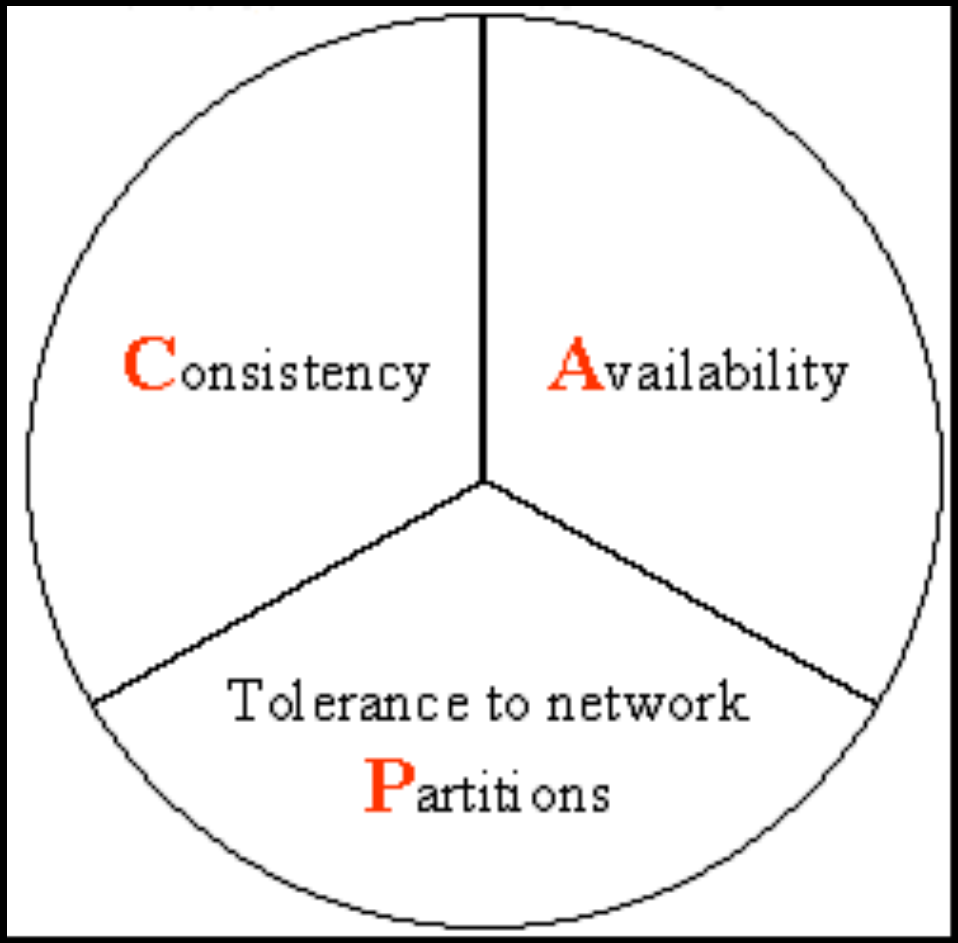
Scalable DB backend solution:

采用mongodb:

支持Auto-Sharding方式，这是一种可以水平扩展的模式，即一个表(mongo中叫collection)/或不同的表中的数据可以分摊到各个节点上，构成一个单一的逻辑数据库.

缺点:
没有RDBMS严格的一致性，ACID。

CAP 原理



CAP原理指的是，一致性，可用性，分区容忍这三个要素最多只能同时实现两个。

一致性

所有节点看到相同的数据。

不能出现，一个玩家看到我10级，另一个玩家看到9级。或者我第一次看到是10级，第二次看到是9级。

分区容忍性

在出现网络分区的情况下，分离的系统也能正常运行。

(例如，路由器坏掉，导致本来是一组的集群，分离为两个独立的部分。)

[1,2,3,4,5,6]---> [1,2,3] , [4,5,6]

可用性

对于一个可用性的分布式系统，每一个非故障的节点必须对每一个请求作出响应。(即必须返回一个结果)

The CAP theorem states that:

网络分区不可避免(集群)，因此，我们只能在一致性和可用性之间做选择。

mongodb 的选择

```
const (
    Eventual  mode = 0
    Monotonic mode = 1
    Strong mode = 2
)
```

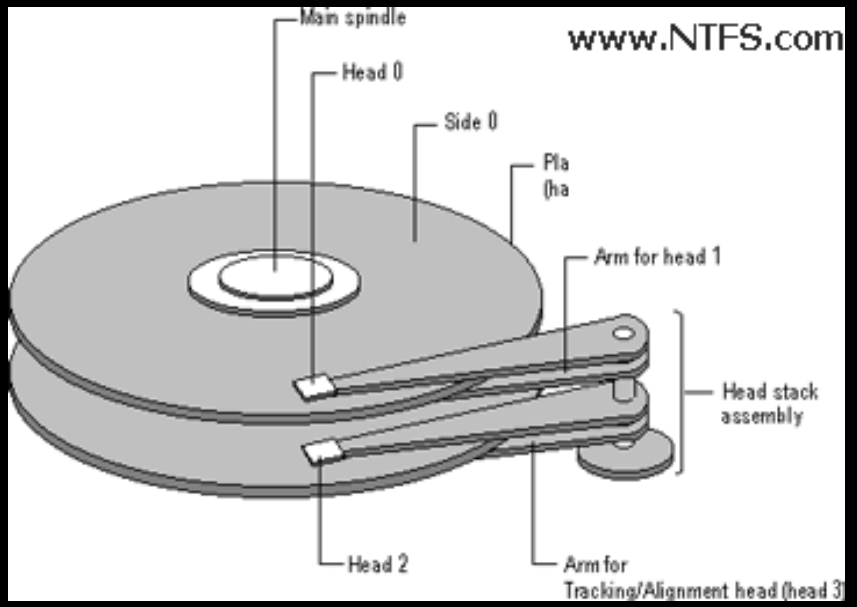
Eventual, 最弱(最终)的一致性(有可能乱序,10,9,10)
Monotonic, 单调一致性, (不乱序, 但不是最新,1,2,3...)
Strong, 完全一致, (只访问主节点, 负载均衡不起作用)

根据业务场景来选择, gonet默认是Monotonic.

关于磁盘IO

随机IO很慢(要旋转磁头)

顺序IO很快(盘面旋转)



把数据组织为一块数据写入，会减少IO次数

RDBMS如果表多了，会很慢。
(random seek： 100seek /second)
NoSQL通常就快在这里。

gonet玩家数据持久化策略

Linux内核的磁盘数据持久化有两种方式：

1. 超过一定的时间，[pdflush]将脏数据写入磁盘。（vm. dirty_expire_centisecs）

2. 脏数据数量超过一定值，写入磁盘。
(vm. dirty_bytes)

gonet仿照linux内核，也提供这两种机制：

1. 超过一定的写操作数目，写入数据库
2. 超过一定的时间，写入数据库。

Scalable Game Server

设计考虑：

1. 一种支持并行的语言：没有side-effect (erlang要适应func. prog)，门槛低，上手快，golang是最佳选择。

2. GS 必须为同构的，即玩家在任何一个GS上登录，都一样。

3. GS之间透明交互，以构成：

A Single Logical Game Server

gonet网络模型

1. 一个goroutine对应一个 connection, 类似于线程模型.

2. 一个connection包含一个Session对象

ps : 网络底层golang用 epoll 实现IO复用

Session定义：

Session表示一个来自客户端的连接, 对应一个玩家。

每个Session包含一个消息队列, 主要用于接收来自其他玩家的IPCObject

```
type Session struct {
    IP net.IP
    MQ chan IPCObject   <--- 左边这个
    User    *User
    .....
```

玩家消息循环

```
for {
    select {
        case msg, ok := <-in:    // 网络
        case msg, ok := <-sess.MQ: // IPCObject
        case <-std_timer:  // 处理连接超时持久化等
    }
}
```

如何实现玩家交互？

IPCObject + HUB

IPCObject

```
type IPCObject struct {
    SrcID      int32  // sender id
    DestID     int32  // destination id
    Multicast  bool   // group delivery
    Service    int16  // service type
    Object     []byte // json formatted object
    Time       int64  // sent time
    MarkDelete bool   // for db mark as delete
}
```

是唯一的IPC/RPC数据交换方式

欧氏几何公理

1. 任意两个点可以通过一条直线连接。
2. 任意线段能无限延伸成一条直线。
3. 给定任意线段，可以以其一个端点作为圆心，该线段作为半径作一个圆。
4. 所有直角都全等。
5. 若两条直线都与第三条直线相交，并且在同一边的内角之和小于两个直角，则这两条直线在这一边必定相交。

(欧几里德用这5个公设，建立了整个几何学大厦)

给我一个IPCObject，我可以撬动整个地球！

简洁的内涵是复杂系统稳健的根基

$E=mc^2$

代码从9000行，写到7000行

IPCObject的特点：

1. 可以携带任何数据；
2. 不需要知道对方是在哪个服务器，只需要指明发给谁；
3. 仅有一个函数调用
4. 传输格式和存储格式完全一致。

```
func Send(src_id, dest_id int32, service int16,
                multicast bool,
                object interface{}) (ret bool)
```

object 会被json.Marshal()成 []byte 数据

IPCObject应用举例：

玩家对玩家发消息
玩家对联盟发消息
GM对玩家
SYS对玩家（id号为0的特殊用户）
发邮件、发消息、发宝石等等。。。。。

包罗万象，囊括海内。

Example:

```
ipc.Send(1, 2, ipc.SERVICE_PING, false, "Hello")
```

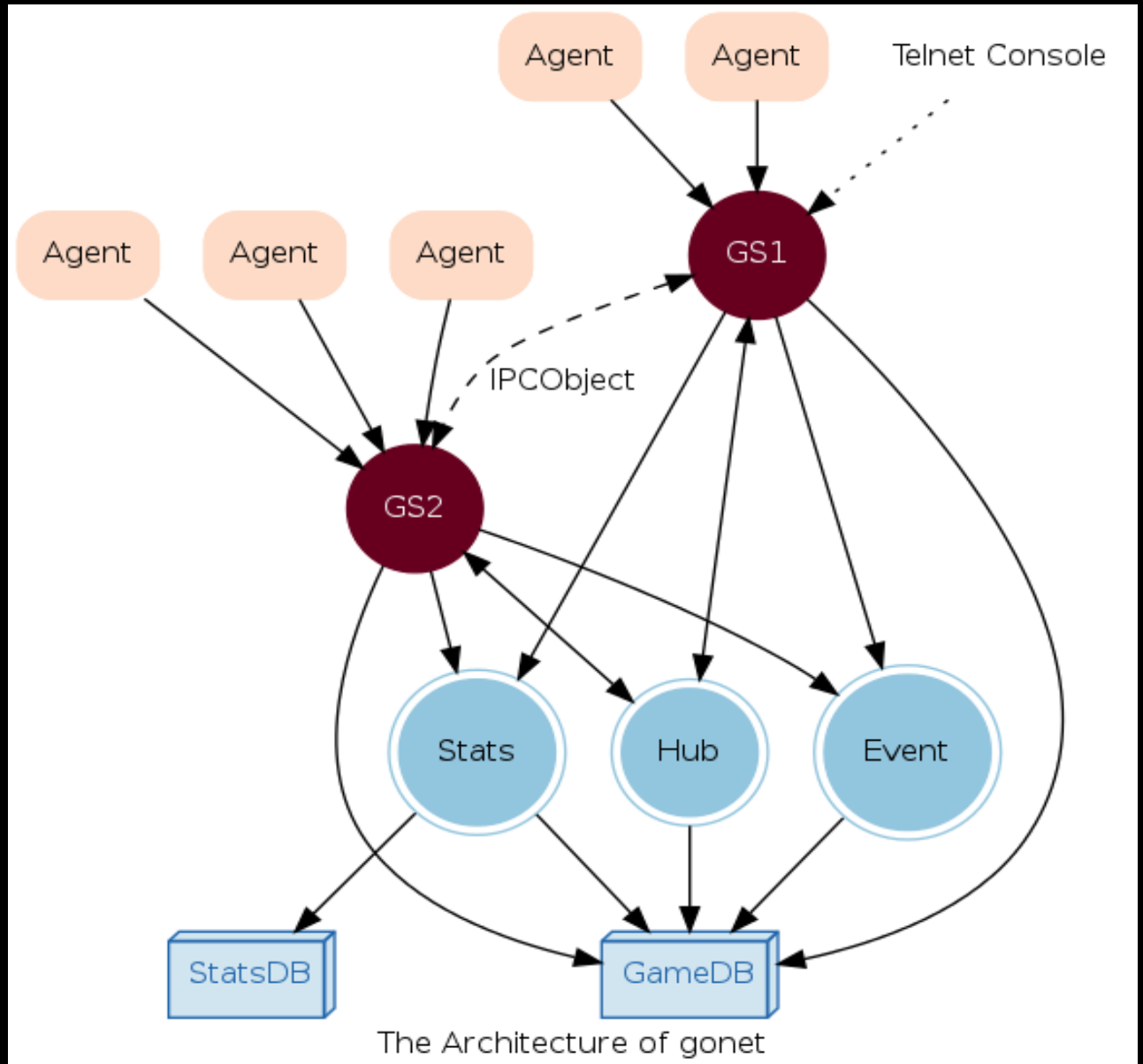
```
obj := &SomeType{txt:"good job"}
ipc.Send(1, 2, ipc.SERVICE_TALK, false, obj)
```

HUB Server(HUB):

目的:记录玩家是否登录，在哪个GS登录的。
并完成跨GS之间的玩家消息转发.

HUB只存在一个.

The whole picture



The Architecture of gonet

IPCObject持久化设计

我们不仅在线时能用IPCObject
离线时也能用

Scene #1

如是我闻:
一时 Alice, Bob 在同一服务器登录, Alice 向 Bob 发消息。

GS检查到Bob也在同一服务器，于是直接发送到Bob.MQ(消息队列)
队列如下:

|Alice| --> | MSGn| MSGn-1|...|MSG1|

(PS. 默认消息队列长度为128)

| Alice| MSGn-1| MSGquit|...|MSG1|

大多数情况下，Bob处理了alice消息，就丢弃。
但如果bob 尚未处理alice这条消息就离线。
那么**所有未处理消息，会push到db**.

通过 len(chan) 可以检查未处理的消息个数:

```
close(sess.MQ)
for len(sess.MQ) > 0 {
    ipcobject := <-sess.MQ
    forward_tbl.Push(&ipcobject)
}
```

Scene #2
Alice在线，向离线Bob发

Send函数发现Bob并未在本GS登录，于是把消息转发到HUB，HUB发现Bob不在任何一台GS登录，最后：

HUB push消息到db

forward_tbl.Push(&ipcobject)

Scene #3
Alice Bob 在不同服务器登录

Send函数发现Bob并未在本GS登录，于是把消息forward到HUB，HUB发现Bob在另一台GS登录，于是:

HUB把消息forward到Bob所在的GS

Scene #3 continued...

消息forward 到Bob所在GS后，在这个间隙，Bob可能离线，于是:

GS push 消息到db

Scene #4
玩家登录

登录后，启动过程直接从db中：

objs := forward_tbl.PopAll(user_id)

pop出所有数据，并从db中删除消息。

Scene #4 continued

db累计了大量离线时收到的数据，有可能超过session.MQ 这个channel的长度。(默认128)

线性的消息处理会Blocking Forever....

解决办法：

go LoadIPCObjects(sess.User.Id, sess.MQ)
(单独一个goroutine异步喂IPCObject)

Scene #4 continued

在喂数据期间，闪断。

即，喂了一半。

解决办法：

把另一半消息 **push到db**

结论：

IPCObject is
99.99%
safe

只有掉电，会丢掉正在被处理中的消息。

HUB

the IPCObject packet switcher

问题：

GS之间的消息都要通过HUB转发，HUB能hold住么？

理论极限：

根据调查，主流1Gbps NIC能支撑的PPS (packet per second)区间为：
在平均60字节的packet大小下，速率：

RX: 680,000 pps,
TX: 500,000 ~ 840,000 pps

不好的nic driver，会导致效率降低一半。

（后面有数据来源链接，以intel网卡为主）

网络优化：

1. 用最新的内核
2. 用最新的网卡驱动
3. 绑定网络数据包处理到单独CPU core, (cpu affinity)
4. GS到HUB之间的socket通信，开启Nagle Algorithm. (打包发送更有效率)

(注：Nagle算法go语言默认是关闭的，其他多数语言默认是开启的)

提高HUB并发的方法

1. 对于玩家状态信息(在线否, 位于哪个GS等信息的读取)采用行级锁(记录锁)

2. 一个GS对应一个HUB的goroutine, 顺序处理, 而不是一个请求对应一个。几十万 个goroutine不断的诞生, 消亡, GC hold不住, 会导致系统非常卡, 也破坏CPU cache.

3. 因此, 对每个GS要开一个很大的消息队列(100K?), 接受来自GS的消息, 排队处理。(一个GS对应一个chan)

行级锁：

```
type PlayerInfo struct {
    Id          int32
    State       byte
    .......
    Host        int32    // host
    _lock       sync.Mutex // Record lock
}
```

源码结构

源码结构

agent -- GAME服务器

cfg -- config.ini 读取

db -- 数据库驱动

event -- EVENT服务器

gamedata -- 游戏数值处理

helper -- 辅助函数，candy funcs....

hub -- HUB服务器

inspect -- Telnet Console for GS

misc -- 算法等

scripts -- awk bash 脚本

stats -- 统计服务器

types -- 玩家数据结构

事件服务器(ES)

定义：

用于处理定期发生的事件，如：
升级CD，（到某个时间点，确定升级完成）
系统推送，（如在某个时刻推送一个邮件）

ES处理的，是和玩家是否上线都无关的任务。

定时器管理

问题：

100万个玩家，有100万个建筑升级事件，等待时间长短不等，短则几秒，多则几天，怎么高效的设计定时器？

1. 肯定不需要用一个事件一个goroutine去sleep()等待（不现实）

2. 也肯定不需要每一秒都去检查几天后才升级好的建筑。

思路：

可以尝试定义一组列表。

每秒都检查的，每30秒检查的，每1min，每30min...

例如，CD时间为40秒的，寻找最近的一个间隔，即30秒。

问题：

1. 剩下的10秒怎么办：
2. 怎么划分间隔比较合理？避免在某个列表不要累计太多的任务。

gonet的方式,

按 2的n次方划分:
2^0 = 1 秒
2^1 = 2 秒
2^2 = 4 秒
...
2^16 = 65536 秒 (18小时)

默认共16个时间间隔

通过一秒的timer, 检查各个列表, 只需要低位mask就可以判断该列表是否需要处理。
mask := 1<< sec -1

(sec是经过的秒数)

列表跳转

我们从2^16列表开始检查，因为2^n次方的特性，2^16必然也是 2^15, 2^14... 2^0 间隔。

如果剩余的时间，小于当前所在的时间隔。
从 2^N 移动到 2^(N-1) 这个间隔列表，而且只移动一级。
例如，我们有一个5秒的CD事件：

位于 2^2 间隔，触发，剩余时间 1s < 4s，
移动到2^1间隔列表，
检查2^1列表时候，剩余时间 1s < 2s，
移动到 2^0间隔列表。

游戏数值处理

设计思路：

WYSIWYG：(所见即所得)

TBL | F1 | F2|...|Fn|
R1 | V | V | ... |V|
R2 | V | V | ... |V|
...
Rn | V | V | ... |V|

直接映射策划的excel二维表。

数值的三个维度

(TBL, ROW, FIELD) -> Value

获取数值的方式:

GetInt(TBL, ROW, FIELD)
GetFloat(TBL, ROW, FIELD)
GetString(TBL, ROW, FIELD)

由逻辑去决定数值类型。

数值热加载

kill -HUP agent

便捷的csv读取

```
pattern := os.Getenv("GOPATH") +
"/src/gamedata/data/*.csv"

files, err := filepath.Glob(pattern)
```

打开目录下 *.csv，读入每个csv到hashmap。

＝＝＝＝＝＝＝＝＝＝

于是你只需要把csv扔到data下，什么都不用管了

并且，保证原子性！

协议代码自动生成

处理内部通用协议

api.txt :
packet_type:1
name:user_login_req
payload:user_login_info
desc:用户登陆请求包

proto.txt
#用户登陆发包
user_login_info=
mac_addr string
client_version integer
new_user boolean
user_name string
===

处理文本就用最善处理文本的工具！

AWK:

- 语法简单(类C, 半天学会)
- 完全用Regular Expression
- 代码量很小, 易于维护
- 不需要其他软件, 系统自带

Bash:

- 把AWK输出拼接为最终的.go源码文件

```
BEGIN { RS = ""; FS ="\n"
print "var Code map[string]int16 = map[string]int16 {"
}
{
    for (i=1;i<=NF;i++)
    {
        if ($i ~ /^#.*/) {
            continue
        }

        split($i, a, ":")
        if (a[1] == "packet_type") {
            array["packet_type"] = a[2]
        } else if (a[1] == "name") {
            array["name"] = a[2]
        } else if (a[1] == "payload") {
            array["payload"] = a[2]
        } else if (a[1] == "desc") {
            array["desc"] = a[2]
        }
    }

    if ("packet_type" in array && "name" in array) {
        print "\t\""array["name"]"\":"array["packet_type"]",\t// "array["desc"]
    }

    delete array
}
END {  print "}\n"  }
```

生成的代码片段

```
var Code map[string]int16 = map[string]int16{
    "ping_req":          0,   // PING
    "login_req":         1,   // 登陆
    "logout_req":        2,   // 登出
    "changescore_req":   3,   // 改变分数
    "getlist_req":       4,   // 获取列表
    "raid_req":          5,   // 攻击
    "protect_req":       6,   // 加保护
    "free_req":          7,   // 结束攻击
    "getinfo_req":       8,   // 读取玩家信息
    "adduser_req":       9,   // 注册一个新注册的玩家
    "forward_req":       100, // 转发IPC消息
    "forwardgroup_req": 101, // 转发IPC消息到联盟
}
```

The Telnet Console for GS

神说，要有光，就有了光。
神看光是好的，就把光暗分开了

一个小的telnet服务器

可以在线观察玩家数据。

```
xtaci@~/gonet/src$ telnet localhost 8800
Trying ::1...
Connected to localhost.
Escape character is '^]'.
GameServer Console
type 'help' for usage
gonet> l
[
        3
]
length: 1
gonet> p 3
&{IP:<nil> MQ:0xc202c57000 User:0xc2000b57e0 Estates:0xc2000c1640 Soldiers:<nil> Heroes:<nil> Grid:<nil>
Events:[] LoggedIn:false KickOut:false ConnectTime:2013-06-21 16:59:54.863986113 +0800 CST LastPacketTime:
1371805194 LastFlushTime:1371805194 OpCount:0 LatencySamples:0xc202b52fd0}
gonet> p 3.User
{
        "Id": 3,
        "Type": 0,
        ...
}

gonet> p 3.Events
null
length: 0
```

实现方式

Lex/Yacc

golang自带一个goyacc工具做语法分析

词法分析用的nex（来自standford)

几个重要算法

玩家动态排名算法

问题:
有大量的玩家,
[id1, id2, ..., idN]
每个玩家有一个分数:
[score1,score2,....., scoreN]
分数不断变动。

如何获取排名为N的玩家？
(类似于coc找实力相当的玩家)

先排序，再获取第N个？

太慢，最快的排序算法，都需要 O(N*logN) -- quicksort。

数据库 order by？更不靠谱，数据库排序会用到 filesort，有IO。

gonet采用的方法

基于红黑树的扩展(动态有序统计）

```
type Node struct {
    left   *Node
    right  *Node
    parent *Node

    score int // the score
    size  int // the size of this subtree
    color int

    data interface{} // associated data
}
```

效率: O(logN)

```
          [score:1042 size:1]
      *[score:1043 size:3]
          [score:1044 size:1]
 [score:1045 size:9]
          [score:1046 size:1]
      *[score:1047 size:5]
              *[score:1048 size:1]
          [score:1049 size:3]
              *[score:1050 size:1]
```

size表示这棵子树的大小。
具体工作方式请看misc/alg/dos/代码

确定玩家IP来源地区算法：

```
"1.0.0.0","1.0.0.255","16777216","16777471","AU","
Australia"
"1.0.1.0","1.0.3.255","16777472","16778239","CN","China"
"1.0.4.0","1.0.7.255","16778240","16779263","AU","
Australia"
"1.0.8.0","1.0.15.255","16779264","16781311","CN","China"
"1.0.16.0","1.0.31.255","16781312","16785407","JP","Japan"
"1.0.32.0","1.0.63.255","16785408","16793599","CN","China"
"1.0.64.0","1.0.127.255","16793600","16809983","JP","
Japan"
.....
```

FROM:TO:.....: CN:China

（来源：http://www.maxmind.com/zh/home）

用区间树处理(红黑树的扩展)

```
      [5 15 m->16 Value: F]
           [4 14 m->14 Value: E]
[3 13 m->16 Value: D]
           [2 12 m->12 Value: C]
      [1 11 m->12 Value: B]
           [0 10 m->10 Value: A]
```

查询效率，依然是O(logN)，即可以随时判定其当前登入的国家。

Cheat Detection

问题定义：

如何判定玩家没有CD欺骗，例如：
B的修建必须依赖A完成，建造顺序为：

A .... 30 min ... B

同步调用的情况下，完全依照服务器的时间，修建的时候，服务器直接判定B不能修建即可。

完全异步的情况：

正常情况下，客户端判定A已经建成，就容许B修建，服务器端只做结果验证，为了流畅的体验，以客户端时间为准(启动校时)。

如何欺诈：

Eve连续发给服务器三个数据包：

1. 40min之前A开始建造
2. 确认A在10min之前建造完成。
3. 开始建造B

基于固定值的做法：

数据包宣称的时间，与数据包真实到达的时间的差值，必须不能太大，例如 <10s。
基本能解决问题，但不优雅。

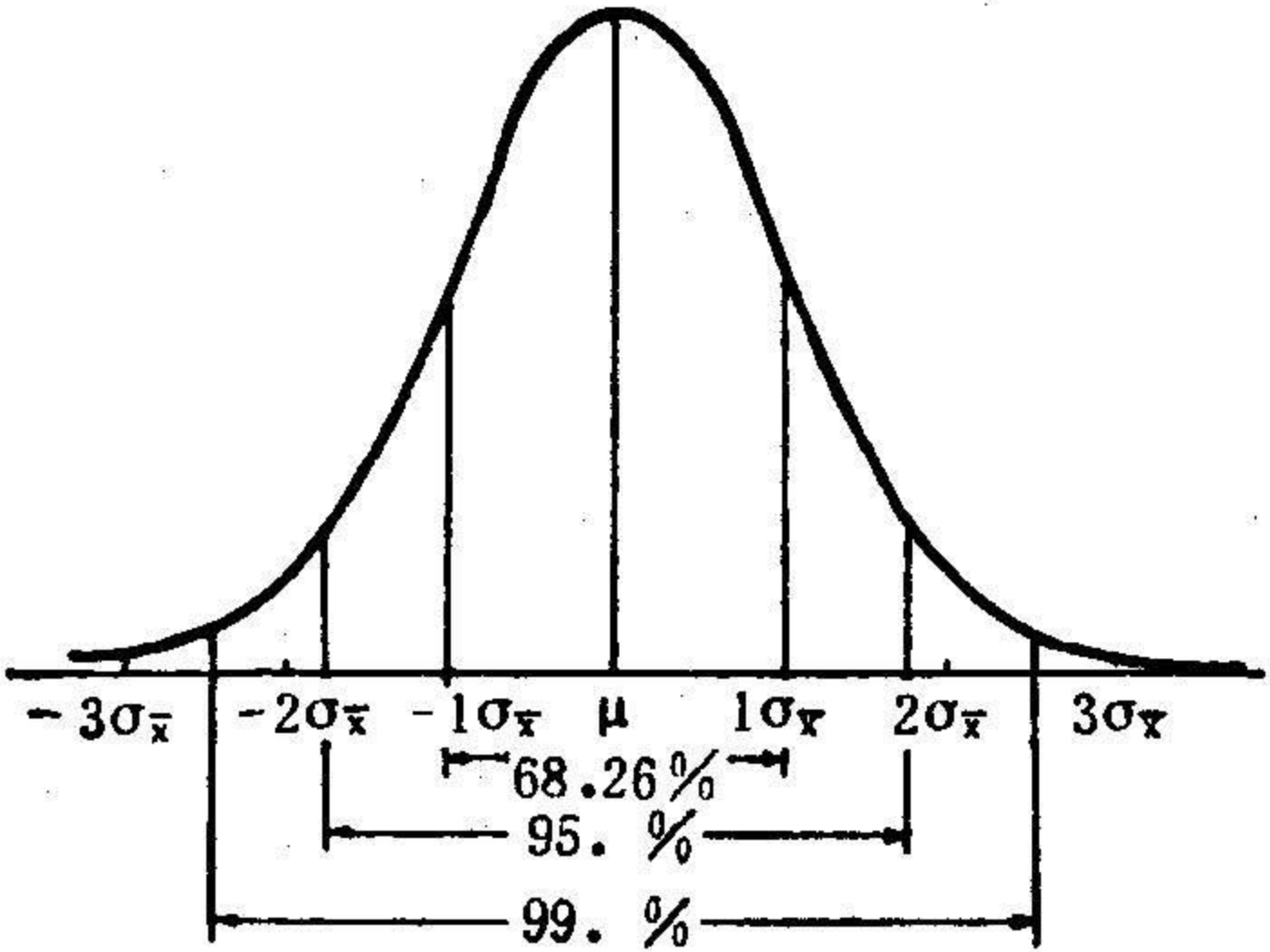
区别对待高延迟玩家，和低延迟玩家，例如：
局域网玩家的延迟应该在 ～5ms 附近抖动。
GPRS的玩家也许在 ～2秒 附近抖动。

基于统计的做法

采集网络延迟作为样本，让客户端携带timestamp
即：latency :=
(数据包到达时服务器时间 － 数据包携带的客户端时间)

我们假设网络延迟的时间，符合正态分布：
即 ping 值的抖动。

采集一定数量的样本，默认128个，时间窗，FIFO。


$-3\sigma_{\bar{x}}$
$-2\sigma_{\bar{x}}$
$-1\sigma_{\bar{x}}$
μ
$1\sigma_{\bar{x}}$
$2\sigma_{\bar{x}}$
$3\sigma_{\bar{x}}$
68.26%
95. %
99. %

结论：

位于2-sigma的之外的，都是小概率事件，<5%。

基于统计的方式，可以动态调整对欺骗的判定值，更贴近真实的情况。

References:

http://xiezhenye.com/2012/12/mongodb-sharding-%E6%9C%BA%E5%88%B6%E5%88%86%E6%9E%90.html

http://pdos.csail.mit.edu/~rtm/e1000/

http://www.nuclearcat.com/mediawiki/index.php/Intel_Gigabit_Performance

http://wiki.networksecuritytoolkit.org/nstwiki/index.php/LAN_Ethernet_Maximum_Rates,_Generation,_Capturing_%26_Monitoring#Gigabit_Ethernet_Using_TCP.2FIP

http://sebug.net/paper/databases/nosql/Nosql.html

http://www.julianbrowne.com/article/viewer/brewers-cap-theorem

gonet