I'm OWenT
Challenge Everything

关于BUS通信系统的一些思考（二）

Table Of Content

> 接上文

目录

BUS系统的设计思路

虽然我很不愿意再设计一套BUS系统，但是现有的一些确实都没有特别符合我的口味的。所以还是尝试设计一个出来。

结构设计

简单来说，我希望BUS系统可以简单、高效、稳定。

节点标识

首先，在**节点标识**方面，类似ZeroMQ的用字符串来标识端点的做法我认为是不必要的。这点上可以参照前面两种的设计，但某些情况下32位作类型分割的可能会不够，所以可以使用***64位标识***。其实还有一个重要的原因是64位数字可以整个存放在CPU寄存器里，可以通过一个汇编指令进行比较操作，无论性能还是可以表示的节点个数都很足够。

节点间关系

第二点就是**节点间关系**，我觉得可以设计成***树形结构***，而不是像上面一样的代理节点+数据节点的结构。再考虑BUS通信一般会碰到的几种消息流转方式。

第一中情况是两个节点直连。那么节点里要**记录直连的节点表。**

节点A->节点B:
节点B->节点A:

第二种是需要通过公共父节点转发。

节点A->公共父节点:找不到直连信息，发给父节点
公共父节点->节点B:转发消息

这种情况又有分支，一种是接下来父节点通知两个子节点直接建立连接通道，另一种是不通知，每次都由父节点转发。如果子节点要建立直连通道则如下图所示：

节点A->公共父节点:找不到直连信息，发给父节点
公共父节点->节点B:转发消息
公共父节点->节点A:通知节点A直连到节点B
节点A->节点B:有直连信息，直接发送

按照之前真实系统的设计，节点分为三层的情况下（第一层是集群管理节点；第二层为物理机代理节点；第三层是业务进程节点）。第一层节点再转发的时候会通知第二层节点之间直连，第二层也会通知第三层节点之间互联，所以这里***建议默认开启通知子节点互相直连***的通知。

这里的第二个问题在于需要转发时的消息路由。在没有目标节点信息的情况下，当节点需要发送消息，是直接扔给父节点呢还是直接返回错误呢？这两种方式都各有利弊。一，没有目标节点就发给父节点，这种方式很简单，但是发送节点自身不能立刻感知是否发送成功，如果不成功需要等父节点通知，这一定是一个异步的过程；二，没有目标节点就直接返回错误，这种方式就会导致父节点在收到节点变更的通知后要把整个路由表下发，并且这期间必然存在时延。

前一种需要**错误回执协议**（假设数据不会丢失，那么正确转发的情况下不需要回包），而且这种*错误回执是可选的*。而后一种需要**路由表同步协议**，并且节点内至少要对节点ID做索引。

所以按照这样的设计思路，节点注册到父节点时要报告给父节点自己是否需要全局路由表，并且**当父节点没有全局路由表时，子节点也不能有**（这里不符合条件时最好注册出错）。另外父节点需要维护一个有全局路由表的子节点列表。

在线上实际运行的环境里还碰到一个问题，就是***当有大量需要全局路由表的子节点注册时，只需要把最终结果广播一下***就好了。不需要每次注册上来子节点都广播通知，这样会浪费大量通信流量。这里可以拿一个定时器保护一下。

另一个问题是**节点注册消息的时延问题**，如果两个相同ID的节点出现在两个不同的地方同时注册，那么当他们传递到第一个有全局路由表的节点时可能都是合法的。要解决这个问题方式之一是选举出一个总裁决节点，但是决策这个总裁决节点也存在时延，而如

果要引入Paxos算法又过于复杂了。而且这个问题很容易在配置管理层解决，所以这里倾向于参考*子网掩码*的做法，**给每一个节点设置子节点范围**，注册时节点ID和其子节点可变范围都必须在范围内。比如，节点上报ID为0x1234，子节点可变范围是24位，则0x000000 *1234* **000001**到0x000000 *1234* **FFFFFF** 都是可用的子节点ID（其中0x0000001234000000是这个节点的ID）。

当然还有第三种方式，就是没有公共父节点的情况。

节点A->A的父节点:找不到直连信息，发给父节点
A的父节点-->B的父节点:任何方式都可能
B的父节点->节点B:直接转给节点B

这种情况可以简单暴力了，每次有消息都转给父节点就好了。

通信模式

接下来时通信模式。每个节点都有一个唯一ID，但是通信模式可以有很多。其中最主流的就是网络Socket。BUS系统应该设计成容易拓展或者拆除某些通信模式，所以通信模式必须抽象出来。

另外一点，就是希望BUS系统里，节点的接收端只有一个。这样可以做到地址收敛，简化通信模型。

通信连接的建立

当有多种连接方式的时候，问题之一就是两个节点间应该以什么连接方式互联。

很多BUS系统的通道是读写分离和控制通道和数据通道分离的，这么设计的好处是简单、高效。带来的坏处就是，通道缓冲区浪费和可能某一个通道堵塞但是另一个通道正常。比如，控制通道堵塞，但是数据通道正常，这时候父节点可能回认为子节点下线；又或是控制通道正常，但是数据通道堵塞，这时候父节点回认为子节点正常，但这时候数据是发送不成功的。所以理想情况下**通道还是应该只有一个**，这样也就意味着要有控制协议和包头。

再拿之前的共享内存的例子来说，进程节点只有共享内存通道一种，但是代理节点有socket和共享内存通道两种。节点再注册时，怎么连接到父节点的通信通道和自身的接收通道是子节点决定的。如果我们把通道按优先级分化，假设网络Socket的优先级是3，Unix Socket的优先级是2，共享内存的优先级是1。那么子节点到父节点的通道优先级优先级高的一方连接低一方或者相等的一方一定是通的（当然是不出现故障的情况下）。

双方都是高优先级时：

节点B->父节点:共享内存，优先级1（通路）
父节点->节点A:共享内存，优先级1（通路）
节点B->节点A:共享内存，优先级1（通路）

一方是高优先级时：

节点B->父节点:共享内存，优先级1（通路）
父节点->节点A:网络Socket，优先级3（通路）
节点B->节点A:网络Socket，优先级3（通路）

节点B->父节点:网络Socket，优先级3（通路）
父节点->节点A:共享内存，优先级1（通路）
节点B->节点A:网络Socket，优先级3（如果节点A有网络Socket接收通道，通路）
节点B-->节点A:共享内存，优先级1（不一定通）

双方是低优先级时：

节点B->父节点:网络Socket，优先级3（通路）
父节点->节点A:网络Socket，优先级3（通路）
节点B->节点A:网络Socket，优先级3（通路）
节点B-->节点A:共享内存，优先级1（如果节点A有共享内存通道，不一定通）

所以**子节点上报时还要通知兄弟节点接入的允许的通道类型。**同样，*如果没有允许的连接通道，则父节点也不需要再发兄弟节点直连通知。*

网络Socket

网络Socket就很简单，因为本来就是面向连接的。加一下断线重联什么的就好了。然后linux下加个epoll、windows下加个iocp、其他什么系统价格kqueue什么的就好了。

Unix Socket

这玩意纯属IPC用得，操作和网络Socket一样，不能跨机器通信。

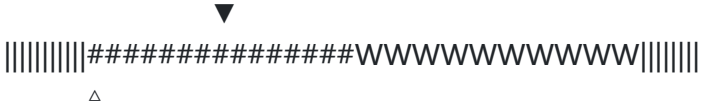
共享内存

使用共享内存最大的困难就是前面的收敛接收端点的需求。对于socket而言，因为一定是面向连接的一对一的所以比较好说。而共享内存要收敛接收端就必须实现至少**多写单读**的共享内存通道。

单读单写的共享内存通道

前面提及的BUS系统有些在内存和共享内存通道上是单读单写的模式。这种方式比较简单，对于一个用于循环队列的内存块。

单读单写需要一个读游标一个写游标。写入操作时，先写入数据。



再移动写游标位置。当然写入前要先判断是否有足够的写入空间，并且不会抵达读游标的位置。

```
                    ▼
||||||||||#########################|||||||
         △
```

而读操作的时候，首先要判断读游标没有到写游标的位置，这样表示有数据。然后读入时，也是先读数据。

```
                    ▼
||||||||||RRRRRR###################|||||||
         △
```

再移动游标位置。

```
                    ▼
||||||||||||||||####################|||||||
              △
```

这种情况，在共享内存里操作甚至不需要原子操作，只要注意去掉寄存器缓存（C/C++里加*volatile*关键字）就可以了。不需要原子操作的原因在于，每个节点只会操作一个游标，并且一个节点只操作空白区域，另一个节点只操作数据区域。

单读多写的共享内存通道

单读多写的通道有助于把收消息节点收敛到一处。而单读多写意味着最大的难点在于单处读取和多处写入不冲突，并且当有节点出错的时候能保证赃数据被跳过。

为了解决上诉问题，可以把内存分为若干个内存块，然后每个内存块有一个信息头，记录了这个块内的标记位(对齐到4字节，便于编译优化)（包含写完标记位、是否起始node标记位和是否有后续节点的标记位）。

另外首节点还需要附加CRC校验码（使用自定义memcpy函数，copy的同时做校验并清0）、数据总长度和第一次尝试读取的时间。

最后整个内存块前端有一个整体head，记录了每个数据node的大小，数据node的个数，原子操作的读游标，写游标，统计信息和一些配置，比如读取时间容忍值（据Google一个文档说内存访问大约每毫秒可以到4MB，所以1毫秒的容忍值已经绰绰有余）。最后对齐到4KB（默认一个分页）用于以后拓展。

对于冲突问题

1. **读-读冲突**：只考虑单点读，没有这个问题。
2. **读-写冲突**：head有写完毕标记位，当写数据块准备完毕时才开始读。
3. **写-写冲突**：写游标是原子操作，每个节点写缓冲区独立。
4. **写进程崩溃**：会产生赃数据块，即写完标记永远是未写完。这时候可以利用上上面提到的第一次读取时间。如果是0，则取当前时间赋值，否则如果超出容忍值，就视为赃数据块。取时间可以使用clock函数（Linux下实测每次执行消耗约160ns），也可以用汇编直接提取CPU时钟。一般情况下系统应该在数百次读取无数据后休眠至少一个时间片的时间(Linux下一般最少有4ms)，这时候写进程还没写完基本可以认为是出现赃数据。

5. **读进程崩溃**：移动读游标是最后的操作，下次启动时可以继续，不会丢失数据。

2014/11/07 实际实现过程中发现共享通道时的读-写冲突和写-写冲突是不能完全避免的，另外多进程结构下的原子操作也很难保证强一致。所以在代码中增加了校验和自动重试。最终实现的代码中多进程发消息时，消息丢失率在三亿分之一左右。我觉得属于可接受范围，以后有时间可以抠细节去优化调整它。

多读多写的共享内存通道

这个可以作为以后拓展项。不是基本功能可以暂不提供。

共享内存消息通知

以上通道完成以后，共享内存消息还只能通过轮询得知是否有数据。对于每个线程只有一个节点的情况还比较好说，但是节点个数多了以后就有必要仿照epoll提供通知机制。常见的解决方案有：

1. fifo
2. eventfd/signalfd
3. socket 各有利弊吧。无论使用哪种，都需要注意的是*通知的性能会远低于数据收发*，所以每次通知需要尝试读完通道里的消息；另外要注意重复通知的问题。

消息通知也不是核心功能，早期也可以不提供。

其他杂项

序列化、反序列化和包头

为了减少内存消耗，需要在包头里对表示整形的数据做一些**压缩处理**。

比较成熟的解决方案有很多，比如Protobuf的(varint)[]。每个字节的第一个bit用于表示后一个字节是不是这个varint，其他7个bit表示数据。

还有Cap’ s Proto的压缩方案。简化为int型压缩可以总结为：第一个字节作为bitmap表示后面8个非0字节的位置，然后后面依次跟非0字节的内容；如果第一个字节是0x00表示数字0；第一个字节是0xFF则第10个字节再按上面的流程走一次。

另外还有MessagePack的int压缩方案。第一个字节描述类型，后面跟数据。

比较上面几种压缩方式，Protobuf和Cap’ s Proto都*支持不限长度的int*，但是由于系统里的Cap’ s Proto的方案，只要不是0，就至少需要两字节。只有当数字大于7个字节时，protobuf的长度才会比Cap’ s Proto长；再考虑到bus通信中的消息一般不会很大，一个字节的protobuf的表示范围是[0, 128)，涵盖了大部分消息包长度，两个字节是[0, 16K)基本涵盖所有消息包长度。所以这里推崇使用***Protobuf的方案***。

在包头方面，**第一层包头**一般在共享内存或者socket表示长度上，共享内存的数据都在同一台机器上，进程间系统架构一致，所以为了简单、高效，**共享内存包头**直接上*裸内存*数据即可。如果在socket上通信则可以选择用前面提到的**varint**。

第二层包头用于区分控制指令和消息转发。这部分建议用Flatbuffers来打解包。当然也可以选择Protobuf或者Cap’ s Proto或者MessagePack。推荐Flatbuffers的原因是**简**

单高效。但是要注意一点就是Flatbuffers依赖比较高版本的编译器，而使用Cap’s Proto必须保证通行的机器之间的架构一致（这一点再服务端比较容易达成）。

第三部分一般来说这里直接就是数据区了。但是也可以增加一些拓展功能包头，比如可以拿zlib或者Snappy做数据压缩，再或者拿openssl、boringssl或者Crypto++来做加解密。有点像IPv6的扩展包头。

最后

这些想法最终我会尝试一个实现放在github上，实现过程中可能会碰到一些问题会导致这些想法的细微变更。届时会同步更新到blog里。并且在单元测试和压力测试通过后会把地址更新到这里。

GitHub地址: https://github.com/atframework/libatbus

> Written with StackEdit.

Article/ Blablabla *2014-08-05*

bus ipc 无锁队列 消息队列 进程间通信

上一篇**LLVM + Clang + Libcxx + Libcxxabi 工具链编译**

下一篇**关于BUS通信系统的一些思考（一）**

在 OWENT 上还有

pbc的一个陈年老BUG

5 年前 • 4条评论

协程框架(libcopp) 小幅优化

3 年前 • 2条评论

最近抽空继续对 libcopp 进行了更新和小幅优化。 首先的Merge了 boost.context …

VC和GCC内成员函数指针实现的研究(一)

5 年前 • 2条评论

朱元 • 5 年前

看了你的这篇文章后了解到flat_buffer等类似库的设计思想 自己写了一个C++03就可用，无需单独撰写IDL的库。
协议直接写在.h头文件里就行，直接支持C++POD原生struct 原生union(可以自己做一些压缩) 前向声明类型等。
收包即用，无需解包，支持从任意一个节点开始进行数据安全检查，能检查缓冲区偏移量上下溢出，数据不在正确的对齐位置上等非法数据。字符串成员可以用c_str命令来保证获取到结尾是0的合法数据。
encoding比flat_buffer爽利，支持输出json流（但不支持从json导入，人力有限。。）。
支持针对小型包对偏移类型，数组长度类型进行自定义类型的存储优化。所有类型都支持重新赋值（但不鼓励）。字段预分配内存大小会根据根据协议的最坏情况进行计算。

https://github.com/yuanzhub