白杨应用支撑平台

HAC Manager 用户手册

Version 1.14

版本控制

版本号	修改时间	修改内容	修改人	审阅者
1.0	2016-05-30	创建	白杨	
1.1	2018-02-14	调整部分 MSP API 的语义；一些次要更新	白杨	
1.2	2018-05-11	新增“Emergency Stop”命令支持	白杨	
1.3	2018-12-06	为端口注册 API 新增“URN_ALL”选项支持	白杨	
1.4	2018-12-14	新增批量端口注销、批量消息发送和批量消息组播相关 API	白杨	
1.5	2019-07-13	一些次要更新	白杨	
1.6	2019-07-19	新增“Allow Slave Online”选项支持	白杨	
1.7	2019-07-30	新增“Before Normal Stop”命令支持	白杨	
1.8	2019-08-26	新增“After Start”命令支持	白杨	
1.9	2019-10-18	将“On Fatal”和“After Failback”命令调整为阻塞执行	白杨	
1.10	2021-01-04	进一步澄清各回调命令的触发机制的相关细节	白杨	
1.11	2021-04-08	新增 BYDMQ 支持	白杨	
1.12	2021-04-18	新增 GetStatus API	白杨	
1.13	2022-02-02	澄清一些细节	白杨	
1.14	2023-03-28	新增毫秒级精度支持	白杨	

目录

1. 概述

BYPSS 是一种强一致、高可用、高性能、大容量的分布式协调专利算法。BYPSS 提供了与传统 Paxos/Raft 算法完全相同的一致性和可用性保证，并在此基础上实现了的性能、容量、响应速度、网络利用率等各方面的显著提升。

HAC Manager 作为《白杨应用支撑平台》BYPSS 组件中的配套工具，利用 BYPSS 提供的强一致、高可用和高性能的分布式协调组件，实现了对任意第三方应用（托管服务）的服务选举、故障检测、故障转移和故障恢复等功能：对于竞选相同服务的多个 HAC Manager 实例来说，竞选成功（取得所有权）的节点可以执行用户事先配置的指令来启动相应的托管服务（Service/Daemon）；相应地，在失去所有权时，HAC Manager 则会自动停止该托管服务。

HAC Manager 支持对等（多主）和主从（Master / Slave）两种集群模式。在主从模式下，集群检测到 Master 节点故障下线时，会启动 Slave 节点上的托管服务来接替其工作（Failover）；而当 Master 节点恢复上线后，可配置 Slave 节点是否要将该托管服务的执行权限重新交换给 Master（Failback）。

与此同时，Slave 节点还支持竞选避让机制，可确保在发生网络抖动时尽可能由主节点重新取得托管服务的所有权。避免由于环境微量抖动引起的频繁切换。

配合 DRBD、HAST、DataKeeper、DFS、Ceph、GlusterFS、Lustre 等分布式存储技术或 SAN 等共享型存储方案，HAC Manager 的主从模式可以在无需任何修改的前提下，方便地将一个传统的单机服务（如：传统 SQL 数据库、全文搜索引擎、报表生成服务、用户业务逻辑等）转化为强一致、抗脑裂（Split Brain）的高可用多活 IDC 集群（HAC）。

与主从模式不同，HAC Manager 的对等模式除了可实现强一致的高可用集群（HAC）以外，还能够同时支持分布式高性能计算（HPC）等场景。不过对等模式需要对第三方托管服务进行少量修改才能正常开启。实现方式为：托管服务通过本地回环地址（127.0.0.1）与 HAC Manager 进行通信，在 HAC Manager 的辅助下使用 BYPSS 服务完成服务发现、服务选举、故障检测、故障转移、故障恢复、分布式锁、任务调度，以及消息路由和消息分发等分布式协调作业。此时 HAC Manager 扮演的则是服务网格（Service Mesh）中的 Local Agent 角色（sidecar 模式）。

注：以上所述 BYPSS 算法受我方拥有的国家和国际发明专利保护。

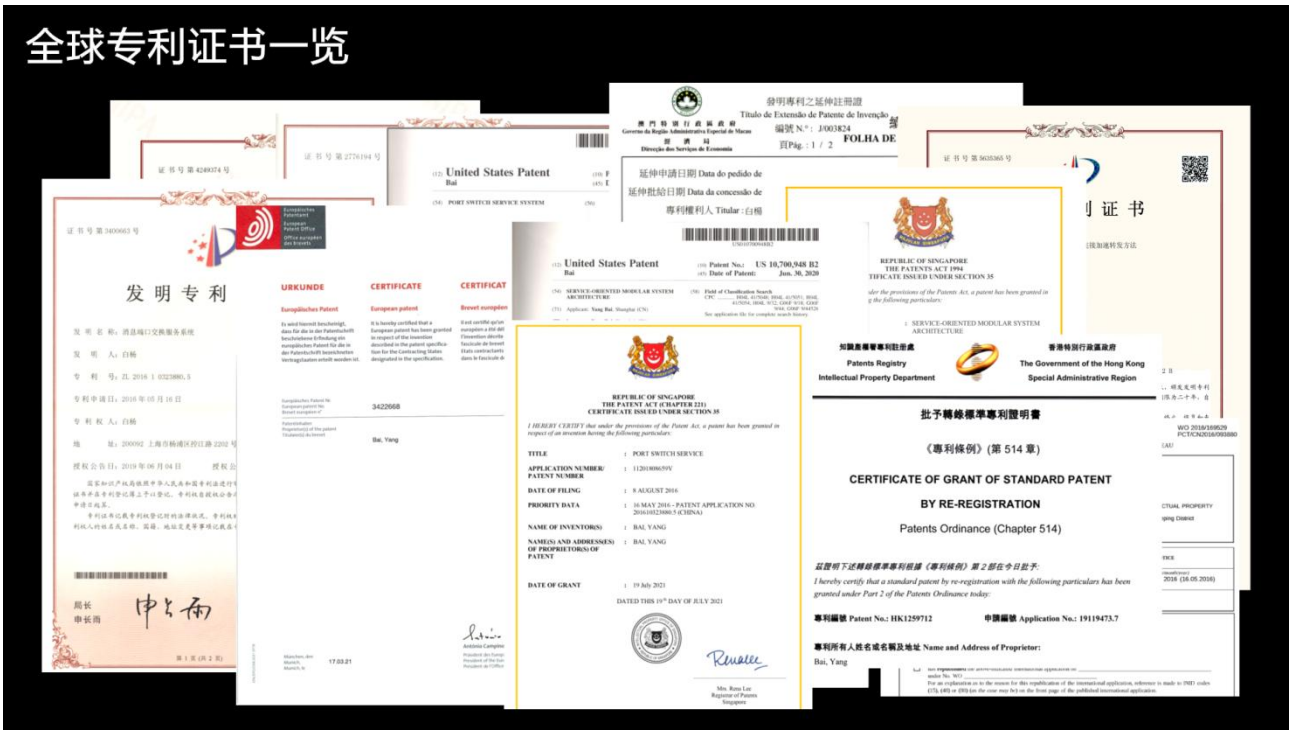
1.1 应用支撑平台

《白杨应用支撑平台》作为我方的核心竞争力，使用汇编、C/C++ 构建，包含数百万行代码和上千个成熟的通用组件。上述单点高并发和多活 IDC 分布式集群等核心技术均由白杨应用支撑

平台提供。多年来，基于支撑平台的各种产品已被广泛部署于包括兴业银行（China CIB）、中石油（CNPC）、华安保险（Sinosafe Insurance）、淘宝网（taobao.com）、易果网（yiguo.com）、烟台万华集团、法国兴业银行（SOCIETE GENERALE）、中国农业银行（ABC）、宝马集团（BMW）、陕汽集团（陕汽重卡）、德尔福汽车（Delphi）、美联航（United Airlines）、GE（美国通用电气）、贝塔斯曼（Bertelsmann）、埃森哲（Accenture）、光大银行（CEB）、壹基金（One Foundation）、中国宋庆龄基金会、中国联通（China Unicom）、中国移动（China Mobile）、国家电网（SGCC）等各大企业在内的不同生产环境中：

真实生产环境下的大范围部署不但为上层应用提供了可靠的、平台无关的底层环境，也进一步检验了支撑平台的可靠性、稳定性、可移植性、高效性等各方面指标。

支撑平台使用汇编、C/C++ 构建，支持 Windows、Linux、BSD、IBM AIX、HP-UX、Solaris、MAC OS X、vxWorks、QNX、DOS, WinCE (Windows Mobile), NanoGUI、eCos、RTEMS 以及 Android、iOS 等绝大多数主流操作系统。支持 x86/x64、ARM、IA64、MIPS、RISC-V、POWER、SPARC 等主流硬件平台。在提供了大量高品质可重用组件的同时，也确保了良好的可移植性。



上千成熟可靠的高品质功能组件可在性能、功能和稳定性等方面大幅提升软件产品的品质，并为产品的开发带来了难以想象的便利，例如：

1.1.1 单点支撑千万量级并发的高效 IO 服务器组件

支撑平台使用汇编和异步 IO 对网络服务组件进行了优化，通过 DMA + 硬件中断实现内存零拷贝的高效异步网络服务。性能、可靠性和可伸缩性都很强。可在 2011 年出厂的，当时售价不足 2 万元人民币的单台至强 5600 系入门级 1U PC Server 上支撑上千万 TCP/HTTP 并发连接。相对应地，一般由 Java / .NET 开发的服务器端，在相同配置的机器上，单点最高仅可支撑 3000 到 5000 并发，PHP 则更低（具体可参考《白杨应用支撑平台技术白皮书》：3.2.1、3.3.1 以及 3.3.2 小节）。

1.1.2 强一致多活 IDC 高可用（HAC）和高性能（HPC）服务器集群

强一致、抗脑裂（Split Brain）的多活 IDC 分布式高可用（HAC）和高性能（HPC）计算集群支持：独有的 nano-SOA 大规模分布式架构在保持高内聚、低耦合设计的前提下，将单点性能提升到了远超传统 SOA 架构的水平，同时简化了集群部署，提高了集群的可维护性。

白杨消息端口交换服务（BYPSS）：一种基于多数派算法的，强一致（抗脑裂）、高可用的分布式协调组件，可用于向集群提供服务发现、故障检测、服务选举、分布式锁等传统分布式协调服务，同时还支持消息分发与路由等消息中间件功能。由于通过专利算法消除了传统 Paxos/Raft

中的网络广播和磁盘 IO 等主要开销，再加上批量模式支持、并发散列表、高并发服务组件等大量其它优化，使得 BYPSS 可在延迟和吞吐均受限的跨 IDC 网络环境中支持百万节点、万亿端口量级的超大规模计算集群。

带强一致保证的多活 IDC 技术是现代高性能和高可用集群的关键技术，也是业界公认的主要难点。作为实例：2018 年 9 月 4 日微软美国中南区某数据中心空调故障导致 Office、Active Directory、Visual Studio 等服务下线近 10 小时不可用；2015 年 8 月 20 日 Google GCE 服务中断 12 小时并永久丢失部分数据；2015 年 5 月 27 日、2016 年 7 月 22 日以及 2910 年 12 月 5 日等支付宝数次中断数小时；以及 2013 年 7 月 22 日、2023 年 3 月 29 日微信服务中断数小时等重大事故均属于产品未能实现多活 IDC 架构，单个 IDC 故障导致服务全面下线的惨痛案例。

在上述方面，我方拥有超过十年的积累，掌握多项受国家和国际发明专利保护的分布式架构和算法。得益于这些领先的强一致、高可用、高性能分布式集群算法和架构，我们在蓝鲸、白豚、职业精等全线产品上，均实现了真正的多活 IDC 架构，为客户提供了无以伦比的数据可靠性和服务可用性保证。

1.1.2.1 分布式协调服务

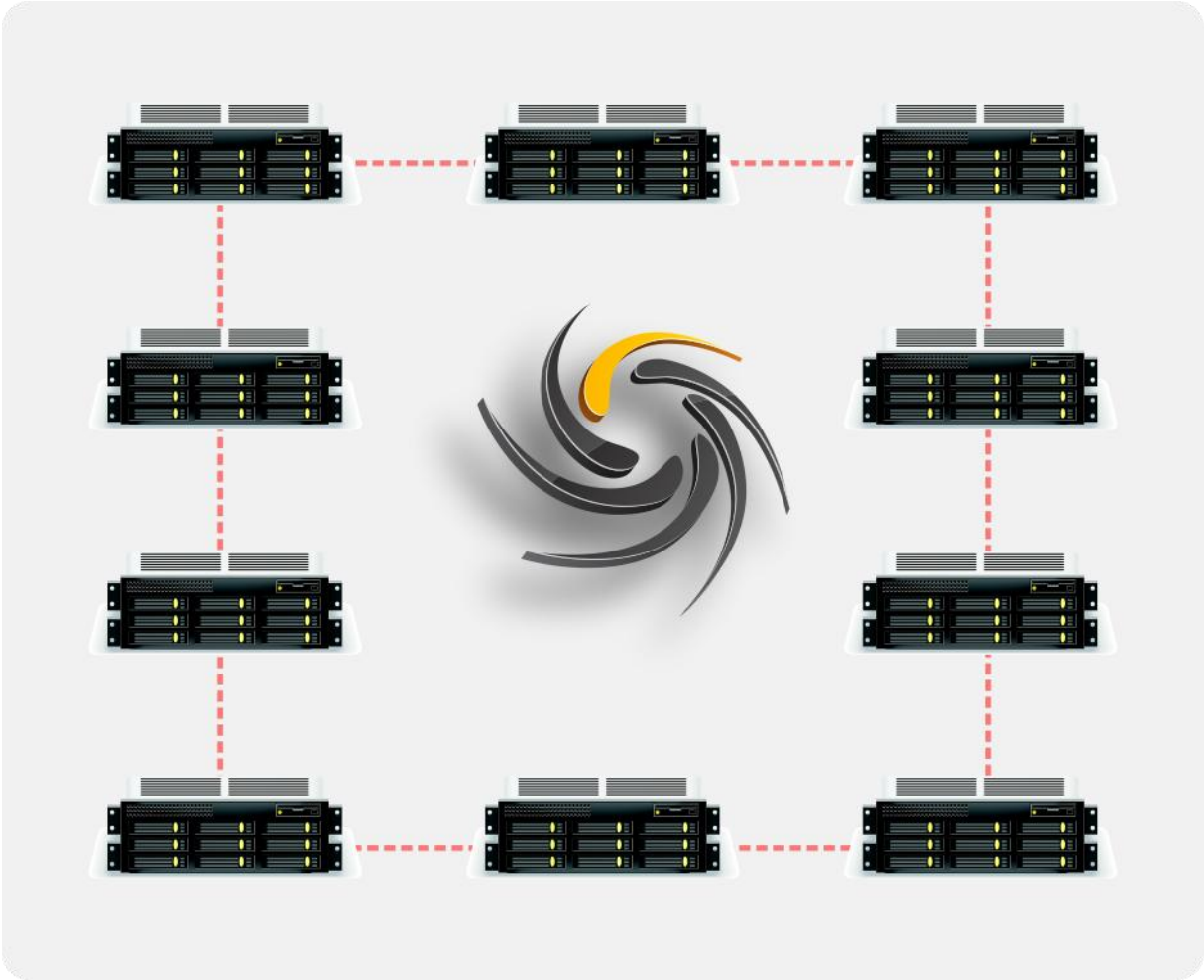


图 1

分布式协调服务为集群提供服务发现、服务选举、故障检测、故障转移、故障恢复、分布式锁、任务调度，以及消息路由和消息分发等功能。

分布式协调服务是分布式集群的大脑，负责指挥集群中的所有服务器节点协同工作。将分布式集群协调为一个有机整体，使其有效且一致地运转。实现可线性横向扩展的高性能（HPC）和高可用（HAC）分布式集群系统。

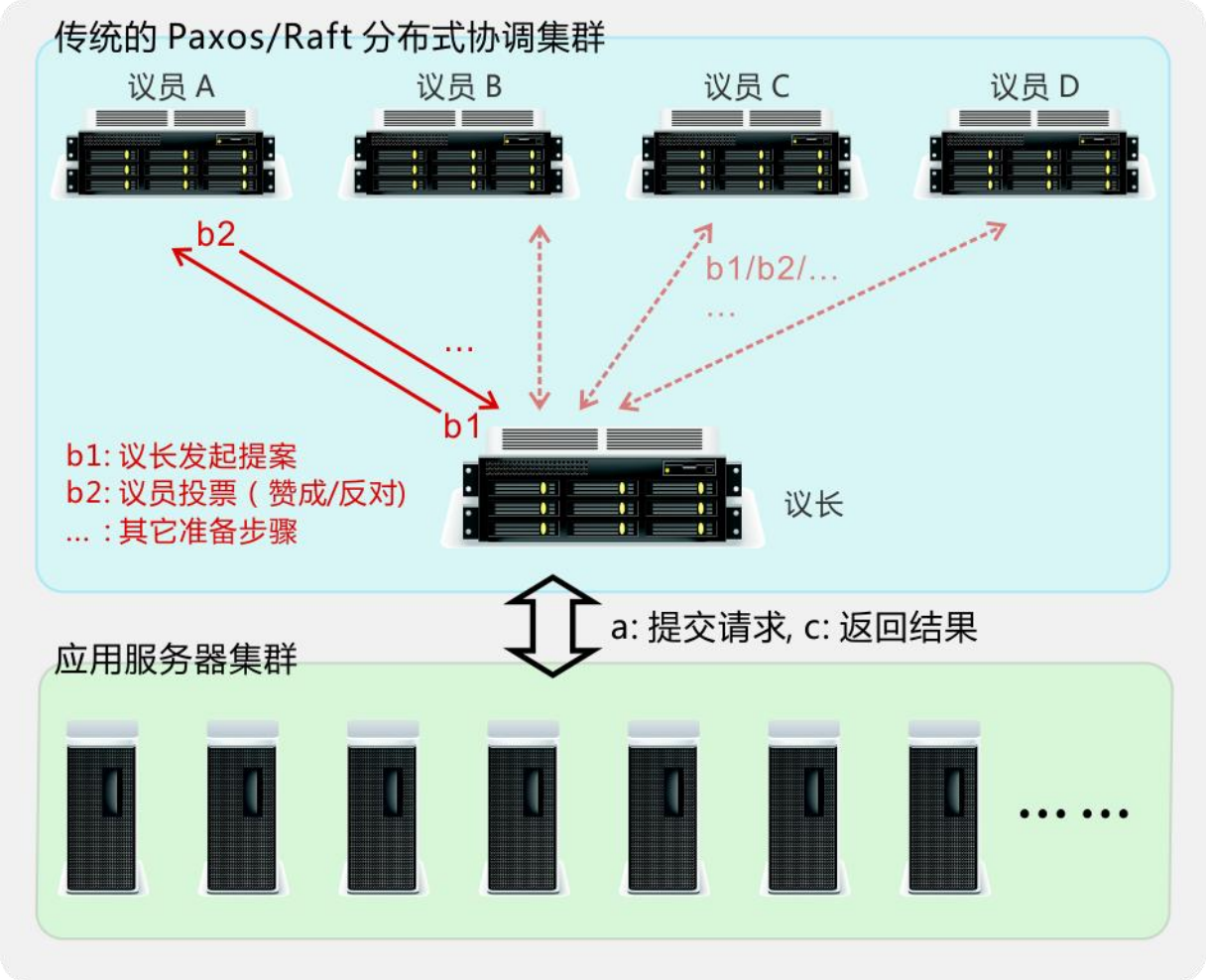


图 2

传统的 Paxos/Raft 分布式协调算法为每个请求发起投票，产生至少 2 到 4 次网络广播（b1、b2、…）和多次磁盘 IO。使其对网络吞吐和通信时延要求很高，无法部署在跨 IDC（城域网）环境。

我们的专利算法则完全消除了此类开销。因此大大降低了网络负载，显著提升整体效率。并使得集群跨 IDC 部署（多活 IDC）变得简单可行。

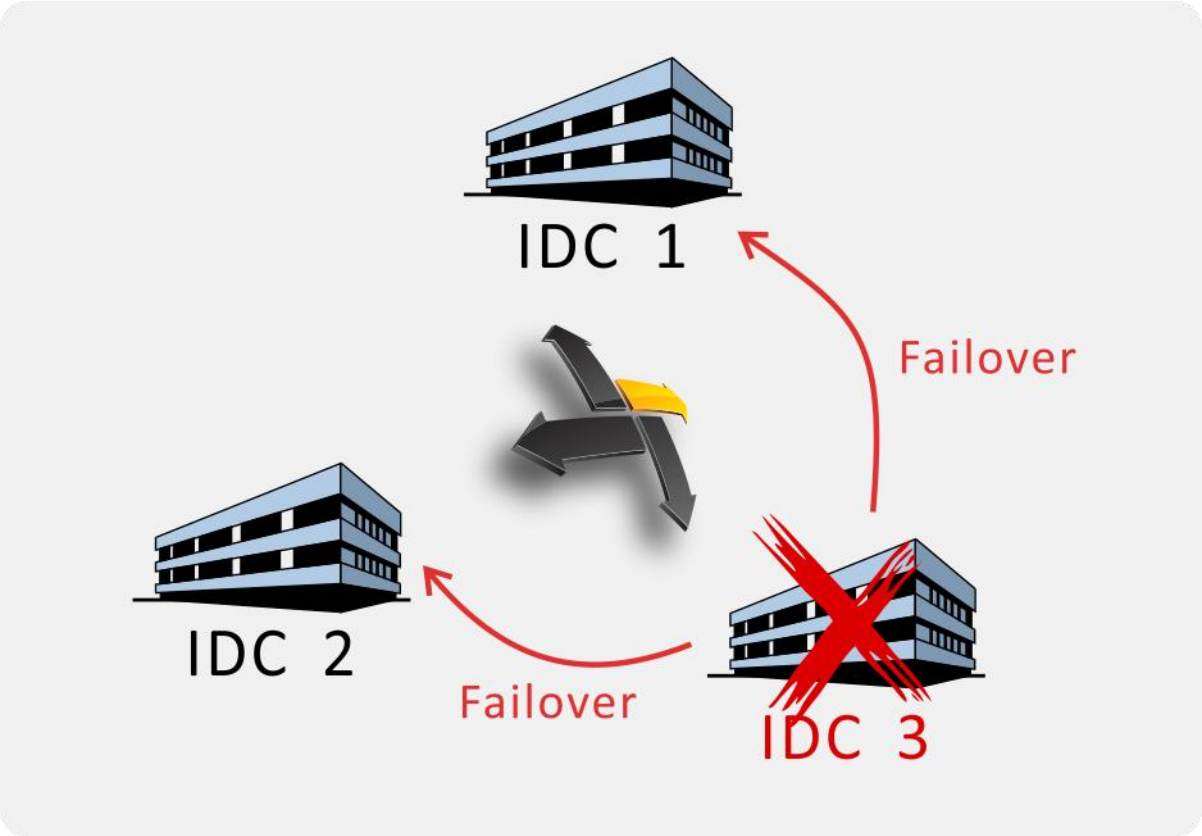


图 3

基于我方独有的分布式协调技术，可实现高性能、强一致的多活 IDC 机制。可在毫秒级完成故障检测和故障转移，即使整座 IDC 机房下线，也不会导致系统不可用。同时提供强一致性保证：即使发生了网络分区也不会出现脑裂（Split Brain）等数据不一致的情形。例如：

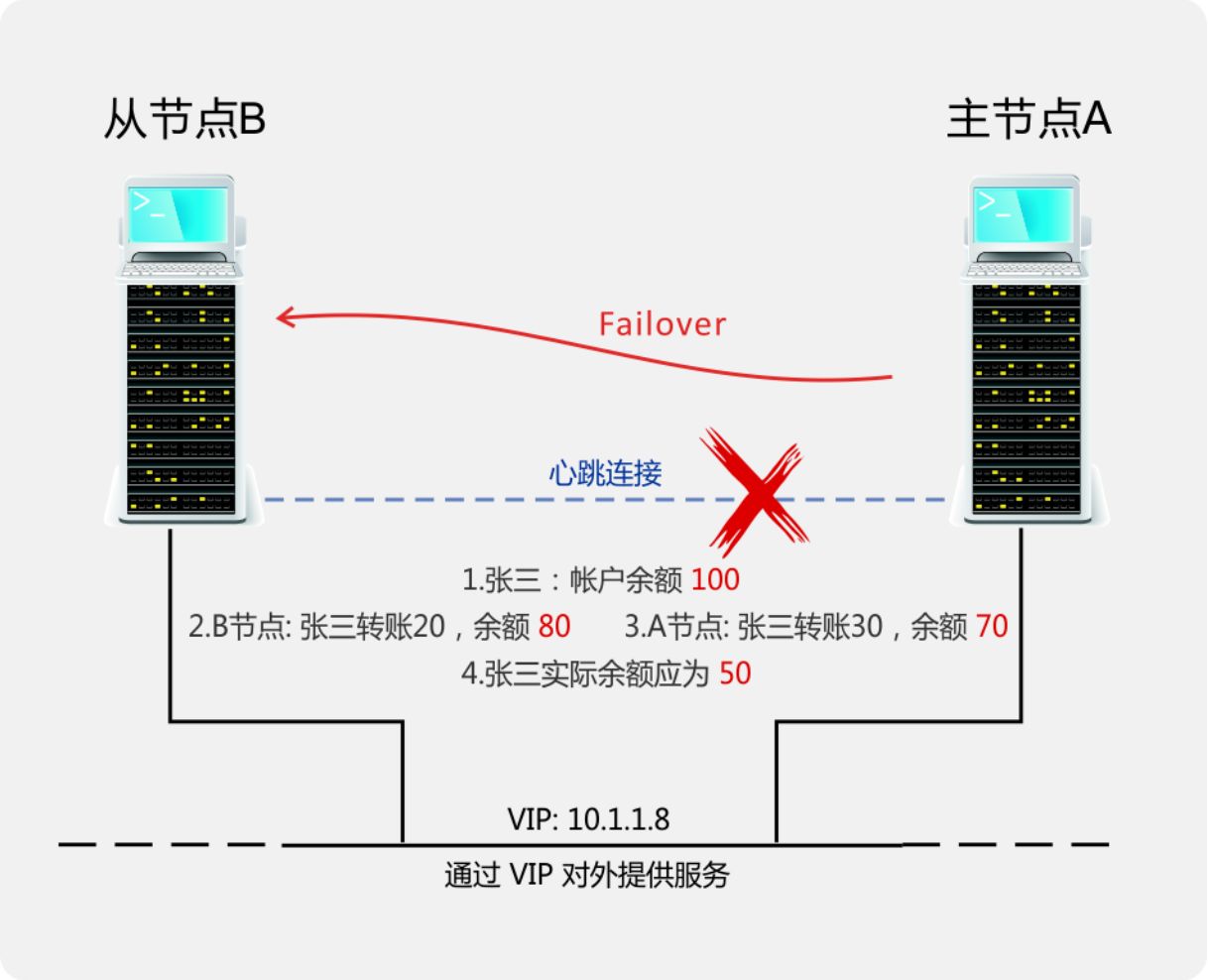


图 4

在传统的双机容错方案中，从节点在丢失主节点心跳信号后，会自动将自身提升为主节点，并继续对外提供服务，以实现高可用。在此种情形中，当主从节点均正常，但心跳连接意外断开时（网络分区），就会发生脑裂（Split Brain）问题，如图 4 所示：此时 A、B 均认为对方已下线，故将自己提升为主节点并分别对外提供服务，产生难以恢复的数据不一致。

我方 BYPSS 服务可提供与传统 Paxos/Raft 分布式算法相同水平的强一致性保证，从根本上杜绝脑裂问题的发生。

类似地：工行、支付宝等服务也有异地容灾方案（支付宝：杭州 → 深圳、工行：上海 → 北京）。但在其异地容灾方案中，两座 IDC 之间并无 Paxos 等分布式协调算法保护，因此无法实现强一致，也无法避免脑裂。

举例来说，一个在支付宝成功完成的转账交易，可能要数分钟甚至数小时后才会从杭州主 IDC 被异步地同步到深圳的灾备中心。杭州主 IDC 发生故障后，若切换到灾备中心，意味着这些未同步的交易全部丢失，并伴随大量的不一致。比如：商家明明收到支付宝已收款提示，并且在淘宝交易系统看到买家已付款，并因此发货。但由于灾备中心切换带来的支付宝交易记录丢失，导致在支付宝中丢失了相应的收入，但淘宝仍然提示买家已付款。因此，工行、支付宝等机构在主 IDC

发生重大事故时，宁可停止服务几个小时甚至更久，也不愿意将服务切换到灾备中心。只有在主 IDC 发生大火等毁灭级事故后，运营商才会考虑将业务切换到灾备中心（这也是灾备中心建立的意义所在）。

因此，异地容灾与我方的强一致、高可用、抗脑裂多活 IDC 方案具有本质区别。

由于消除了 Paxos/Raft 算法中的大量广播和分布式磁盘 IO 等高开销环节，配合支撑平台中的高并发网络服务器、以及并发散列表等组件。使得 BYPSS 分布式协调组件除了上述优势外，还提供了更多优秀特性：

批量操作：允许在每个网络包中，同时包含大量分布式协调请求。网络利用率极大提高，从之前的不足 5% 提升到超过 99%。类似于一趟高铁每次只运送一位乘客，与每班次均坐满乘客之间的区别。实际测试中，在单千兆网卡上，可实现 400 万次请求每秒的性能。在当前 IDC 主流的双口万兆网卡配置上，可实现 8000 万次请求每秒的吞吐。比起受到大量磁盘 IO 和网络广播限制，性能通常不到 200 次请求每秒的 Paxos/Raft 集群，有巨大提升。

超大容量：通常每 10GB 内存可支持至少 1 亿端口。在一台插满 64 根 DIMM 槽的 1U 尺寸入门级 PC Server 上（8TB），可同时支撑至少 800 亿对象的协调工作；在一台 32U 大型 PC Server 上（96TB），可同时支撑约 1 万亿对象的分布式协调工作。相对地，传统 Paxos/Raft 算法由于其各方面限制，通常只能有效管理和调度数十万对象。

问题的本质在于 Paxos / Raft 等算法中，超过 99.99%的代价都消耗在了网络广播（投票）和落盘等行为上。而这些行为的目的就是要保证数据的可靠性（数据要同时存储在多数节点的持久化设备上）。而服务发现、服务选举、故障检测、故障转移、故障恢复、分布式锁、任务调度等分布式协调功能所涉及到的恰恰又都是没有长期保存价值的临时性数据。因此花费超过 99.99%的精力来持久化地保存它们的多个副本是毫无意义的——就算真的发生主节点下线等罕见灾难，我们也可以极高的效率，在瞬间就重新生成这些数据。

就好像张三买了一辆车，这辆车有个附加保险服务，其条款为：在张三万一发生了致命交通意外时，它能提供一种时光倒流机制，将其带回到意外发生之前的一瞬间来避免这场意外的发生。当然，这么牛的服务肯定也很贵，它大概需要预付张三家族在接下来的三生三世里能获得的所有财富。而且即使张三在驾驶这辆车的过程中，始终未发生过致命交通事故，那这些预先支付的服务费也是一分钱都不能减免的。这么昂贵的服务，且不说一般人一生中大概率都不会发生致命交通事故（更别提还要指定具体的某辆车）。即使真发生了，这个三代赤贫的代价也难说就值得吧？

而我们则为自己的汽车产品提供了另一种不同的附加服务：虽然没有时光倒流功能，但我们的服务可以在张三发生致命事故后，将所有受害方全体连车带人瞬间原地满血复活（是一根头发丝都不会少、一块漆皮都不会掉那种满血）。最关键的是，该服务无需预先收取任何费用。张三只需要在每次这样的灾难发生以后，支付相当于其半个月的工资的再生技术服务费就可以了。

综上，我方专利的分布式协调算法，在提供与传统 Paxos/Raft 算法相同等级的强一致性和高可用性保证之同时，极大地降低了系统对网络和磁盘 IO 的依赖，并显著提升了系统整体性能和

容量。对于大规模、强一致分布式集群的可用性（HAC）和性能（HPC）等指标均有显著提升。

关于 BYPSS 服务的进一步描述，详见：2.3 消息端口交换服务。

1.1.3 高效、高强度的密码编码学组件

包含公钥算法、对称加密算法、数据编解码、散列和消息验证算法、数据压缩算法等基础功能组件。除此之外，支撑平台还提供了多个经过高度抽象、可开箱即用的高级密码编码学功能组件，例如：

支持实时压缩和强加密的虚拟文件系统（VFS），VFS 支持包括 AES（128/256）、TwoFish 等在内的数十种强加密算法，使用 AES-NI、SSE4 等汇编指令集优化，效率高。在蓝鲸、白豚、职业精等全线产品中，我们均使用该组件为产品的数据库和配置类数据提供整库级的实时压缩和强加密保护。另外还包括基于公钥体系架构（PKI）的强加密通信保护组件等。

近年来安全问题频发，亚马逊、沃尔玛、Yahoo、Linkedin（领英）、OpenAI（ChatGPT）、索尼、摩根大通、UPS、eBay、京东、支付宝、一号店、协程、12306、网易、CSDN、中国人寿、以及各大酒店集团（如家、汉庭、锦江、洲际、喜来登、万豪等）等国内外知名企业均频频报出大量用户信息泄露的严重安全事件，安全保障已经刻不容缓。

我方所有数据库（整库）和本地配置数据均存放在我方自主研发的，支持实时（on-the-fly）数据压缩和强加密的虚拟文件系统（VFS）中进行全方位保护。支持数十种业界公认的强加密安全算法，即使系统管理员也无法窥视企业数据。

除此之外，我方独有的高性能网络安全隧道（BYST）组件可在保证通信安全的同时，为用户提供高性能、高吞吐和高网络利用率的 VPN 服务，在局域、城域以及广域网络中进一步帮助用户提升网络通信的性能和安全性。

基于业界标准的强加密算法保证了即使未来出现了每秒钟可完成一千万亿次密钥破解尝试的超级计算机，也需要平均五千四百万亿年才能破解一个密钥。安全性得到了极大保证（具体可参考《白杨应用支撑平台技术白皮书》中的第 4 节）。

1.1.4 数据查询分析引擎

支撑平台中还包含了表达力优于 SQL 的查询分析引擎，自有查询引擎除了可以摆脱对特定 DBMS 的依赖，使我们的产品可以自由地在 MySQL、MS SQL Server、Oracle、DB2、SQLite 等 RDBMS 以及 MongoDB、Cassandra 等 NoSQL 数据库间灵活切换。更增加了基于 UNICODE 字符集的 ARE 高级正则查询、表中套表的关联查询、复杂虚拟字段等 SQL 没有的高级特性。

查询引擎通过汇编优化的 C/C++ 代码实现词法分析、语法分析、语义分析/中间代码生成、

优化等步骤，并可在 2010 出厂的 Thinkpad W510（4 核 8 线程，主频 1.6G）笔记本上，仅用其中一核一线程即可达到每秒 1300 万次以上的表达式求值效率。

1.1.5 更多...

我们并不依靠“商业秘密”来保护核心竞争力。相反，我们使用更公开透明的商标、认证、版权、专利和公证保管等手段来保护自己的合法权益。因此，我们的技术细节均公开于相应的文档中，详情可见《白杨应用支撑平台技术白皮书》：

http://baiy.cn/doc/asp_whitepaper.pdf
或者：http://baiy.cn/doc/asp_whitepaper_en.pdf（英文版）

等文档，包括 Hacker News（全球最大的计算机科学新闻网站）、Google Blogger、CSDN 以及博客园在内的多家国内外媒体均转载或报道了这些论文。相较于“严守秘密”，我们相信公开透明下的大量同行审评，加上实际生产环境下的严酷考验，更有利于产品品质的提升。

2. 术语和背景知识回顾

2.1 术语表

术语	解释
CConfig	配置数据格式（见下文）
DWORD	整形（双字，32bit）
QWORD	整形（四字，64bit）
FLOAT	IEEE754 双精度浮点（64bit）
STRING（ASCII）	ASCII 编码的字符串 [\x00-\x7F]
STRING（UTF-8）	UTF-8 编码的字符串 [\u0000-\uFFEF]，在 Query String、Header Field 等位置使用时通常需要使用 url encoding（RFC3986）规则进行换码
STRING（HEX）	从整数或二进制内容转换而来的 HEX 编码字符串 [0-9,A-F]
STRING（DIGITAL）	从整数转换而来的十进制字符串 [0-9]
BLOB	BLock of Bytes，二进制数据块
CSV	点号分割格式，详见 RFC4180
INI	常用配置文件格式。详见：http://en.wikipedia.org/wiki/INI_file
JSON	JavaScript 数据结构定义格式，详见 RFC4627
XML	可扩展标记语言，参见：http://www.w3.org/TR/REC-xml/
托管服务	交由 HAC Manager 控制的第三方应用

2.2 MSP（Message Session Protocol）协议

在对等（多主）模式下，托管服务需要与本机的 HAC Manager 通过 MSP 协议进行信令交互。MSP 是白杨应用支撑平台中定义的一种基于消息的最简二进制会话协议。MSP 基于 TCP 协议实现，其包格式如下图所示：

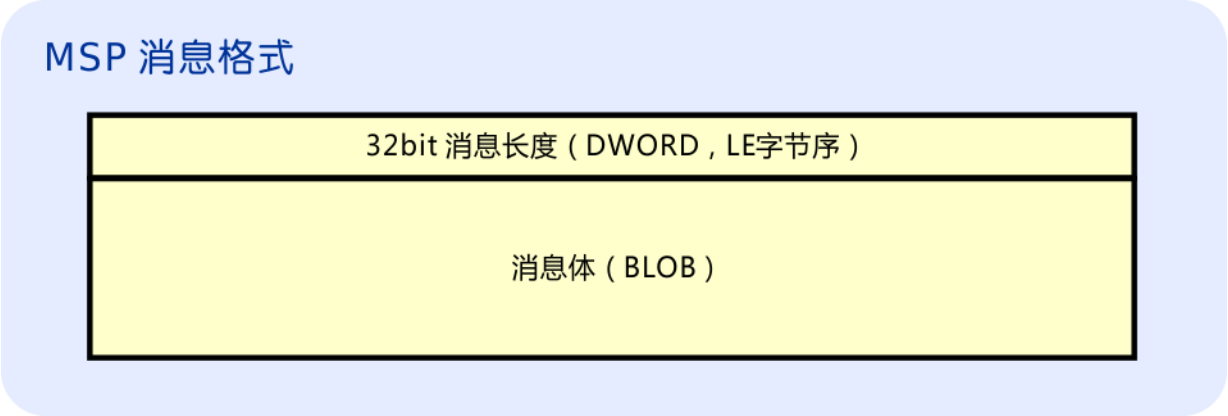


图 5

由图 5 可见，一个完整的 MSP 消息被分为包头和消息体两个部分。其中消息体的内容对协议透明，可以是任意二进制数据（BLOB），而包头中则仅包含一个 LE 字节序的 32bit 无符号整数，用以说明消息体的字节尺寸。

例如，发送一条内容为“Hello”的 MSP 消息，则其 TCP 封包内容的长度为 9 字节。其中前 4 个字节为 LE 字节序的 0x00000005，代表其消息体的字节长度为 5。而后 5 个字节则是内容为“Hello”的字符串。

注意：在主从模式下，托管服务也可以使用此协议与 HAC Manager 进行交互，但不同于对等模式，主从模式下此功能为可选项，并无强制要求。

2.3 消息端口交换服务

白杨消息端口交换服务（BYPSS）设计用于单点支撑万亿量级端口、百万量级节点规模，每秒处理千万至十亿量级消息的高可用、强一致、高性能分布式协调和消息交换服务。其中关键概念包括：

* 连接（Connection）：每个客户端（应用集群中的服务器）节点至少与端口交换服务保持一个 TCP 长连接。

* 端口（Port）：每个连接上可以注册任意多个消息端口，消息端口由一个 UTF-8 字符串描述，必须在全局范围内唯一，若其它客户端节点已注册了相同的消息端口，则端口注册失败。

端口交换服务对外提供的 API 原语包括：

* 等待消息（WaitMsg）：客户端集群中的每个节点均应保持一个到端口交换服务的 TCP 长连接，并调用此方法等待消息推送。此方法将当前客户端连接由消息发送连接升级为消息接收连接。

每个节点号只能对应一个消息接收连接，若一个节点尝试同时发起两个消息接收连接，则较早的那个接收连接将被关闭，并且绑定到当前节点上的所有端口都将被注销。

* 续租（Relet）：若端口交换服务在指定的间隔内未收到来自某个消息接收连接的续租请求，则判定该节点已经下线，并释放所有属于该节点的端口。续租操作用来周期性地向端口交换服务提供心跳信号。

* 注册端口（RegPort）：连接成功建立后，客户端应向端口交换服务注册所有属于当前节点的消息端口。可以在一个端口注册请求中包含任意多个待注册端口，端口交换服务会返回所有注册失败（已被占用）的端口列表。调用者可以选择是否需要为注册失败的端口订阅端口注销通知。

  需要注意的是，每次调用 WaitMsg 重建消息接收连接后，都需要重新注册当前节点上的所有端口。

* 注销端口（UnRegPort）：注销数据当前节点的端口，可一次提交多个端口，执行批量注销。

* 消息发送（SendMsg）：向指定的端口发送消息（BLOB），消息格式对交换服务透明。若指定的端口为空串，则向端口交换服务上的所有节点广播此消息；亦可同时指定多个接收端口，实现消息组播。若指定的端口不存在，则安静地丢弃该消息。客户端可在一次请求中包含多个消息发送命令，主动执行批量发送，服务器端也会将发往同一节点的消息自动打包，实现消息批量推送。

* 端口查询（QueryPort）：查询当前占用着指定端口的节点号，及其 IP 地址。此操作主要用于实现带故障检测的服务发现，消息投递时已自动执行了相应操作，故无需使用此方法。可在同一请求中包含多个端口查询命令，执行批量查询。

* 节点查询（QueryNode）：查询指定节点的 IP 地址等信息。此操作主要用于实现带故障检测的节点解析。可以一次提交多个节点，实现批量查询。

端口交换服务的客户端连接分为以下两类：

* **消息接收连接（1:1）**：接收连接使用 WaitMsg 方法完成节点注册并等待消息推送，同时通过 Relet 接口保持属于该节点的所有端口被持续占用。客户端集群中的每个节点应当并且仅能够保持一个消息接收连接。此连接为长连接，由于连接中断重连后需要重新进行服务选举（注册端口），因此应尽可能一直保持该连接有效并及时完成续租。

* **消息发送连接（1:N）**：所有未使用 WaitMsg API 升级的客户端连接均被视为发送连接，发送连接无需通过 Relet 保持心跳，仅使用 RegPort、UnRegPort、SendMsg 以及 QueryPort 等原语完成非推送类的客户端请求。客户端集群中的每个节点通常都会维护一个消息发

送连接池，以方便各工作线程高效地与端口转发服务保持通信。

与传统的分布式协调服务以及消息中间件产品相比，端口转发服务主要有以下特点：

* 功能性：端口转发服务将标准的消息路由功能集成到了服务选举（注册端口）、服务发现（发送消息和查询端口信息）、故障检测（续租超时）以及分布式锁（端口注册和注销通知）等分布式协调服务中。是带有分布式协调能力的高性能消息转发服务。通过 QueryPort 等接口，也可以将其单纯地当作带故障检测的服务选举和发现服务来使用。

* 高并发、高性能：由 C/C++/汇编实现；为每个连接维护一个消息缓冲队列，将所有端口定义及待转发消息均保存在内存中（Full in-memory）；主从节点间无任何数据复制和状态同步开销；信息的发送和接收均使用纯异步 IO 实现，因而可提供高并发和高吞吐的消息转发性能。

* 可伸缩性：在单点性能遭遇瓶颈后，可通过级联上级端口交换服务来进行扩展（类似 IDC 接入、汇聚、核心等多层交换体系）。

* 可用性：最低 5 毫秒内完成故障检测和主备切换的高可用保证，基于多数派的选举算法，避免由网络分区引起的脑裂问题。

* 一致性：保证任意给定时间内，最多只有一个客户端节点可持有某一特定端口。不可能出现多个客户端节点同时成功注册和持有相同端口的情况。

* 可靠性：所有发往未注册（不存在、已注销或已过期）端口的消息都将被安静地丢弃。系统保证所有发往已注册端口消息有序且不重复，但在极端情况下，可能发生消息丢失：

  - 端口交换服务宕机引起主从切换：此时所有在消息队列中排队的待转发消息均会丢失；所有已注册的客户端节点均需要重新注册；所有已注册的端口（服务和锁）均需要重新进行选举/获取（注册）。

  - 客户端节点接收连接断开重连：消息接收连接断开或重连后，所有该客户端节点之前注册的端口均会失效并需重新注册。在接收连接断开到重连的时间窗口内，所有发往之前与该客户端节点绑定的，且尚未被其它节点重新注册的端口之消息均被丢弃。

可见，白杨消息端口转发服务本身是一个集成了故障检测、服务选举、服务发现和分布式锁等分布式协调功能的消息路由服务。它通过牺牲极端条件下的可靠性，在保证了强一致、高可用、可伸缩（横向扩展）的前提下，实现了极高的性能和并发能力。

可以认为消息端口交换服务就是为 μSOA 架构量身定做的集群协调和消息分发服务。μSOA 的主要改进即：将在 SOA 中，每个用户请求均需要牵扯网络中的多个服务节点参与处理的模型改进为大部分用户请求仅需要同一个进程空间内的不同 BMOD 参与处理。

这样的改进除了便于部署和维护，以及大大降低请求处理延迟外，还有两个主要的优点：

* 将 SOA 中，需要多个服务节点参与的分布式事务或分布式最终一致性问题简化成为了本地 ACID Transaction 问题（从应用视角来看是如此，对于分布式 DBS 来说，以 DB 视角看来，事务仍然可以是分布式的），这不仅极大地简化了分布式应用的复杂度，增强了分布式应用的一致性，也大大减少了节点间通信（由服务间的 IPC 通信变成了进程内的指针传递），提高了分布式应用的整体效率。

* 全对等节点不仅便于部署和维护，还大大简化了分布式协作算法。同时由于对一致性要求较高的任务都已在同一个进程空间内完成，因此节点间通信不但大大减少，而且对消息中间件的可靠性也不再有过高的要求（通常消息丢失引起的不一致可简单地通过缓存超时或手动刷新来解决，可确保可靠收敛的最终一致性）。

在此前提下，消息端口交换服务以允许在极端情况下丢失少量未来得及转发的消息为代价，来避免磁盘写入、主从复制等低效模式，以提供极高效率。这对 nano-SOA 来说是一种非常合理的选择。

2.3.1 极端条件下的可靠性

传统的分布式协调服务通常使用 Paxos 或 Raft 之类基于多数派的强一致分布式算法实现，主要负责为应用提供一个高可用、强一致的分布式元数据 KV 访问服务。并以此为基础，提供分布式锁、消息分发、配置共享、角色选举、服务发现、故障检测等分布式协调服务。常见的分布式协调服务实现包括 Google Chubby（Paxos）、Apache ZooKeeper（Fast Paxos）、etcd（Raft）、Consul（Raft+Gossip）等。

Paxos、Raft 等分布式一致性算法的最大问题在于其极低的访问性能和极高的网络开销：对这些服务的每次访问，无论读写，都会产生至少三次网络广播——以投票的方式确定本次访问经过多数派确认（读也需要如此，因为主节点需要确认本次操作发生时，自己仍拥有多数票支持，仍是集群的合法主节点）。

在实践中，虽可通过降低系统整体一致性或加入租期机制来优化读操作的效率，但其总体性能仍十分低下，并且对网络 IO 有很高的冲击：Google、Facebook、Twitter 等公司的历次重大事故中，很多都是由于发生网络分区或人为配置错误导致 Paxos、Raft 等算法疯狂广播消息，致使整个网络陷入广播风暴而瘫痪。

此外，由于 Paxos、Raft 等分布式一致性算法对网络 IO 的吞吐和延迟等方面均有较高要求，而连接多座数据中心机房（IDC）的互联网络通常又很难满足这些要求，因此导致依赖分布式协调算法的强一致（抗脑裂）多活 IDC 高可用集群架构难以以合理成本实现。作为实例：2015 年 8 月 20 日 Google GCE 服务中断 12 小时并永久丢失部分数据；2015 年 5 月 27 日和 2016 年 7 月 22 日支付宝两次中断数小时；2013 年 7 月 22 日微信服务中断数小时；以及 2017 年 5 月英国航空瘫痪数日等重大事故均是由于单个 IDC 因市政施工（挖断光缆）等原因下线，同时未能成功构建多

活 IDC 架构，因此造成 IDC 单点依赖所导致的。

前文也已提到过：由于大部分采用 SOA 架构的产品需要依赖消息中间件来确保系统的最终一致性。因此对其可用性（部分节点故障不会影响正常使用）、可靠性（即使在部分节点故障时，也确保消息不丢失、不重复、并严格有序）、功能性（如：发布/订阅模型、基于轮转的任务分发等）等方面均有较严格的要求。这就必然要用到高可用集群、节点间同步复制、数据持久化等低效率、高维护成本的技术手段。因此消息分发服务也常常成为分布式系统中的一大主要瓶颈。

与 Paxos、Raft 等算法相比，BYPSS 同样提供了故障检测、服务选举、服务发现和分布式锁等分布式协调功能，以及相同等级的强一致性、高可用性和抗脑裂（Split Brain）能力。在消除了几乎全部网络广播和磁盘 IO 等高开销操作的同时，提供了数千、甚至上万倍于前者的访问性能和并发处理能力。可在对网络吞吐和延迟等方面无附加要求的前提下，构建跨多个 IDC 的大规模分布式集群系统。

与各个常见的消息中间件相比，BYPSS 提供了一骑绝尘的单点百万至千万条消息每秒的吞吐和路由能力——同样达到千百倍的性能提升，同时保证消息不重复和严格有序。

然而天下没有免费的午餐，特别是在分布式算法已经非常成熟的今天。在性能上拥有绝对优势的同时，BYPSS 必然也有其妥协及取舍——BYPSS 选择放弃极端（平均每年 2 次，并且大多由维护引起，控制在低谷时段，基于实际生产环境多年统计数据）情形下的可靠性，对分布式系统的具体影响包括以下两方面：

* 对于分布式协调服务来说，这意味着每次发生 BYPSS 主节点故障掉线后，所有的已注册端口都会被强制失效，所有活动的端口都需要重新注册。

  例如：若分布式 Web 服务器集群以用户为最小调度单位，为每位已登陆用户注册一个消息端口。则当 BYPSS 主节点因故障掉线后，每个服务器节点都会得知自己持有的所有端口均已失效，并需要重新注册当前自己持有的所有活动（在线）用户。

  幸运的是，该操作可以被批量化地完成——通过批量端口注册接口，可在一次请求中同时提交多达数百万端口的注册和注销操作，从而大大提升了请求处理效率和网络利用率：在 2013 年出厂的至强处理器上（Haswell 2.0GHz），BYPSS 服务可实现每核（每线程）100 万端口/秒的处理速度。同时，得益于我方自主实现的并发散列表（每个 arena 都拥有专属的汇编优化用户态读者/写者高速锁），因此可通过简单地增加处理器核数来实现处理能力的线性扩展。

  具体来说，在 4 核处理器+千兆网卡环境下，BYPSS 可达成约每秒 400 万端口注册的处理能力；而在 48 核处理器+万兆网卡环境下，则可实现约每秒 4000 万端口注册的处理能力（测试时每个端口名称的长度均为 16 字节），网卡吞吐量和载荷比都接近饱和。再加上其发生概率极低，并且恢复时只需要随着对象的加载来逐步完成重新注册，因此对系统整体性能几乎不会产生什么波动。

为了说明这个问题，考虑 10 亿用户同时在线的极端情形，即使应用程序为每个用户分别注册一个专用端口（例如：用来确定用户属主、完成消息分发等），那么在故障恢复后的第一秒内，也不可能出现“全球 10 亿用户心有灵犀地同时按下刷新按钮”的情况。相反，基于 Web 等网络应用的固有特性，这些在线用户通常要经过几分钟、几小时甚至更久才会逐步返回服务器（同时在线用户数=每秒并发请求数 x 用户平均思考时间）。即使按照比较严苛的“1 分钟内全部返回”（平均思考时间 1 分钟）来计算，BYPSS 服务每秒也仅需处理约 1600 万条端口注册请求。也就是说，一台配备了 16 核至强处理器和万兆网卡的入门级 1U PC Server 即可满足上述需求。

作为对比实例：官方数据显示，淘宝网 2015 年双十一当天的日活用户数（DAU）为 1.8 亿，同时在线用户数峰值为 4500 万。由此可见，目前超大型站点瞬时并发用户数的最高峰值仍远低于前文描述的极端情况。即使再提高数十倍，BYPSS 也足可轻松支持。

* 另一方面，对于消息路由和分发服务来说，这意味着每次发生 BYPSS 主节点故障掉线后，所有暂存在 BYPSS 服务器消息队列中，未及发出的待转发消息都将永久丢失。可喜的是，μSOA 架构不需要依赖消息中间件来实现跨服务的事务一致性。因此对消息投递的可靠性并无严格要求。

  意即：μSOA 架构中的消息丢失最多导致对应的用户请求完全失败，此时仍可保证数据的全局强一致性，绝不会出现“成功一半”之类的不一致问题。在绝大多数应用场景中，这样的保证已经足够——即使支付宝和四大行的网银应用也会偶尔发生操作失败的问题，这时只要资金等帐户数据未出现错误，那么稍候重试即可。

  此外，BYPSS 服务也通过高度优化的异步 IO，以及消息批量打包等技术有效降低了消息在服务器队列中的等待时间。具体来说，这种消息批量打包机制由消息推送和消息发送机制两方面组成：

  BYPSS 提供了消息批量发送接口，可在一次请求中同时提交数以百万计的消息发送操作，从而大大提升了消息处理效率和网络利用率。另一方面，BYPSS 服务器也实现了消息批量打包推送机制：若某节点发生消息浪涌，针对该节点的消息大量到达并堆积在服务器端消息队列中。则 BYPSS 服务器会自动开启批量消息推送模式——将大量消息打包成一次网络请求，批量推送至目的节点。

  通过上述的批量处理机制，BYPSS 服务可大大提升消息处理和网络利用效率，确保在大部分情况下，其服务器端消息队列基本为空，因此就进一步降低了其主服务器节点掉线时，发生消息丢失的概率。

  然而，虽然消息丢失的概率极低，并且 nano-SOA 架构先天就不怎么需要依赖消息中间件提供的可靠性保证。但仍然可能存在极少数对消息传递要求很高的情况。对于此类情况，可选择使用下列解决方案：

  - 自行实现回执和超时重传机制：消息发送方对指定端口发送消息，并等待接收该消

息处理回执。若在指定时段内未收到回执，则重新发送请求。

- 直接向消息端口的属主节点发起 RPC 请求：消息发送方通过端口查询命令获取该端口属主节点的 IP 地址等信息，并直接与该属主节点建立连接、提交请求并等待其返回处理结果。BYPSS 在此过程中仅担当服务选举和发现的角色，并不直接路由消息。对于视频推流和转码、深度学习等有大量数据流交换的节点间通信，也建议使用此方式，以免 BYPSS 成为 IO 瓶颈。

- 使用第三方的可靠消息中间件产品：若需要保证可靠性的消息投递请求较多，规则也较复杂，也可单独搭建第三方的可靠消息分发集群来处理这部分请求。

综上所述，可以认为 BYPSS 服务就是为 nano-SOA 架构量身定做的集群协调和消息分发服务。BYPSS 和 nano-SOA 架构之间形成了扬长避短的互补关系：BYPSS 以极端条件下系统整体性能的轻微波动为代价，极大提升了系统的总体性能表现。适合用来实现高效率、高可用、高可靠、强一致的 nano-SOA 架构分布式系统。

2.3.2 BYPSS 特性总结

BYPSS 和基于 Paxos、Raft 等传统分布式一致性算法的分布式协调产品特性对比如下：

项目	BYPSS	ZooKeeper、Consul、etcd…
可用性	高可用，支持多活 IDC	高可用，支持多活 IDC
一致性	强一致，主节点通过多数派选举	强一致，多副本复制
并发性	千万量级并发连接，可支持数十万并发节点	不超过 5000 节点
容量	每 10GB 内存可支持约 1 亿消息端口；每 1TB 内存可支持约 100 亿消息端口；两级并发散列表结构确保容量可线性扩展至 PB 级。	通常最高支持数万 KV 对。开启了变更通知时则更少。
延迟	相同 IDC 内每次请求延迟在亚毫秒级（阿里云中实测为 0.5ms）；相同区域内的不同 IDC 间每次请求延迟在毫秒级（阿里云环境实测 2ms）。	由于每次请求需要至少三次网络广播和多次磁盘 IO，因此相同 IDC 中的每操作延迟在十几毫秒左右；不同 IDC 间的延迟则更长（详见下文）。
性能	每 1Gbps 网络带宽可支持约 400 万次/秒的端口注册和注销操作。在 2013 年出厂的入门级至强处理器上，每核心可支持约 100 万次/秒的上述端口操作。性能可通过增加带宽和处理器核心数量线性扩展。	算法本身的特性决定了无法支持批量操作，不到 100 次每秒的请求性能（由于每个原子操作都需要至少三次网络广播和多次磁盘 IO，因此支持批量操作毫无意义，详见下文）。
网络利用率	高：服务器端和客户端均具备端口注册、端口注销、消息发送、端口查询、节点查询等原语的批量打包能力，网络载荷比可接近 100%。	低：每请求一个独立包（TCP Segment、IP Packet、Network Frame），网络载荷比通常低于 5%。
可伸缩性	有：可通过级联的方式进行横向扩展。	无：集群中的节点越多（因为广播和磁盘 IO 的范围更大）性能反而越差。

项目	BYPSS	ZooKeeper、Consul、etcd…
分区容忍	无多数派分区时系统下线，但不会产生广播风暴。	无多数派分区时系统下线，有可能产生广播风暴引发进一步网络故障。
消息分发	有，高性能，客户端和服务器均包含了消息的批量自动打包支持。	无。
配置管理	无，BYPSS 认为配置类数据应交由 Redis、MySQL、MongoDB 等专门的产品来维护和管理。当然，这些 CMDB 的主从选举等分布式协调工作仍可由 BYPSS 来完成。	有，可当作简单的 CMDB 来使用，这种功能和职责上的混淆不清进一步劣化了产品的容量和性能。
故障恢复	需要重新生成状态机，但可以数千万至数亿端口/秒的性能完成。实际使用中几无波动。	不需要重新生成状态机。

上述比较中，延迟和性能两项主要针对写操作。这是因为在常见的分布式协调任务中，几乎全部有意义的操作都是写操作。例如：

操作	对服务协调来说	对分布式锁来说
端口注册	成功：服务选举成功，成为该服务的属主。 失败：成功查询到该服务的当前属主。	成功：上锁成功。 失败：上锁失败，同时返回锁的当前属主。
端口注销	放弃服务所有权。	释放锁。
注销通知	服务已下线，可更新本地查询缓存或参与服务竞选。	锁已释放，可重新开始尝试上锁。

上表中，BYPSS 的端口注册对应 ZooKeeper 等传统分布式产品中的“写/创建 KV 对”；端口注销对应“删除 KV 对”；注销通知则对应“变更通知”服务。

由此可见，为了发挥最高效率，在生产环境中通常不会使用单纯的查询等只读操作。而是将查询操作隐含在端口注册等写请求中，请求成功则当前节点自身成为属主；注册失败自然会返回请求服务的当前属主，因此变相完成了属主查询（服务发现/名称解析）等读操作。

需要注意的是，就算是端口注册等写操作失败，其实还是会伴随一个成功的写操作。因为仍然要将发起请求的当前节点加入到指定条目的变更通知列表中，以便在端口注销等变更事件发生时，向各个感兴趣的节点推送通知消息。因此写操作的性能差异极大地影响了现实产品的实际表现。

2.3.3 基于 BYPSS 的高性能集群

从高性能集群（HPC）的视角来看，BYPSS 与前文所述的传统分布式协调产品之间，最大的区别主要体现在以下两个方面：

1. 高性能：BYPSS 通过消除网络广播、磁盘 IO 等开销，以及增加批处理支持等多种优化手段使分布式协调服务的整体性能提升了上万倍。

2. 大容量：约每 10GB 内存 1 亿个消息端口的容量密度，由于合理使用了并发散列表等数据结构，使得容量和处理性能可随内存容量、处理器核心数量以及网卡速率等硬件升级而线性扩展。

由于传统分布式协调服务的性能和容量等限制，在经典的分布式集群中，多以服务或节点作为单位来进行分布式协调和调度，同时尽量要求集群中的节点工作在无状态模式。服务节点无状态的设计虽然对分布式协调服务的要求较低，但同时也带来了集群整体性能低下等问题。

与此相反，BYPSS 可轻松实现每秒千万次请求的处理性能和万亿量级的消息端口容量。这就给分布式集群的精细化协作构建了良好的基础。与传统的无状态集群相比，基于 BYPSS 的精细化协作集群能够带来巨大的整体性能提升。

我们首先以最常见的用户和会话管理功能来说明：在无状态的集群中，在线用户并无自己的属主服务器，用户的每次请求均被反向代理服务随机地路由至集群中的任意节点。虽然 LVS、Nginx、HAProxy、TS 等主流反向代理服务器均支持基于 Cookie 或 IP 等机制的节点粘滞选项，但由于集群中的节点都是无状态的，因此该机制仅仅是增加了相同客户端请求会被路由到某个确定后台服务器节点的概率而已，仍无法提供所有权保证，也就无法实现进一步的相关优化措施。

而得益于 BYPSS 突出的性能和容量保证，基于 BYPSS 的集群可以用户为单位来进行协调和调度（即：为每个**活动用户**注册一个端口），以提供更优的整体性能。具体的实现方式为：

1. 与传统模式一样，在用户请求到达反向代理服务时，由反向代理通过 HTTP Cookie、IP 地址或自定义协议中的相关字段等方式来判定当前请求应该被转发至哪一台后端服务器节点。若请求中尚无粘滞标记，则选择当前负载最轻的一个后端节点来处理。

2. 服务器节点在收到用户请求后，在本地内存表中检查该用户的属主是否为当前节点。

   a) 若当前节点已是该用户属主，则由此节点继续处理用户请求。

   b) 若当前节点不是该用户的属主，则向 BYPSS 发起 RegPort 请求，尝试成为该用户的属主。此请求应使用批量方式发起，以进一步提高网络利用率和处理效率。

      i. 若 RegPort 请求成功，说明当前节点已成功获取该用户的所有权，此时可将用户信息由后端数据库加载到当前节点的本地缓存中（应使用批量加载优化），并继续处理此用户相关请求。

      ii. 若 RegPort 请求失败，说明指定用户正归于另一个节点管辖，此时应重新设置反向代理能够识别的 Cookie 等粘滞字段，将其指向正确的属主节点。并要求反向代理服务或客户端重试请求。

与传统架构相比，考虑到无状态服务也需要通过 MySQL、Memcached 或 Redis 等技术来实现专门的用户和会话管理机制，因此以上实现并未增加多少复杂度，但是其带来的性能提升却非

常巨大，对比如下：

项目	BYPSS HPC 集群	传统无状态集群
1 运维	省去用户和会话管理集群的部署和维护成本。	需要单独实施和维护用户管理集群，并为用户和会话管理服务提供专门的高可用保障，增加故障点、增加系统整体复杂性、增加运维成本。
2 网络	几乎所有客户请求的用户匹配和会话验证工作都得以在其属主节点的内存中直接完成。内存访问为纳秒级操作，对比毫秒级的网络查询延迟，性能提升十万倍以上。同时有效降低了服务器集群的内部网络负载。	每次需要验证用户身份和会话有效性时，均需要通过网络发送查询请求到用户和会话管理服务，并等待其返回结果，网络负载高、延迟大。 由于在一个典型的网络应用中，大部分用户请求都需要在完成用户识别和会话验证后才能继续处理，因此这对整体性能的影响很大。
3 缓存	因为拥有了稳定的属主服务器，而用户在某个时间段内总是倾向于重复访问相同或相似的数据（如自身属性，自己刚刚发布或查看的商品信息等）。因此服务器本地缓存的数据局部性强、命中率高。 相较于分布式缓存而言，本地缓存的优势非常明显： 1. 省去了查询请求所需的网络延迟，降低了网络负载（详见“项目 2”中的描述）。 2. 直接从内存中读取已展开的数据结构，省去了大量的数据序列化和反序列化工作。 与此同时，如能尽量按照某些规律来分配用户属主，还可进一步地提升服务器本地缓存的命中率。例如： a) 按租户（公司、部门、站点）来分组用户； b) 按区域（地理位置、游戏中的地图区域）来分组用户； c) 按兴趣特征（游戏战队、商品偏好）来分组用户。 等等，然后尽量将属于相同分组的用户优先分配给同一个（或同一组）服务器节点。显而易见，选择合适的用户分组策略可极大提升服务器节点的本地缓存命中率。	无专属服务器，用户请求随机到达集群中的任意服务器节点；本地缓存命中率低；各节点重复缓存的内容多；需要以更高的成本为代价依赖分布式缓存。 后端数据库服务器的读压力高，要对其实施分库分表、读写分离等额外优化。

项目	BYPSS HPC 集群	传统无状态集群
	这使得绝大部分与用户或人群相关的数据均可在本地缓存命中，不但提升了集群整体性能，还消除了集群对分布式缓存的依赖，同时大大降低了后端数据库的读负载。	
4 更新	由于所有权确定，能在集群全局确保任意用户在给定时间段内，均由特定的属主节点来提供服务。再加上现代服务器突发宕机故障的概率也较低。 因此可以将用户属性中频繁变化但重要性或时效性较低的部分缓存在内存中，待积累一段时间后再批量更新至数据库。这可大大降低后端数据库服务器的写压力。 例如：商城系统可能随着用户的浏览（比如每次查看商品）进程，随时收集并记录用户的偏好信息。若每次用户查看了新商品后，都需要即时更新数据库，则负载较高。再考虑到因为服务器偶发硬件故障导致丢失最后数小时商品浏览偏好数据完全可以接受，因此可由属主节点将这些数据临时保存在本地缓存中，每积累数小时再批量更新一次数据库。 再比如：MMORPG 游戏中，用户的当前位置、状态、经验值等数据随时都在变化。属主服务器同样可以将这些数据变化积累在本地缓存中，并以适当的间隔（比如：每 5 分钟一次）批量更新到数据库中。 这不但极大地降低了后端数据库要执行的请求数量，而且将多个用户的数据在一个批量事务中打包更新也大大减少数据库操作时的磁盘刷新动作，进一步提升了效率。 此外，由专门的属主节点发起对用户属性的更新也避免了无状态集群中多个节点同时请求更新同一对象时的争抢问题，进一步提高了数据库性能。	由于用户的每次请求都可能被转发到不同服务器节点来处理，因此无法实现累积写入优化和批量写入优化。后端数据库的写负担非常重。 存在多个节点争抢更新同一条记录的问题，进一步加重了数据库负担。 为此要对后端数据库进行额外的分库分表等优化，还会引发“需要由应用层来自行处理分布式事务”等副作用。
5 推送	由于同一用户发起的所有会话均被集中在同一个属主节点内统一管理，因此可非常方便地向用户推送即时通知消息（Comet）。	由于同一用户的不同会话被随机分配到不同节点处理，因此需要开发、部署和维护专门的消息推送集群，同时专门确保该集群的高性能和高可用性。

项目	BYPSS HPC 集群	传统无状态集群
	若发送消息的对象与消息接消息的收用户处于相同节点，则可直接将该消息推送给收件人麾下的所有活动会话。 否则只需将消息定向投递到收件人的属主节点即可。消息投递可使用 BYPSS 实现（直接向收件人对应端口发消息，应启用消息批量发送机制来优化），亦可通过专用的消息中间件（如：Kafka、RocketMQ、RabbitMQ、ZeroMQ 等）来完成。 若按照本表“项目 3”中描述的方法，优先将关联更紧密的用户分配到相同属主节点的话，则可大大提升消息推送在相同节点内完成的概率，此举可显著降低服务器间通信的压力。 因此我们鼓励针对业务的实际情况来妥善定制用户分组策略，合理的分组策略可实现让绝大部分消息都在当前服务器节点内本地推送的理想效果。 例如：对游戏类应用，可按地图对象分组，将处于相同地图副本内的玩家交由同一属主节点进行管理——传统 MMORPG 中的绝大部分消息推送都发生在同一地图副本内的玩家之间（AOI 范围）。 再比如：对于 CRM、HCM、ERP 等 SaaS 应用来说，可按照公司来分组，将隶属于相同企业的用户集中到同一属主节点上——很显然，此类企业应用中，近 100%的通信都来自于企业内部成员之间。 这样即可实现近乎 100%的本地消息推送率，达到几乎消除了服务器间消息投递的效果，极大地降低了服务器集群的内部网络负载。	这不但增加了开发和运维成本，而且由于需要将每条消息先投递到消息推送服务后，再由该服务转发给客户端，因此也加重了服务器集群的内部网络负载，同时也加大了用户请求的处理延迟。
6 平衡	集群可使用主被动负载平衡相结合的手段进行调度。 被动平衡：集群中的每个节点均会定期将其麾下不再活跃的用户和会话卸载掉，同时批量通知 BYPSS 服务释放这些用户所对应的端口。此算法实现了宏观上的负载平衡（以较长的时间周期来说，集群是平衡的）。	若启用了反向代理中的节点粘滞选项，则其负载平衡性与 BYPSS 集群的被动平衡算法相当。 若未启用反向代理中的节点粘滞选项，则在从故障中恢复时，其平衡性低于 BYPSS 主动平衡集群。与此同时，为了保证本地缓存命中率等其它性能指标不被过分劣化，管理员通常不会禁用

项目	BYPSS HPC 集群	传统无状态集群
	主动平衡：集群会通过 BYPSS 服务推选出负载平衡协调节点，该节点连续监视集群中各个节点的负载情况，并主动发出指令进行负载调度（如：要求 A 节点将其麾下 5000 位用户的所有权转移给 B 节点）。不同于宏观层面的被动平衡，主动平衡机制可以在更短的时间片内，以迅捷的反应速度来达成集群的快速配平。 主动平衡通常在集群中的部分节点刚刚从故障中恢复（因此处于空载状态）时效果明显，它比被动平衡反应更加迅速。如：在一个多活 IDC 集群中，某个 IDC 的光缆故障刚刚被修复而恢复上线时。	节点粘滞功能。 另外，SOA 架构的多个服务间，容易产生负载不平衡，出现一些服务超载，另一些轻载的情况，nano-SOA 集群则无此弊端。

值得一提的是，这样的精准协作算法并不会造成集群在可用性方面的任何损失。考虑集群中的某个节点因故障下线的情况：此时 BYPSS 服务会检测到节点已下线，并自动释放属于该节点的所有用户。待其用户向集群发起新请求时，该请求会被路由到当前集群中，负载最轻的节点。这个新节点将代替已下线的故障节点，成为此用户的属主，继续为该用户提供服务（见前文中的步骤 2-b-i）。此过程对用户透明，不需要在客户端中加入额外的处理逻辑。

上述讨论以几乎所有网络应用中都会涉及的用户和会话管理功能为例，为大家展示了 BYPSS HPC 集群精细协调能力的优势。但在多数真实应用中，并不只有用户管理功能。除此之外，应用中通常还会包含可供其用户操作的其它对象。例如在优酷、土豆、youtube 等视频网站中，除了用户以外，至少还有“可供播放的视频”这种对象。

下面我们就以“视频对象”为例，探讨如何使用 BYPSS 的精细化调度能力来大幅提升集群性能。

在这个假想的视频点播类应用中，与前文描述的用户管理功能类似，我们首先通过 BYPSS 服务为每个**活动的视频对象**选取一个属主节点。其次，我们将视频对象的属性分为以下两大类：

* 普通属性：包含了那些较少更新，并且尺寸较小的属性。如：视频封面和视频流数据在 S3 / OSS 等对象存储服务中的 ID、视频标题、视频简介、视频标签、视频作者 UID、视频发布时间等等。这些属性均符合读多写少的规律，其中大部分字段甚至在视频正式发布后就无法再做修改。

  对于这类尺寸小、变化少的字段，可以将其分布在当前集群中，各个服务器节点的本地缓存内。本地缓存有高性能、低延迟、无需序列化等优点，加上缓存对象较小的尺寸，再配合用户分组等进一步提升缓存局部性的策略，可以合理的内存开销，有效地提升应用整体性能（详见下文）。

* 动态属性：包含了所有需要频繁变更，或尺寸较大的属性。如：视频的播放次数、点赞

次数、差评次数、平均得分、收藏数、引用次数，以及视频讨论区内容等。

我们规定这类尺寸较大（讨论区内容）或者变化较快（播放次数等）的字段只能由该视频对象的属主节点来访问。其它非属主节点如需访问这些动态属性，则需要将相应请求提交给对应的属主节点来进行处理。

意即：通过 BYPSS 的所有权选举机制，我们将那些需要频繁变更（更新数据库和执行缓存失效），以及那些占用内存较多（重复缓存代价高）的属性都交给对应的属主节点来管理和维护。这就形成了一套高效的分布式计算和分布式缓存机制，大大提升了应用整体性能（详见下文）。

此外，我们还规定对视频对象的任何写操作（不管是普通属性还是动态属性）均必须交由其属主来完成，非属主节点只能读取和缓存视频对象的普通属性，不能读取动态属性，也不能执行任何更新操作。

由此，我们可以简单地推断出视频对象访问的大体业务逻辑如下：

1. 在普通属性的读取类用户请求到达服务器节点时，检查本地缓存，若命中则直接返回结果，否则从后端数据库读取视频对象的普通属性部分并将其加入到当前节点的本地缓存中。

2. 在更新类请求或动态属性读取类请求到达服务器节点时，通过本地内存表检查当前节点是否为对应视频对象的属主。

   a) 若当前节点已是该视频的属主，则由当前节点继续处理用户请求：读操作直接从当前节点的本地缓存中返回结果；写操作视情形累积在本地缓存中，或直接提交给后端数据库并更新本地缓存。

   b) 若当前节点不是该视频的属主，但在当前节点的名称解析缓存表中找到了与该视频匹配的条目，则将当前请求转发给对应的属主节点。

   c) 若当前节点不是该视频的属主，同时并未在当前节点的名称解析缓存表中查找到对应的条目，则向 BYPSS 发起 RegPort 请求，尝试成为该视频的属主。此请求应使用批量方式发起，以进一步提高网络利用率和处理效率。

      i. 若 RegPort 请求成功，说明当前节点已成功获取该视频的所有权，此时可将视频信息由后端数据库加载到当前节点的本地缓存中（应使用批量加载优化），并继续处理此视频相关请求。

      ii. 若 RegPort 请求失败，说明指定视频对象正归于另一个节点管辖，此时可将该视频及其对应的属主节点 ID 加入到本地名称解析缓存表中，并将请求转发给对应的属主节点来处理。

注意：由于 BYPSS 能够在端口注销时（无论是由于属主节点主动放弃所有权，还是该节点故障宕机），向所有对此事件感兴趣的节点推送通知。因此名称解析缓存表不需要类似 DNS 缓存的 TTL 超时淘汰机制，仅需在收到端口注销通知或 LRU 缓存满时删除对应条目即可。这不但能够大大增强查询表中条目的时效性和准确性，同时也有效地减少了 RegPort 请求的发送次数，提高了应用的整体性能。

与经典的无状态 SOA 集群相比，上述设计带来的好处如下：

项目	BYPSS HPC 集群	传统无状态集群
1 运维	基于所有权的分布式缓存架构，省去 Memcached、Redis 等分布式缓存集群的部署和维护成本。	需要单独实施和维护分布式缓存集群，增加系统整体复杂性。
2 缓存	普通属性的读操作在本地缓存命中，若使用“优先以用户偏好特征来分组”的用户属主节点分配策略，则可极大增强缓存局部性，增加本地缓存命中率，降低本地缓存在集群中各个节点上的重复率。 正如前文所述，相对于分布式缓存而言，本地缓存有消除网络延迟、降低网络负载、避免数据结构频繁序列化和反序列化等优点。 此外，动态属性使用基于所有权的分布式缓存来实现，避免了传统分布式缓存的频繁失效和数据不一致等问题。同时由于动态属性仅被缓存在属主节点上，因此也显著提升了系统整体的内存利用率。	无专属服务器，用户请求随机到达集群中的任意服务器节点；本地缓存命中率低；各节点重复缓存的内容多；需要以更高的成本为代价依赖额外的分布式缓存服务。 后端数据库服务器的读压力高，要对其实施分库分表、读写分离等额外优化。 此外，即使为 Memcached、Redis 等产品加入了基于 CAS 原子操作的 Revision 字段等改进，这些独立的分布式缓存集群仍无法提供数据强一致保证（意即：缓存中的数据与后端数据库里的记录无法避免地可能发生不一致）。
3 更新	由于所有权确定，能在集群全局确保任意视频对象在给定时间段内，均由特定的属主节点来提供写操作和动态属性的读操作等相关服务，再加上现代服务器突发宕机故障的概率也较低。 因此可以将动态属性中频繁变化但重要性或时效性较低的部分缓存在内存中，待积累一段时间后再批量更新至数据库。这可大大降低后端数据库服务器的写压力。 例如：视频的播放次数、点赞次数、差评次数、平均得分、收藏数、引用次数等属性都会随着用户点击等操作密集地变化。若每次发生相关的用户点击	由于每次请求都可能被路由到不同服务器节点来处理，因此无法实现累积写入优化和批量写入优化。后端数据库服务器的写负担非常重。存在多个节点争抢更新同一条记录的问题，这进一步加重了数据库负担。 为此要对后端数据库进行额外的分库分表等优化，还会引发“需要由应用层来自行处理分布式事务”等副作用。

项目	BYPSS HPC 集群	传统无状态集群
	事件后，都需要即时更新数据库，则负载较高。而在发生“属主节点由于硬件故障宕机”等极端情况时，丢失几分钟的上述统计数据完全可以接受。因此，我们可以将这些字段的变更积累在属主节点的缓存中，每隔数分钟再将其统一地批量写回后端数据库。 这不但极大地降低了后端数据库收到的请求数量，而且将多个视频的数据在一个批量事务中打包更新，也大大减少数据库操作时的磁盘刷新动作，进一步提升了效率。 此外，由专门的属主节点单独发起对视频记录的更新也避免了无状态集群中多个节点同时请求更新同一对象时的争抢问题，进一步提高了数据库性能。	
4 平衡	集群可使用主被动负载平衡相结合的手段进行调度。 被动平衡：集群中的每个节点均会定期将其麾下不再活跃的视频对象卸载掉，同时批量通知 BYPSS 服务释放这些视频对应的端口。此算法实现了宏观上的负载平衡（以较长的时间周期来说，集群是平衡的）。 主动平衡：集群会通过 BYPSS 服务推选出负载平衡协调节点，该节点连续监视集群中各个节点的负载情况，并主动发出指令进行负载调度（如：要求 A 节点将其麾下 10000 个视频对象的所有权转移给 B 节点）。不同于宏观层面的被动平衡，主动平衡机制可以在更短的时间片内，以迅捷的反应速度来达成集群的快速配平。 主动平衡通常在集群中的部分节点刚刚从故障中恢复（因此处于空载状态）时效果明显，它比被动平衡反应更加迅速。如：在一个多活 IDC 集群中，某个 IDC 的光缆故障刚刚被修复而恢复上线时。	在从故障中恢复时，其平衡性低于 BYPSS 主动平衡集群。正常情况下则相差不大。 另外，SOA 架构的多个服务间，容易产生负载不平衡，出现一些服务超载，另一些轻载的情况，nano-SOA 集群则无此弊端。

与前文提及的用户管理案例类似，上述精准协作算法不会为集群的服务可用性方面带来任何损失。考虑集群中的某个节点因故障下线的情况：此时 BYPSS 服务会检测到节点已下线，并自动释放属于该节点的所有视频对象。待用户下次访问这些视频对象时，收到该请求的服务器节点

会从 BYPSS 获得此视频对象的所有权并完成对该请求的处理。至此，这个新节点将代替已下线的故障节点成为此视频对象的属主（见前文中的步骤 2-c-i）。此过程对用户透明，不需要在客户端中加入额外的处理逻辑。

通过上述两个案例可以看出，相对于传统的 Redis + MySQL 等现有分布式缓存 + DB 的集群架构，BYPSS 集群还存在如下额外优势：

1. BYPSS 集群**保证强一致性**：正如前文所述，BYPSS 提供强一致、多活 IDC 高可用的高可用和高性能集群计算（HAC+HPC）能力。而 Redi 等分布式缓存，即使加入了 Revision 字段并利用其实现原子（CAS）操作，仍然无法保证其与 DB 之间的强一致性，只能通过加入失效超时（TTL）等机制来缓解数据不一致带来的危害（但此举同时也引入了缓存击穿、缓存雪崩等其它问题）。

2. BYPSS 集群可提供**数万倍量级的网络和 CPU 性能提升**：BYPSS 集群以纳秒级的内存直接访问替代了毫秒级的 Redis 网络查询，性能提升几个数量级。同时避免了每次缓存访问时频繁的数据序列化和反序列化过程，显著降低了 CPU 和网络开销。

3. BYPSS 集群可**完全避免缓存击穿**（失效）问题：由于 BYPSS 集群中的每个对象均仅缓存在其属主节点上，仅该对象的唯一属主节点有权向 DB 发送访问请求，因此不可能发生缓存击穿（不会产生针对某个热点对象的并发 DB 查询）。

4. BYPSS 集群可**轻易避免缓存穿透**（无效）问题：由于 BYPSS 集群中的每个对象均有其唯一属主节点，因此很容易在属主节点上对无效（不存在等）的对象 ID 进行标记和过滤（例如：高性能 ID 并发散列集合、布隆过滤器等）。因此可以轻易避免缓存穿透问题。

5. BYPSS 集群可**完全避免缓存雪崩**（大面积失效）问题：首先，由于 BYPSS 集群中缓存与 DB 数据间是强一致的，因此无需设置一个缓存过期时间来缓和缓存数据不一致的危害。因此缓存雪崩的一大前提：大量缓存同时到期就不复存在了。其次，由于消除了专门的缓存集群，所有缓存数据均内嵌在 App 服务器内，自然也就不存在“缓存节点宕机”这种问题了。与此同时，BYPSS 提供的强一致多活 IDC 高可用集群能力可保证 App Server 集群不会出现大规模宕机事故（不可抗力除外，真发生不可抗事件导致集群整体下线的话，那跟缓存雪崩已不是一个级别的问题了）。

以上对“用户管理”和“视频服务”案例的剖析均属抛砖引玉。在实际应用中，BYPSS 通过其高性能、大容量等特征提供的资源精细化协调能力可适用于包括互联网、电信、物联网、大数据批处理、大数据流式计算等广泛领域。

2.4 分布式消息队列服务（BYDMQ）

白杨分布式消息队列服务（BYDMQ，读作“by dark”）是一种强一致、高可用、高性能、高

吞度、低延迟、可线性横向扩展的分布式消息队列服务。可支持单点千万量级的并发连接以及单点每秒千万量级的消息转发性能，并支持集群的线性横向扩展。

BYDMQ 自身亦依赖 BYPSS 来完成其服务选举、服务发现、故障检测、分布式锁、消息分发等分布式协调工作。BYPSS 虽然也包含了高性能的消息路由和分发功能，但其主要设计目还是为了传递任务调度等分布式协调相关的控制类信令。而 BYDMQ 则专注于高吞吐、低延迟的大量业务类消息投递等工作。将业务类消息转移到 BYDMQ 后，可使 BYPSS 的工作压力显著降低。

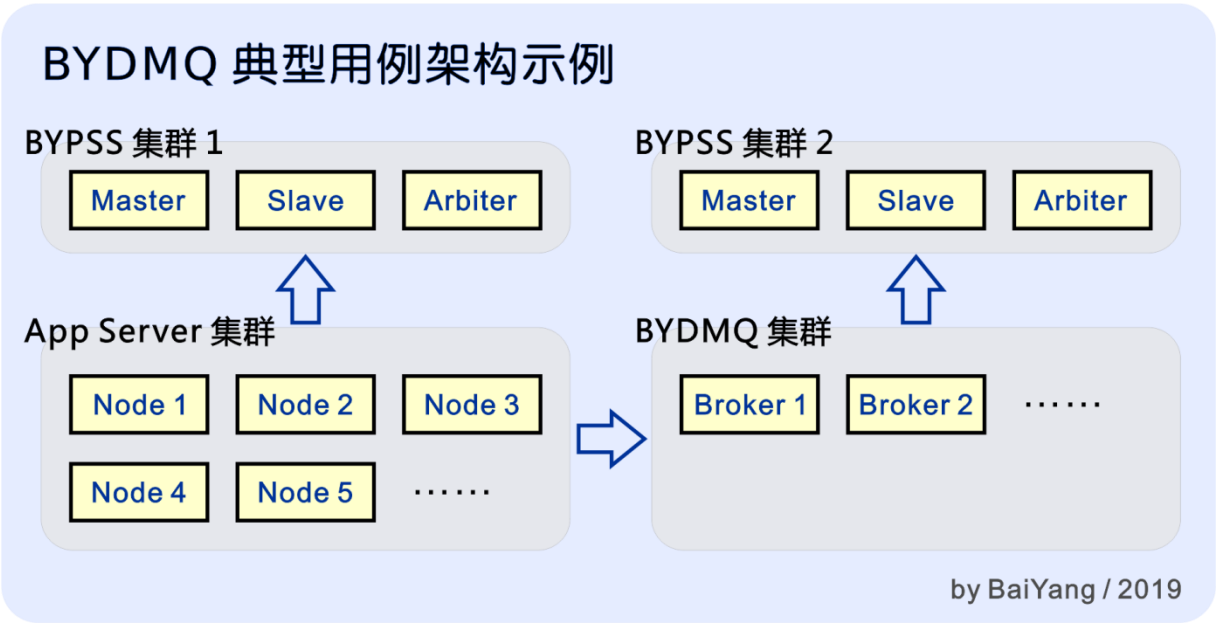


图 6

如图 6 所示，在典型用例中，BYDMQ 与 App Server 集群各自拥有一套独立的 BYPSS 集群，它们分别负责各自的分布式协调任务，App 集群依赖 BYPSS1 完成分布式协调，而其消息通信则依赖 BYDMQ 集群来完成。

不过在研发、测试环境，或业务量不大的生产环境中，也可以让 AppServer 和 BYDMQ 集群共享同一套 BYPSS 服务。另外需要指出的是，此处描述的“独立集群”仅是指逻辑上的独立。而从物理上说，即使是两个逻辑上独立的 BYPSS 集群，也可以共享物理资源。例如：一个 Arbiter 节点完全可以被多个 BYPSS 集群所共享；甚至两个 BYPSS 集群中的 Master、Slave 节点还可以互为主备，这样即简化了运维管理负担，又能够有效节约服务器硬件和能源消耗等资源。

在继续介绍 BYDMQ 的主要特性前，我们首先要澄清一个概念，即：消息队列（消息中间件，MQ）的可靠性问题。众所周知，“可靠的消息传递”包含三个要素——消息在投递过程中，要能够做到不丢失、不乱序和不重复才能称之为可靠。令人遗憾的是，这世间目前并不真正存在同时满足以上三个条件的消息队列产品。或者换句话说，我们目前尚无法在可接受的成本范围内实现同时满足以上三要素的消息队列产品。

要说明这个问题，请考虑以下案例：

图 7

如上图所示，在这一案例中，消息生产者由节点 A、B、C 组成，而消息消费者包含了节点 X、Y、Z，生产者与消费者之间通过一个消息队列连接。现在消息生产者已经生产了 5 条消息，并将它们依序成功提交到了消息队列。

在这样的情形下，我们来逐一讨论消息投递的可靠性问题：

* **消息不丢失**：这点是三要素里最容易保证的。可拆分为两步来讨论：

  - 存储可靠性：可以通过将每条消息同步复制到消息队列服务中的其它节点（Broker）并确保落盘来保证；同时需要使用 Paxos、Raft 等分布式共识协议来确保多个副本间的一致性。但是显而易见，与无副本的纯内存方案相比，由于增加了磁盘 IO、网络复制以及共识投票等步骤，此方案会极大（数千甚至上万倍）地降低消息队列服务的性能。

  - ACK 机制：在生产者向消息队列提交消息，以及消息队列向消费者投递消息时，均增加 ACK 机制来确认消息投递成功。发送方在指定时间内未收到 ACK 机制则重发（重新投递）消息。

    以上两步措施虽可在很大程度保证消息不丢失（至少一次送达），但是可以看到其开销十分巨大，对性能的劣化非常显著。

    与此同时，也应该注意到多副本间的故障检测和主从切换、以及消息收发时的超时重传等技术手段均会引入各自的延迟。而且其中每个步骤中引入的延迟通常都超过数秒钟。

    在真正的用户场景中，这些额外的延迟使得“消息不丢失”的保证除了平白增加了巨大开销以外，大多没有任何实际用处：如今的用户在发起一个请求（如：打开一个链接、提交一个表单等）后，很少有耐心等待许久——若等待数秒后仍然没有响应，他们早就关闭页面离开或 F5 重新刷新了。

此时无论用户是关闭了页面还是重新发起了请求，已经延迟了几秒（甚至更久）才到达的那条消息（老请求）都已经没有了价值。不但如此，处理这些请求更是在白白消耗网络、运算和存储资源而已——因为其处理结果已经无人问津。

* **消息不重复**：考虑前文提到的 ACK 机制：消费者在处理完一条消息后，需要向消息队列服务回复一条对应的 ACK 信令来确认该消息已被消费。例如：假设上图中的 1 号消息是一个转账请求，消息队列将该消息投递给节点 X 后，节点 X 必须在处理完该转账请求后，向消息队列服务发生一条形如“ACK：Msg=1”的信令来告知消息队列服务，该消息已被处理。而 MQ 无法确保消息不重复的矛盾即在于此：

  仍然按照上例中的假设，MQ 将消息 1 投递到了节点 X，但在规定的时间内却并未收到来自节点 X 的确认（ACK）信令，此时有很多种可能，例如：

  - 消息 1 未被处理：由于网络故障，节点 X 并未收到该消息。
  - 消息 1 未被处理：节点 X 收到了消息，但由于故障掉电而未能及时保存和处理该消息。
  - 消息 1 已被处理：节点 X 收到了并处理了该消息，但由于故障掉电而并能及时向 MQ 服务返回对应的 ACK 信令。
  - 消息 1 已被处理：节点 X 收到了并处理了该消息，但由于网络故障，对应的 ACK 信令未能成功返回至 MQ 服务。

    等等。由此可见，在消息投递超时后，MQ 服务是无法得知该消息是否已被消费的。雪上加霜的是，由于前文所述的原因（不能让用户等太久），这个超时通常还要设置的尽量短，这就让 MQ 服务正确感知实际情况变得更加不可能。

    此时为了保证消息不丢失，MQ 通常会假设消息未被处理，而重新分发该消息（例如在超时后，将消息 1 重新分发给节点 Y）。而这就势必无法再保证消息不重复了，反之亦然。

* **消息不乱序**：从上例中可以看出，所谓“消息不乱序”是指 MQ 中的消息要按照先来后到，以“1、2、3、4、5”的顺序逐一被消费。要保证严格的不乱序，就要求 MQ 必须等待一条消息处理结束（收到 ACK）后，才能继续分发队列中的下一条消息，这至少带来了以下问题：

  - 首先，MQ 中的多条消息无法被并行地消费。例如：MQ 无法将消息 1、2、3 同时分别派发给节点 X、Y、Z，这使得大量消费者节点长期处于饥饿（空闲）状态，甚至于即使在正在执行消息处理的节点上（比如节点 X）也会有大量处理器核心、SMT 单元等计算资源被浪费。

- 其次，在处理一条消息的过程中，所有其后续消息均只能处于等待状态。若一条消息投递失败（超时），则在其“超时-重新投递”期间内，则会长时间阻塞其所有后续消息，使得它们无法被及时处理。

  由此可见，保证消息严格有序会极大地影响系统整体的消息处理性能、增加硬件采购和运维成本，同时也会显著破坏用户体验。

由以上论述可知，现阶段尚无在合理成本下提供消息可靠传递的MQ产品问世。在此前提下，目前的解决方案主要是依赖 App Server 自身的业务逻辑（如：等幂操作、持久化状态机等）算法来克服这些问题。

反过来说：无论使用号称多“可靠”的 MQ 产品，现在的 App 业务逻辑中也均需要处理和克服上述种种消息投递不可靠的情形。既然 MQ 本质上做不到消息可靠，同时 App 也已经克服了这些不可靠性，那又何必再花费性能被劣化几千、甚至几万倍的代价来在 MQ 层实现支持“分布式存储 + ACK 机制”的方案呢？

基于上述思想，BYDMQ 并不像 RabbitMQ、RocketMQ 等产品那样，提供所谓（实际无法达到）的“可靠性”保证。相反，BYDMQ 采用“尽力送达”的模式，仅在确保不损失性能的前提下，尽可能地保证消息被可靠送达。

正如前文所述，由于 App 已经克服了消息传递过程中偶尔出现的不可靠。因此这样的设计抉择在极大提升了系统性能之余，并未实际增加业务逻辑的开发工作量。

基于上述设计理念的 BYDMQ 包含了以下特性：

* 与 BYPSS 一样基于白杨应用支撑平台中的各优质跨平台组件实现，如：单点支持千万量级并发的网络服务器组件；支持多核线性扩展的并发散列表容器等等。这些优质高性能组件使得 BYDMQ 在可移植性、可扩展性、高容量、高并发处理能力等方便均拥有非常好的表现。

* 与 BYPSS 一样，在客户端和服务器端均拥有成熟的消息批量打包机制。支持连续消息的自动批量打包，大大提高网络利用率和消息吞吐量。

* 与 BYPSS 一样支持 pipelining 机制：使得客户端无需等待一条指令的响应结果即可连续发送下一条指令，显著降低了命令处理延迟、提高了网络吞吐、有效增加了网络利用率。

* 每个客户端（App Server）可注册一个专属 MQ，并通过长连接 + 心跳的方式保活，对应的属主 Broker（BYDMQ 节点）也通过此长连接向客户端实时推送到达的消息（带有批量打包机制）。

* 客户端通过一致性散列算法来推测一个 MQ 的属主（Broker），Broker 在首次收到针对指定 MQ 的请求（如：注册、发消息等）时，将通过 BYPSS 服务来竞选成为该 MQ 的属

主。若竞选成功则继续处理，竞选失败则引导客户端重新连接到正确的属主节点。

与此同时，BYDMQ 会通过 BYPSS 服务实时感知集群变化（如：现有 Broker 下线、新的 Broker 上线等），并将这些变化实时推送到每个客户端节点。这就保证了除非 BYDMQ 集群正在发生剧烈变化（大量 Broker 节点上线或下线等），否则通过一致性散列算法推定属主的准确性是非常高的，从而基本无需再次重定向请求。

此外，即使一致性散列算法推定错误，该 MQ 的实际属主也会被客户端节点自动记忆到本地快查表中，确保下次向这个 MQ 发送消息时能够直接投递到正确的 Broker。

这种由客户端直接向对应 MQ 之属主（Broker）投递消息的方法避免了服务器集群中的复杂路由和消息的多次中转，将消息投递的网络路由降至最简，极大地提升了消息投递的效率，有效降低了网络负载。

通过 BYPSS 来为 MQ 选举属主节点的做法则为集群提供了“每个 MQ 在全局范围内唯一”的一致性保证。与此同时，BYPSS 服务也负责在各个 Broker 节点之间分发一些控制类信令（如：节点上下线通知等等），使 BYDMQ 集群可以被更好地统一协调和调度。

* 所有待分发的消息仅存储在对应 MQ 属主（Broker）节点的内存中，避免了写盘、复制、共识投票等大量无用开销。

* 发送方可以为每条消息分别设置其生命期（TTL）和发送失败时的最大重试次数。可根据消息的类型和价值精确控制其消耗的资源。对时效性短或不重要的请求，可及时使其失效，避免在各个环节继续空耗资源，反之亦然。

* 支持分散投递：当指定 MQ 所属客户端节点未上线，或其连接断开超过指定的时长后，此消息队列中的所有待投递消息将被随机发送到任意一个仍可正常工作的 MQ 中。

  分散投递方案预期客户端（App Server）也是一个基于 BYPSS 的 nano-SOA 架构集群。此时若一个 App Server 节点因为运维、硬件故障或网络分区等原因下线，则该节点下辖的所有对象都会被管理该集群的 BYPSS 释放。

  此时系统可随机将发往此节点的请求散布至其它仍正常工作的节点，可让这些目标节点通过 RegPort 重新获取该请求相关对象的所有权，并接替其已经下线的原属主节点继续完成处理。这就大大降低了在一个 App Server 节点异常下线的瞬间，与之相关请求的失败率，优化了客户体验。同时随机散布也使得下线节点麾下的对象被集群中仍然工作的其它节点均分，保证了集群的负载平衡。

综上，BYDMQ 通过在一定程度上牺牲了本就无法真正保证的消息可靠性，再配合消息打包、pipelining、属主直接投递等方式，极大地提升了消息队列服务的单点性能。同时得益于 BYPSS 为其引入的强一致、高可用、高性能的分布式集群计算能力，使其拥有了优异的线性横向扩展能力。加之其对每条消息的灵活控制、以及分散投递等特性，最终为用户提供了一款超高性能的高

品质分布式消息队列产品。

2.5 CConfig 数据结构

CConfig 是应用支撑平台提供的一种用于存储元信息的复杂数据结构。其特性如下：

* 支持树形层次结构：与 Windows 注册表相同，CConfig 可支持任意层次的子键嵌套。

* 支持丰富的值类型：与 Windows 注册表类似，可在任意键下添加子键、整形（DWORD）、字符串型（STRING）、字符串集合型（STRING_SET）以及二进制（BLOB）型等项目。除此之外，还支持 Windows 注册表中没有的布尔型（BOOL）、大整形（QWORD）和浮点型（DOUBLE）等扩展类型。

* 任意复杂度的 CConfig 对象都可以被保存为一个紧凑高效的数据块（BLOB），从而作为一个字段写入数据库。

* 二进制，解析效率高：CConfig 的存储格式基于 ISXF。ISXF 是一种在 libutilitis 中定义的平台无关的二进制格式数据表达形式。非常类似于 ASN.1 标准中的 BER 编码和 SUN 的外部数据表示（XDR）编码（作为 RPC 通信的基础，XDR 被广泛用于各个领域）。实际上，ISXF 正是在上述两种标准的基础上加以改进得到的。相比 ini、xml 等格式，ISXF 省略了高昂的文本解析和构造过程，极大地提高了数据访问效率。 同时紧凑的二进制格式也避免了空间上的浪费。

* 国际化：CConfig 中的字符串类型完全以 Unicode 字符集（UTF-8 编码）保存。辗转在各种不同语言的操作系统上也不会出现显示/保存乱码、数据不正确之类的问题。

* 跨平台：由于 CConfig 格式的存储结构基于 ISXF 格式实现。 保证了它以平台无关的格式存储。CConfig 格式的数据可以在各个平台间自由交换，并且被无二义性地解析。

* 高效率：CConfig 数据被载入后，所有目录（子键）和值都存放在 B+树中。即使是在非常庞大的项目中检索和访问也可以保持很高的效率。同时，不同于 Windows 注册表，CConfig 对象中没有 BLOB 型数据的尺寸限制。

* 灵活的交互能力：支持对 CSV/INI/JSON/XML 等格式导入和导出。使第三方应用不必实现复杂的 ISXF/CConfig 二进制数据解析引擎就可使用熟悉的数据格式完成交互。

综上，CConfig 最早是为在非 Windows 平台上虚拟 Windows 注册表服务而开发的功能模块。但在经过了多年的实际使用和改进后，其应用范围已经远超早先的预期，而成为了适合各个方面，用于进行配置和元信息管理的有力工具。

CConfig 数据可以以 CSV、INI、JSON、XML 以及 BLOB 等多种格式进行导入和导出，以下

对 CSV、INI、JSON、XML 等 WebAPI 中的常用格式分别进行说明。

提示：与下文中所用示例相对应的配置文件可从 http://baiy.cn/ccfg/example.cfg 下载。这是一个标准（BLOB）格式的 CConfig 数据文件，可以使用在 2.5.7.1 CConfig 配置编辑器中提到的工具进行访问。

2.5.1 CSV

CConfig 的 CSV 表示符合 RFC4180 规范。CSV 数据中每行描述一个对象（键或值），此格式支持导入和导出双向操作。其字段定义如下：

列号	字段	说明
1	键名	当前对象所在键的全路径 '\' 分割。
2	值名称	值的名称，若当前对象为 KEY 则此字段为空
3	对象类型	指定当前对象的类型，可以为： ▪ **KEY**：子键 ▪ **STRING**：字符串 ▪ **STRING_EXP**：可扩展字符串（可递归解析应用支撑平台环境管理器中定义的变量和底层操作系统中定义的环境变量） ▪ **BLOB**：二进制数据块 ▪ **BOOL**：布尔值 ▪ **DWORD**：32bit 整数 ▪ **QWORD**：64bit 整数 ▪ **FLOAT**：64bit 双精度浮点数 ▪ **STRING_SET**：字符串集合
4	值内容	值的内容，具体格式为： ▪ **KEY**：空 ▪ **STRING**：UTF-8 格式字符串 ▪ **STRING_EXP**：UTF-8 格式字符串 ▪ **BLOB**：HEX 编码的字符串，空格分割 ▪ **BOOL**：yes/true 或 no/false ▪ **DWORD**：十进制字符串表示 ▪ **QWORD**：十进制字符串表示 ▪ **FLOAT**：十进制字符串表示 ▪ **STRING_SET**：使用一个嵌套的类 CSV 格式字符串表示，与标准 CSV 的不同之处在于分隔符不是逗号而是分号、引用符不是双引号而是反单引号。

以下是一个合法的 CSV 格式 CConfig 数据示例（本示例位于 http://baiy.cn/ccfg/example.csv，与 http://baiy.cn/ccfg/example.cfg 相等价）：

```
,bool,BOOL,yes
,float,FLOAT,3.14159
,新值 #1,STRING_SET,"str1<;str2>;str3"";str4';str5="
,新值 #2,BLOB,2F 2F 7B 7B 4E 4F 5F 44 45 50 45 4E 44 45 4E 43
subkey1,,KEY,
subkey1,test,DWORD,123
subkey1,test2,QWORD,12345678901234567890
subkey1,test3,STRING,string
subkey1,test4,STRING_EXP,"exp string"
subkey1\subkey1.1,,KEY,
subkey1\subkey1.1,新值 #1,STRING,"a,b<c>d'e""f.g\h~"
新子键 #1,,KEY,
```

2.5.2 JSON

CConfig 的 JSON 表示符合 RFC4627 规范。将 CConfig 数据导出为 JSON 格式时，将会生成一个 JSON 格式的二维数组定义，其中第一维表示行，第二维表示列。其字段定义与 CSV 格式的版本十分相似。JSON 格式的个字段定义如下：

列号	字段	说明
1	键名	当前对象所在键的全路径‘\’分割。
2	值名称	值的名称，若当前对象为 KEY 则此字段为空
3	对象类型	指定当前对象的类型，可以为： ▪ **KEY**：子键 ▪ **STRING**：字符串 ▪ **STRING_EXP**：可扩展字符串（可递归解析应用支撑平台环境管理器中定义的变量和底层操作系统中定义的环境变量） ▪ **BLOB**：二进制数据块 ▪ **BOOL**：布尔值 ▪ **DWORD**：32bit 整数 ▪ **QWORD**：64bit 整数 ▪ **FLOAT**：64bit 双精度浮点数 ▪ **STRING_SET**：字符串集合

列号	字段	说明
4	值内容	值的内容，具体格式为： ▪ **KEY**：空 ▪ **STRING**：UTF-8 格式字符串 ▪ **STRING_EXP**：UTF-8 格式字符串 ▪ **BLOB**：HEX 编码的字符串，空格分割 ▪ **BOOL**：yes/true 或 no/false ▪ **DWORD**：十进制字符串表示 ▪ **QWORD**：十进制字符串表示 ▪ **FLOAT**：十进制字符串表示 ▪ **STRING_SET**：使用一个嵌套的 JSON 数组表示，其中的引号、回车、换行、换码符等关键字都将被换码，所以需要针对此字段再进行一次 eval 调用。

以下是一个合法的JSON格式CConfig数据示例(本示例位于 http://baiy.cn/ccfg/example.json，与 http://baiy.cn/ccfg/example.cfg 相等价)：

```
[
["","bool","BOOL","yes"],
["","float","FLOAT","3.14159"],
["","新值 #1","STRING_SET","[\"str1<\",\"str2>\",\"str3\\\"\",\"str4'\",\"str5=\"]"],
["","新值 #2","BLOB","2F 2F 7B 7B 4E 4F 5F 44 45 50 45 4E 44 45 4E 43"],
["subkey1","","KEY",""],
["subkey1","test","DWORD","123"],
["subkey1","test2","QWORD","12345678901234567890"],
["subkey1","test3","STRING","string"],
["subkey1","test4","STRING_EXP","exp string\n"],
["subkey1\\subkey1.1","","KEY",""],
["subkey1\\subkey1.1","新值 #1","STRING","a,b<c>d'e\"f.g\\h~"],
["新子键 #1","","KEY",""]
]
```

2.5.3 XML 格式

CConfig 数据的 XML 表示使用”<CConfig>”标签进行标记和界定，其格式较易理解。以下是一个合法的 XML 格式 CConfig 数据示例（本示例位于 http://baiy.cn/ccfg/example.xml，与 http://baiy.cn/ccfg/example.cfg 相等价)：

```
<CConfig>
    <key path="">
        <value name="bool" type="BOOL">yes</value>
```

```
        <value name="float" type="FLOAT">3.14159</value>
        <value name="新值 #1" type="STRING_SET">
            <string>str1&lt;</string>
            <string>str2&gt;</string>
            <string>str3&quot;</string>
            <string>str4&apos;</string>
            <string>str5=</string>
        </value>
        <value name="新值 #2" type="BLOB">2F 2F 7B 7B 4E 4F 5F 44 45 50 45 4E 44 45 4E
43</value>
    </key>
    <key path="subkey1">
        <value name="test" type="DWORD">123</value>
        <value name="test2" type="QWORD">12345678901234567890</value>
        <value name="test3" type="STRING">string</value>
        <value name="test4" type="STRING_EXP">exp string
</value>
    </key>
    <key path="subkey1\subkey1.1">
        <value name="新值 #1" type="STRING">a,b&lt;c&gt;d&apos;e&quot;f.g\h~</value>
    </key>
    <key path="新子键 #1"></key>
</CConfig>
```

需要注意的是：

* key 标签的 path 属性为全路径值。而 key 标签之间不会嵌套，它们之间的嵌套关系通过 path 属性表明。

* STRING_SET 中的每个独立字符串均使用一个”<string>”标签界定。

2.5.4 INI 格式

此格式专门为人类编辑和阅读进行优化，一般情况下不适合作为 WebAPI 交互格式。以下是一个合法的 INI 格式 CConfig 数据示例（本示例位于 http://baiy.cn/ccfg/example.ini，与 http://baiy.cn/ccfg/example.cfg 相等价）：

```
bool:BOOL=yes
float:FLOAT=3.14159
新值 #1:STRING_SET=str1<|str2>|str3"|str4'|str5=
新值 #2:BLOB=2F 2F 7B 7B 4E 4F 5F 44 45 50 45 4E 44 45 4E 43
```

```
[subkey1]
test:DWORD=123
test2:QWORD=12345678901234567890
test3:STRING=string
test4:STRING_EXP=exp string~n

[subkey1\subkey1.1]
新值 #1:STRING=a,b<c>d'e"f.g\h~~

[新子键 #1]
```

需要注意的是：

* INI 格式的换码符为‘~’。不使用传统的 ’\’ 来换码是因为 ’\’ 作为 Windows 下的路径分割符太过常用，而那些缺少编程经验的普通用户都不习惯将 ”C:\Windows\System” 记做 ”C:\\Windows\\System”。

* 换码序列中，回车符应记做 ’~r’，换行符应记做 ’~n’，使用连续的两个换码符来表示换码符本身 ’~~’。若需要对其它字符进行换码，仅需要在它的签名加上换码符即可。例如：如果要换冒号字符，仅需将其记做 ‘~:’ 即可。为了避免解析 INI 文件时发生混淆，INI 格式为不同部分分别定义了最小的强制换码序列如下：

  - 子键名称 ：（方括号中的内容）回车符、换行符。
  - 值名称 ：冒号’:’、回车符、换行符，以及首字母为 ’#’，’;’ 或 ’[’ 的值。
  - 字符串值 ：（STRING 和 STRING_EXP 值）回车符、换行符。
  - 字符串集合：（STRING_SET 值）回车符、换行符、管道分割符’|’

2.5.5 BLOB 格式

CConfig 数据原生使用从 ISO/ITU-T ASN.1-BER 和 SUN RPC-XDR 二进制编码格式优化而来的 ISXF 编码进行二进制存储。ISXF 二进制编码保证了其高效性、紧凑性和平台无关性。但由于需要专门编写编解码引擎以及二进制数据对人类的不友好，因此不建议在 WebAPI 中广泛使用。

总结来说，**INI 格式**是对人类最友好的格式（可读性和可编辑性最强），但同时也是对机器最不友好的（解析开销大，存储效率低）。与之相反的是 **BLOB 格式**，ISXF-BLOB 格式是对机器最友好（解析效率高，存储密度大）但对人类最不友好（完全不可读）的格式。**CSV、JSON、和 XML 格式**则介于两者之间，比较适合用于异构环境中的信息交互。

2.5.6 请求和提交指定格式的 CConfig 数据

在任何返回 CConfig 数据的 WebAPI 调用中，用户都可以通过 Accept 字段来指定需要返回的数据格式。Accept 是一个在 HTTP/1.1 中定义的标准 Header Field，它可以在 HTTP Request Header 中指定。目前支持的 Accept 值包括：

MIME-TYPE	说明
text/csv	以 CSV 格式返回数据
text/ini	以 INI 格式返回数据
application/json	以 JSON 格式返回数据
text/xml	以 XML 格式返回数据
application/x-isxf	以 BLOB（ISXF）二进制格式返回数据
none	用于读请求时表示无需返回相关的 CConfig 数据；用于写请求时表示无需修改 CConfig 格式的字段

类似地，在向服务器提交 CConfig 数据时，您可以通过 Content-Type 标准字段来指定要提交的数据格式。反之，当服务器返回一段 CConfig 数据时，同样也会通过 Response Header 中的 Content-Type 字段来说明格式信息。

2.5.7 配套支持

可为用户提供 CConfig 导出/导入、维护以及各种格式转换功能，并为浏览器端开发团队提供从 CSV/JSON 格式导入和 CSV 格式导出的 JavaScript 实现。

2.5.7.1 CConfig 配置编辑器

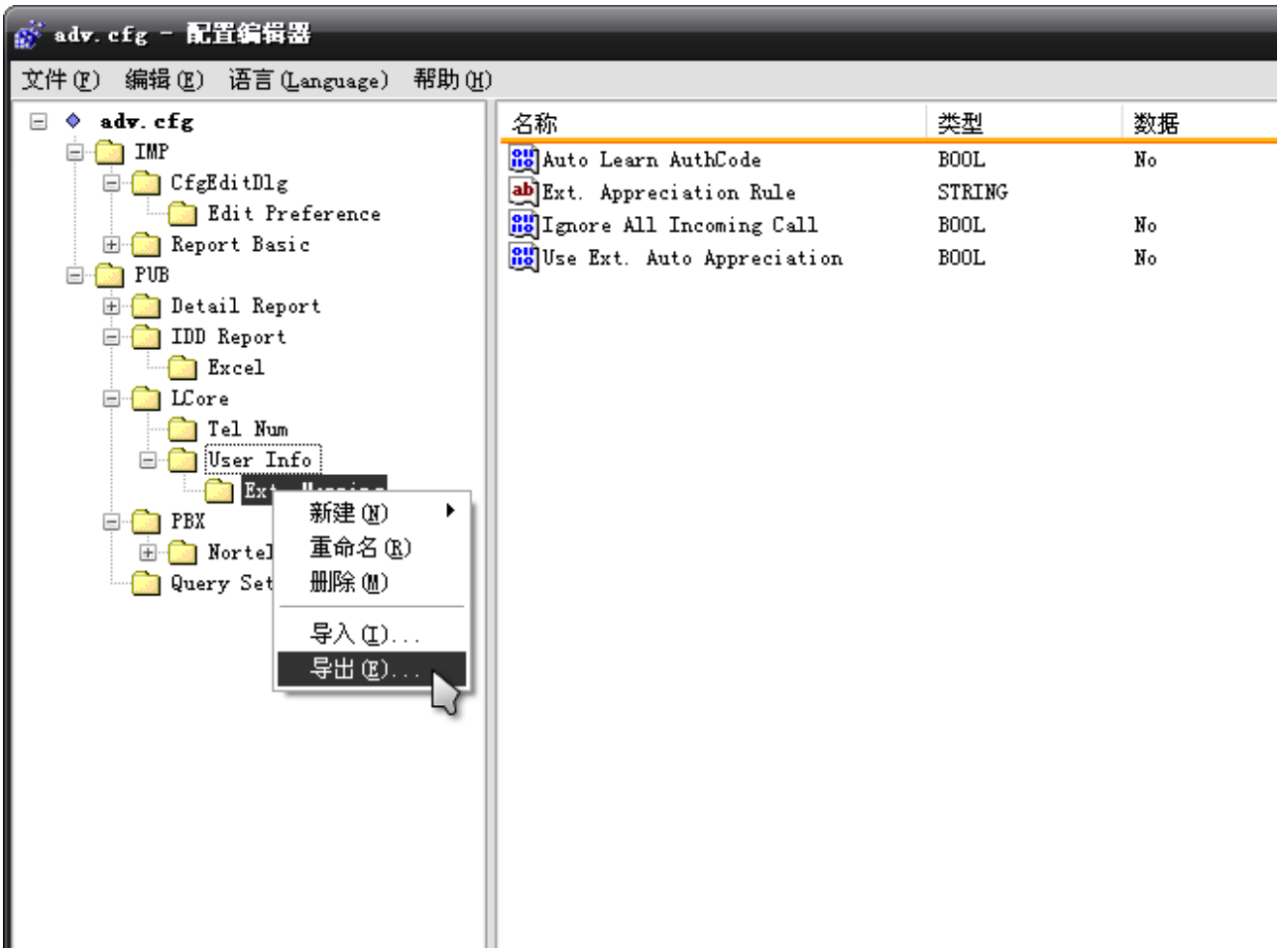


图 8 配置编辑器

作为一款辅助开发工具，配置编辑器提供一个类似 Windows 注册表编辑器的界面，可以方便地创建、编辑和保存 BLOB 格式的 CConfig 数据。与此同时，把配置编辑器还支持 CSV、INI 和 JSON 格式的导出以及从 CSV 或 INI 格式导入，在开发阶段可以方便地完成配置信息创建和格式转换等任务。请至此处：http://baiy.cn/ccfg/cfgedit.zip 下载 Windows 版配置编辑器工具。

2.5.7.2 浏览器端支持

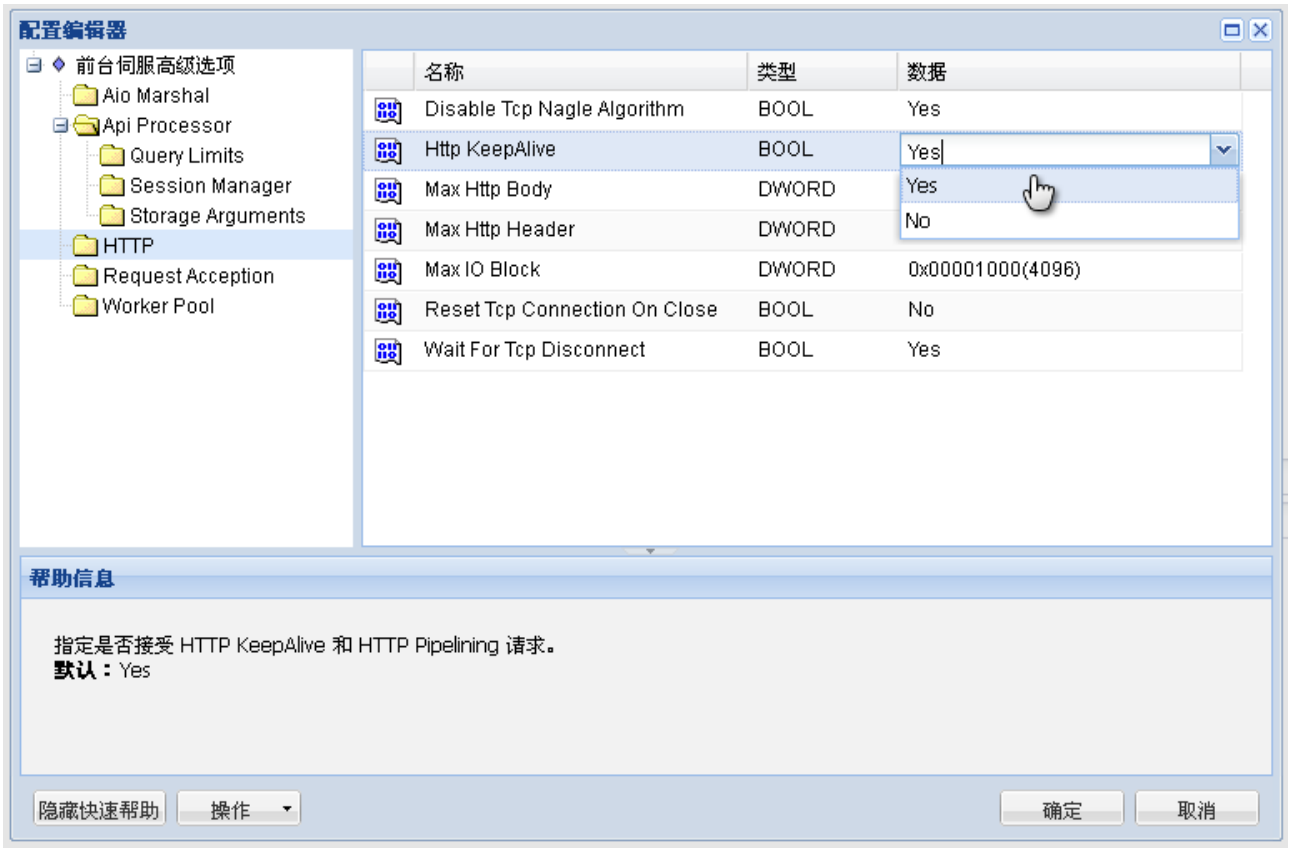


图 9 B/S 架构的配置编辑器组件

我们为用户提供了使用 JavaScript 实现的 CConfig 操作封装。其中包括了 CSV/JSON 解析和生成、子键和值的读取、删除、设置、遍历、以及存在性检查等各类常用操作。与此同时，我们还提供了一套基于 ExtJS 框架的配置编辑器组件。只需一个调用，您就可以将其方便地集成到自己的 B/S 产品中。

2.5.7.3 对 C++、Java、C#、VB.NET 和 PHP 等语言的支持

可以根据用户要求提供针对 C++、Java、C#、VB.NET、PHP、Golang、AngleScript、JavaScript、Duktape/JS 等各类主流语言和脚本引擎的 CConfig 封装组件。

3. 启动和配置 HAC Manager

3.1 命令行参数

HAC Manager 的命令行选项如下：

```
#BaiY_HACMgr -h

================================================================================
  BaiY HAC Manager Ver 1.0.15-0213
================================================================================

USAGE: BaiY_HACMgr [options]

================================================================================
Windows Only OPTIONs:
  -instsrv      install HAC Manager to system.
  -srvargs:     specify the service startup arguments for '-instsvr'.
  -startsrv     start HAC Manager.
  -stopsrv      stop HAC Manager.
  -rmsrv        remove HAC Manager from system.
  -srvname:     specify the service name to use. only useful when setting up
                multiple HAC Managers running concurrently on one server.

================================================================================
OPTIONs:
  -asdaemon      tell ws treat itself as a daemon. mainly used in posix enviro-
                 nment (e.g.: linux / unix ...) to start hm by an 'inittab'.
                 line, or by a script file in the '/etc/rc.d' directory.
  -libicuVer:    if the IBM ICU library is exists, specify the lib version.
                 DEFAULT: '38'.
  -pri:          specify the process priority, could be:
                   'realtime' (highest)
                   'high'
                   'above' (above normal, DEFAULT)
                   'normal' (normal)
                   'below' (below normal)
                   'low
                   'idle (lowest)'
```

```
  -cpumask:     specify the processor affinity mask.

================================================================================
Logging OPTIONs:
  -logfile:     specify a file to save the log message. NOTE: this option sup-
                press the period log file generation in $(hm.RootDir)/log dir.
  -logperiod:   specify the longest days to preserve a log message.
                DEFAULT: 90 days. the max. value is 1096.
  -logwindow:   enable this app write log info to console window.
                you can also specify the log window's line buffer size.
                DEFAULT: 350 lines if this switch is used with no argument.
  -logstdout    write log message to the standard out device.
  -syslog       enable to use the system logging facility. that is: send the
                log message to the 'Event Manager' on Windows or to 'syslogd'
                on an 'Unix Like' system.
  -rfc3164log:  log to the RFC3164 compatible syslog server, which is listening
                on the specified 'ip:port' (either IPv4 or IPv6).
  -loglevel:    setup the lowest message level to log, could be:
                  'debugonly' (the most detailed level)
                  'info' (infomation and above levels, DEFAULT)
                  'warning' (only warning and above)
                  'error'
                  'notice' (important notes)
                  'fatalerror (only fatal message)'
                  'disabled' (don't log anything)
  -logfilter=   wildcard logging source filter
  -logfilter!=  wildcard logging source filter (exclusion)
  -logfilterR=  regular expression logging source filter
  -logfilterR!= regular expression logging source filter (exclusion)
```

3.2 conf.ini 配置文件

HAC Manager 的行为可通过与主进程映像（BaiY_HACMgr）位于相同目录下的 conf.ini 文件进行配置。conf.ini 本质上是一段使用 INI 格式持久化的 CConfig 数据（详见：2.5 CConfig 数据结构），其中可以包含如下内容：

名称	类型	说明
Node ID	DWORD	[必选] 当前节点在 BYPSS 集群中的唯一 ID，不可为 -1（0xFFFFFFFF）。

名称	类型	说明
Service Name	STRING_EXP	[必选] 用于选举托管服务所有权的端口名。只有获得该端口所有权的 HAC Manager 才有权创建对应的托管服务进程。
Before Start	STRING_EXP	启动托管服务前执行的命令。在当前节点成功加入 BYPSS 集群，并获得“Service Name”端口所有权后，由 HAC Manager 自动调用。可以是 Shell 脚本等任何合法的系统命令。HAC Manager 将阻塞等待该命令执行完毕。
Start	STRING_EXP	[必选] 用于启动托管服务的命令。可以是 Shell 脚本等任何合法的系统命令。该命令在“Before Start”指令执行后运行，HAC Manager 将阻塞等待该命令执行完毕。 注意：如果此进程的返回码非零，则 HAC Manager 将判定本次托管服务启动失败，并按照以下顺序执行故障处理流程： 1. 记录相关出错日志。 2. 执行“On Fatal”处理过程（详见下文）。 3. 注销已取得的“Service Name”端口。 4. 终止当前（BaiY_HACMgr）进程。 在 BaiY_HACMgr 进程中止后，Windows Service Manager、upstart、systemd 等系统级服务管理工具可根据管理员预先配置的规则，尝试重启 HAC Manager Daemon。
After Start	STRING_EXP	启动托管服务后执行的命令。在当前节点成功加入 BYPSS 集群，并获得“Service Name”端口所有权后，由 HAC Manager 自动调用。可以是 Shell 脚本等任何合法的系统命令。HAC Manager 将阻塞等待该命令执行完毕。
Before Normal Stop	STRING_EXP	正常（非故障原因）停止托管服务前执行的命令。可以是 Shell 脚本等任何合法的系统命令。HAC Manager 将阻塞等待该命令执行完毕。 在紧急停止（“Emergency Stop”）或异常中止（“On Fatal”）时，不会执行此命令。

名称	类型	说明
Stop	STRING_EXP	[必选] 用于停止托管服务的命令。在当前节点从 BYPSS 集群正常（非故障原因）下线时，由 HAC Manager 自动调用。可以是 Shell 脚本等任何合法的系统命令。HAC Manager 将阻塞等待该命令执行完毕。 注意：如果此进程的返回码非零，则 HAC Manager 将判定本次托管服务停用失败，并按照以下顺序执行故障处理流程： 5. 记录相关出错日志。 6. 执行“On Fatal”处理过程（详见下文）。 7. 注销已取得的“Service Name”端口。 8. 终止当前（BaiY_HACMgr）进程。 在 BaiY_HACMgr 进程中止后，Windows Service Manager、upstart、systemd 等系统级服务管理工具可根据管理员预先配置的规则，尝试重启 HAC Manager Daemon。
Emergency Stop	STRING_EXP	用于在紧急情况下，尽快停止托管服务的命令。在当前节点从 BYPSS 集群异常（由于网络分区等原因，从 BYPSS 服务异常断开）下线时，由 HAC Manager 自动调用。可以是 Shell 脚本等任何合法的系统命令。HAC Manager 将阻塞等待该命令执行完毕。 若此选项不存在，则执行“Stop”指令来停止托管服务。 **注意：**在传统的，基于 DRBD 等难以完成可靠冲突解决的同步技术之主从集群中，应仔细评估此命令与“Dodge Time”选项（见下文）之间的时序关系。应确保此命令能够在从节点完成避让前执行完毕，否则可能产生 DRBD 层面的多主（脑裂）情形：两个 DRBD 镜像设备均工作在 Disconnect 模式下，主节点仍然在执行托管服务的停止过程，期间不断写入磁盘；而此时从节点则已将自身提升为主，并亦开始写盘。
After Stop	STRING_EXP	停止托管服务后执行的命令。可以是 Shell 脚本等任何合法的系统命令。HAC Manager 将阻塞等待该命令执行完毕。 此回调在“Stop”或“Emergency Stop”之后均会被执行，但不会在“On Fatal”回调后触发。

名称	类型	说明
On Fatal	STRING_EXP	指定遇到紧急故障时的处理指令。HAC Manager 不会等待该指令执行完毕。此命令亦不会触发“After Stop”和“Before Normal Stop”等回调。 **“On Fatal”处理过程如下：** 1. 若“On Fatal”处理指令存在，则执行并结束。 2. 否则：若“Emergency Stop”处理指令存在，则依次触发“Emergency Stop”和“After Stop”事件并结束。 3. 否则：若“Stop”处理指令存在，则依次触发“Before Normal Stop”、“Stop”和“After Stop”事件并结束。
Allow Slave Online	BOOL	指定是否允许工作在 Slave 状态（BYPSS 成功连接，但未取得服务所有权）的节点在线工作（例如：只读实例），默认：否。 在默认状态下，若 HACMgr 未持有“Service Name”指定端口的所有权，则自动停止当前服务。此时被托管服务的状态为 offline（stopped）。 但对于某些类型的托管服务（例如：基于 MySQL 半同步主从复制的高可用和读写分离集群等场景）来说，其在 Slave 状态下仍然应当继续保持在线工作（接收 binlog 的只读实例）。 针对这种情况，可将本选项设置为 Yes，此时 HACMgr 将为上述各个事件回调命令增加 <from> 和 <to> 命令行参数，调用格式为：″command <from> <to>″。 其中“<from>”为当前状态，“<to>”则为执行命令后的期望状态（目标状态）。HACMgr 目前支持如下三种状态： - master：在线（主模式）：BYPSS 已成功连接，并获得了服务所有权。 - slave ：在线（从模式）：BYPSS 已成功连接，但未获得服务所有权。 - stopped：下线（已停止）：BYPSS 尚未连接或已断开。 例如：″start stopped master″、″start slave master″、″start stopped slave″，以及 ″stop master stopped″、″stop master slave″、″stop slave stopped″ 等。

名称	类型	说明
		由此可见，在启用了本选项后，Start 命令的语义更偏向 Upgrade/Promote，而 Stop 命令的语义则更接近 Degrade。不过显而易见，“On Fatal”、“Emergency Stop”等命令的目标状态仍然只可能是 stopped（即：不可能“紧急降级”到 slave 状态）。
Slave	KEY	从节点配置，仅在主从模式下，当前节点为从节点时需要配置。
Slave\As Salve	BOOL	指定当前节点是否为从节点（默认：否）。
Slave\Dodge Time	DWORD	指定发现主节点掉线后，当前节点要等待多久才开始尝试将自身升级为主节点（秒，10），若为负数则其绝对值表示毫秒数（最低 -5：5ms，仅在高精度版本并 BYPSS“Lease expire”参数也为负值时有效）。0 表示不避让。 此选项的主要作用： 1. 防止因网络抖动引起的频繁故障转移和故障恢复动作； 2. 避免在故障切换过程中可能出现的多主（脑裂）情形（详见“Emergency Stop”选项相关说明）。
Slave\Failback	KEY	故障恢复相关配置。故障恢复与故障转移是两个相互对应的概念：故障转移（Failover）是指在集群检测到主节点发生故障下线后，由从节点接手其工作、扮演其角色，继续对外提供服务的过程；而故障恢复（Failback）则是当集群检测到主节点修复故障重新上线后，从节点将服务所有权重新归还给主节点的过程。 以下简称*〈SF〉*。
〈SF〉\Enable Failback	BOOL	指定是否启用故障恢复（默认：否）。 **注意**：主从节点均启用此选项后，才能开启 failback 支持。
〈SF〉\After Failback	STRING_EXP	指定要在故障恢复完成后执行的命令（仅在从节点执行）。HAC Manager 不会等待该指令执行完毕。
BYPSS	KEY	存放 BYPSS 客户端相关选项。 详见：2.3 消息端口交换服务。
BYPSS\PS Server1 Addr	STRING	[必选] 指定主服务器监听地址，格式为’IPv4:port’或’[IPv6]:port’。
BYPSS\PS Server2 Addr	STRING	指定从服务器监听地址，格式为’IPv4:port’或’[IPv6]:port’（可选，不填表示单主模式或已通过负载均衡中间件连接）。
BYPSS\AdvOpt	KEY	存放 BYPSS 客户端高级选项。 以下简称*〈BA〉*。

名称	类型	说明
<BA>\Close Timeout	DWORD	BYPSS Client 关闭超时（秒，默认：5）。
<BA>\Lease expire	DWORD	端口租期超时（秒，默认：3），若为负数则其绝对值表示毫秒数（最低 -5：5ms，仅对高精度版本有效）。
<BA>\Max Message Size	DWORD	SendMsg 能接受的最大单个消息尺寸(字节，默认:128KB)。
<BA>\Max Request Size	DWORD	单个 MSP 消息最大尺寸（字节，默认:1MB)。
<BA>\Max Queue Size	DWORD	消息接收队列最大尺寸（消息数，默认:100000)。
<BA>\Max Pool Conn.	DWORD	客户端连接池最大连接数（默认:32)。
<BA>\Async Batch Slice	DWORD	异步批量发送的消息收集窗口（秒，默认:1)。
Message Proxy	KEY	[可选，仅 MSGPROXY 版本支持] 存放 BYPSS 消息代理（即 MSP API，详见 4. MSP API 接口）相关配置。 以下简称*<MP>*。
<MP>\Enabled	BOOL	指定是否启用 BYPSS 消息代理(MSP API)服务(默认：否)。 **注意：仅 MSGPROXY 版支持 MSP API，在不支持 MSGPROXY 的普通版本中，此选项将被忽略。而 BYDMQ 版则是 MSGPROXY 版本下的一个子分支，它为 MSGPROXY 增加了 BYDMQ 分布式消息队列支持。**
<MP>\Message Format	STRING	指定消息体使用的 CConfig 编码格式。可以是“ini”、“json”、“xml”、“csv”以及“isxf”等。详见：2.5 CConfig 数据结构。
<MP>\Stdin Port	DWORD	指定消息接收代理监听端口号（默认：2333)。
<MP>\Stdin Queue Size	DWORD	指定消息接收缓存队列最大尺寸（默认：100000)。
<MP>\Stdout Port	DWORD	指定消息发送代理监听端口号（默认：2334)。
<MP>\Stdout Max Threads	DWORD	指定消息发送代理最大线程数（默认：24)。
<MP>\Max Owner Lookup PA	DWORD	[可选，仅 BYDMQ 版本支持] 指定每个 Arena 中的最大端口属主快查表条目数（默认：1000000)。
<MP>\WarmUp Messenger	DWORD	[可选，仅 BYDMQ 版本支持] 指定发送消息前是否尝试先解析不在快查表中的端口，可以是： - 0：不启用预热查询（默认），若目的端口不在当前 HAC Manager 实例内的端口属主快查表中，则使用 BYPSS 来分发该消息。并通过节点间自动学习来获取其对应的属主节点。 - 1：启用预热查询，由每个 stdout 连接分别进行消息预热查询。 - 其它（大于 1)：使用单独的预热查询消息投递线程对消息进行统一收集、批量预热查询及投递。此时本选项指定该线程的消息队列最大尺寸（消息数)。

名称	类型	说明
		在启用了 BYDMQ 的环境里，节点间自动学习机制永远有效，此选项不会影响该行为。 自动学习过程需要一段很短的时间窗口（LAN 环境下的典型值≤1ms），来习得指定端口的对应属主节点 ID。这就导致了每次向一个新端口发送的首条消息以及这个时间窗口内的后继消息都将通过 BYPSS（而不是 BYDMQ）来分发。因此，对一些总是随机地与新端口进行通信的特殊应用来说，应当启用此选项。 对于未自行收集和批量打包消息发送请求的托管服务，若启用了此选项，则建议使用一个大于 10000 的值，允许 HAC Manager 使用专门的线程批量收集、查询和打包发送消息。 在启用此选项后，每次消息发送时 HAC Manager 将自动检查目的端口有效性，发往无效端口（例如：尚未被注册或已被注销的端口）的消息将被安静地忽略。 在启用此选项后，发往当前节点自身的消息也将被检测并自动进行本地分发。但仍然建议托管服务自行进行本地端口判定，因为：首先将消息打包和发送到 HAC Manager 的行为本身已经产生了不必要的额外开销；其次，HAC Manager 每次判定本地端口均需要向 BYPSS 服务发起额外的 QueryPort 查询请求(在用户使用 0 值 Revision 进行端口注销时，缓存本地节点所属端口可能产生一致性问题)。
<MP>\Owner Lookup Arenas	DWORD	[可选，仅 BYDMQ 版本支持] 指定端口属主快查表 Arena 分区数（默认：8）。
<MP>\BYDMQ Msg TTL	DWORD	[可选，仅 BYDMQ 版本支持] 普通 BYDMQ 消息的 TTL（秒，默认：10），若为负数则其绝对值表示毫秒数（仅对高精度版本有效）。
<MP>\BYDMQ Msg Max Retry	DWORD	[可选，仅 BYDMQ 版本支持] 普通 BYDMQ 消息投递最大重试次数（默认：2）。
<MP>\BYDMQ BatchMsg TTL	DWORD	[可选，仅 BYDMQ 版本支持] 批量 BYDMQ 消息的 TTL（秒，默认：10），若为负数则其绝对值表示毫秒数（仅对高精度版本有效）。
<MP>\BYDMQ BatchMsg Max Retry	DWORD	[可选，仅 BYDMQ 版本支持] 批量 BYDMQ 消息投递最大重试次数（默认：2）。
BYDMQ	KEY	[可选，仅 BYDMQ 版本支持] 存放 BYDMQ 客户端相关选项。详见：2.4 分布式消息队列服务（BYDMQ）。
BYDMQ\DMQ Server List	STRING	指定 BYDMQ 服务器列表，格式为： "<节点 ID>=<IP 地址>[; <节点 ID>=<IP 地址>...]"。

名称	类型	说明
BYDMQ\Server Listen Port	DWORD	指定 BYDMQ 服务器的监听端口（所有 BYDMQ 服务器均使用相同端口，默认：799）。
BYDMQ\AdvOpt	KEY	存放 BYDMQ 客户端高级选项。 以下简称*<DA>*。
<BA>\Close Timeout	DWORD	BYPSS Client 关闭超时（秒，默认：5）。
<BA>\Lease expire	DWORD	消息队列租期超时（秒，默认：3），若为负数则其绝对值表示毫秒数（最低 -5：5ms，仅对高精度版本有效）。
<BA>\Max Message Size	DWORD	SendMsg 能接受的最大单个消息尺寸(字节，默认:128KB)。
<BA>\Max Request Size	DWORD	单个 MSP 消息最大尺寸（字节，默认:4MB）。
<BA>\Max Queue Size	DWORD	消息接收队列最大尺寸（消息数，默认:100000）。
<BA>\Max Pool Conn.	DWORD	客户端连接池最大连接数（默认:64）。
<BA>\Async Batch Slice	DWORD	异步批量发送的消息收集窗口（秒，默认:1）。
Env	KEY	定义 HAC Manager 内部环境变量，每个变量一个值，值的名称为变量名，值的内容为变量值。可在任何 STRING_EXP 类型的配置项中引用在这里和系统环境变量中定义的变量。格式为 $(<变量名>)，例如：'$(foo)' 等。
Env\<Name>	STRING	定义 HAC Manager 内部环境变量，每个变量一个 STRING 类型的值。其中值的名称为变量名，值的内容为变量值。 HAC Manager 中包含了如下预定义内部环境变量： ▬ hm.RootDir: [只读] HAC Manager 进程的启动目录。 ▬ hm.TmpDir: 当前 HAC Manager 独享的临时文件目录。

4. MSP API 接口

正如前文所述，得益于 BYPSS 集群提供的丰富能力，HAC Manager 不仅提供了主从/多主选举、故障发现、故障转移等 HAC 功能，还可以作为本地代理（类似 Service Mesh 中的 sidecar proxy 角色），向托管服务提供 BYPSS 服务中的其它分布式协调功能。

因此，为了进一步提高灵活性，在强一致高可用（HAC）集群的基础上进一步支持高性能分布式计算（HPC）的能力，用户可以要求 HacMgr 将 BYPSS 提供的服务发现、服务选举、故障检测、故障转移、故障恢复、分布式锁、任务调度，以及消息路由和消息分发等各项分布式协调功能暴露给正在接受 HAC Manager 托管的后端应用程序。

BYPSS 代理服务监听在本机（127.0.0.1）上，通过由用户指定的两个 TCP 端口与后端托管服务通信，其功能如下：

* **stdin**：默认监听在 2333 端口，将从 BYPSS 服务收到的端口注销通知，以及其它节点发来的单播、组播和广播消息转发至后端托管进程。后端应用与 stdin 端口之间仅支持单 TCP 连接。

* **stdout**：默认监听在 2334 端口，将后端托管服务发出的分布式协调指令转发给 BYPSS 消息交换服务，同时提供自动消息打包机制。后端应用与 stdout 端口之间可支持并发多连接。

MSP API 功能的启用和禁用，以及上述监听端口等各项配置信息均可在 conf.ini 配置文件中指定（参考：3.2 conf.ini 配置文件）。

stdin 和 stdout 连接均使用 MSP 封包格式，详情请见：2.2 MSP（Message Session Protocol）协议。其消息格式为 CConfig，具体的编码方式亦可在 3.2 conf.ini 配置文件中指定。

使用 MSP API 接口进行开发之前，您需要对 BYPSS 分布式算法的各个原语有基本了解（参考：2.3 消息端口交换服务）。

4.1 MSP API 在托管应用中的典型用法

在启用了 HAC Manager 的 BYPSS 消息代理（MSP API）后，推荐的后端托管应用架构为：

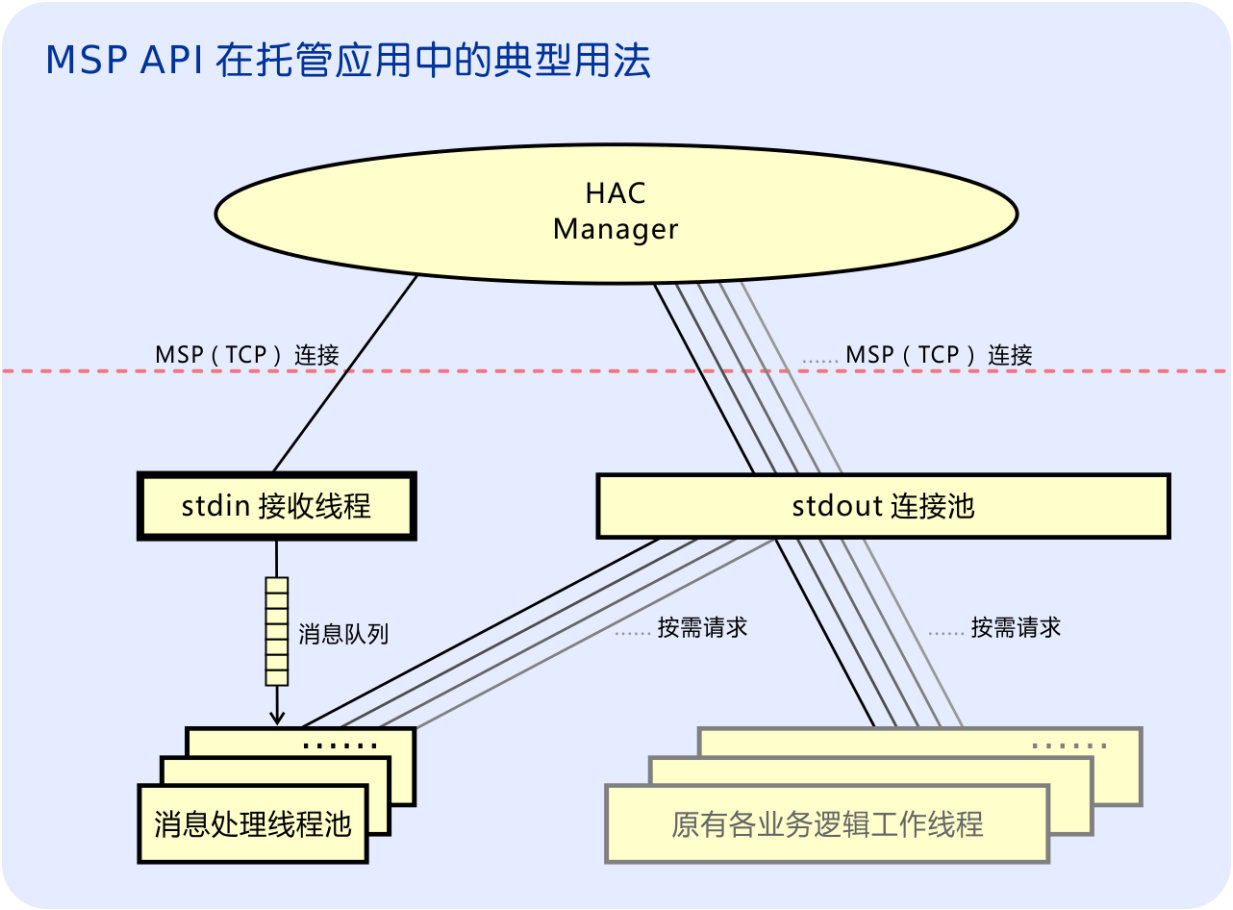


图 10

图 10 展示了 MSP API 的一种典型接入模式。我们建议用户在其托管应用中新增如下组件：

- **stdin 接收线程**：负责与 HacMgr 的 stdin 端口建立长连接，等待消息到达，并将接收到的消息加入专用的消息队列中。
- **消息处理线程池**：不断从 stdin 线程的消息队列中取出并处理到达的消息。
- **stdout 连接池**：保持与 stdout 端口建立的长连接池，在消息处理线程池或托管应用原业务逻辑中，需要与 HAC Manager 进行主动通信时，可从连接池中获取一个空闲连接来使用，并在通信完成后将该连接归还给 stdout 连接池。

  此外，stdout 连接应该被封装为一个更易于使用的类（其下包括 SendMsg、RegPort、UnRegPort 等方法），以便于使用和维护。

需要注意的是，节点上线和节点下线等事件不会通过 stdin 连接通知托管应用。相反，MAC Manager 将通过启动、终止托管应用等行为来对相关事件做出反应。

4.2 stdin 消息推送接口

HAC Manager 目前可通过 stdin 连接推送的消息包括：

4.2.1 端口注销通知（UnRegNotify）

用户在指定端口上订阅了注销通知时推送，端口注销通知的格式为：

名称	类型	说明
TYPE	DWORD	当前消息类别，为 0 表示端口注销通知。
Port	STRING	被注销的端口名称。
OriOwnerNID	DWORD	端口原属主节点 ID。

4.2.2 用户消息推送（UserMsgPush）

由其它节点的托管程序单播、组播或广播到当前节点的用户自定义消息，其格式如下：

名称	类型	说明
TYPE	DWORD	当前消息类别，为 1 表示用户消息。
ToPort	STRING	消息目的端口，为空表示广播消息。
FromNID	DWORD	消息来源节点 ID。
Body	KEY	消息体，其中内容由用户自行定义。

4.3 stdout API 接口

当一个发往 stdout 的命令执行失败时，HAC Manager 返回以下错误信息：

名称	类型	说明
CMD	DWORD	已执行的命令 ID（若在尚未成功解析消息前即发生错误，则此值可能不存在）。
Err	STRING	错误描述。

HAC Manager 在 stdout 连接上提供如下 API 接口：

4.3.1 端口注册（RegPort）

RegPort 命令向 BYPSS 服务请求获得指定端口的所有权，可通过一次指定多个端口来实现批量请求。其请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 0。
UnRegNotify	DWORD	指定要启用的注销通知选项，可以是： ▬ URN_NO = 0, // 无需注销通知 ▬ URN_YES = 1, // 若注册失败，则订阅端口注销通知 ▬ URN_ALL = 2, // 若注册成功，则端口注销时向集群中所有其它节点广播注销通知
Ports	KEY	存放要注册的端口信息
Ports\<Name>	DWORD	每个要注册的端口一个值，值的名称为要进行注册的端口名，值的内容将被忽略。 注意：“#”和“$”开头的端口名为系统保留。若用户指定的端口名由上述符号开头，则 HAC Manager 将会返回相应错误。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（0）。
Success	KEY	存放已成功注册的端口信息。
Success\<Name>	DWORD	每个成功注册的端口一个值，值的名称为已成功注册的端口名，值的内容为该端口本次注册的 Revision。 系统保证返回 Revision 满足如下特征： 1. Revision 是一个不为零的 32 位无符号整数。 2. 若同一个属主多次调用 RegPort 方法对同一个已注册的端口进行重复注册，则系统保证每次成功返回的 Revision 均与上一次成功注册时的不同。
Fail	KEY	存放已失败的注册端口信息。
Fail\<Name>	DWORD	每个注册失败的端口一个值，值的名称为该端口的名称，值的内容为当前拥有该端口的属主节点 ID。 特别地：若值的内容为 -1（0xFFFFFFFF），则表示端口名无效或 BYPSS 服务的端口容量已到达上限。

4.3.2 端口注销（UnRegPort）

UnRegPort 命令向 BYPSS 服务请求释指定端口的所有权，可通过一次指定多个端口来实现批量请求。需要注意的是：一个节点只能注销它所拥有的端口，若请求注销的端口不属于该节点，则此请求将被 BYPSS 服务安静地忽略。

UnRegPort 请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 1。
Ports	KEY	存放要注销的端口列表。
Ports\<Name>	DWORD	每个要注销的端口一个值，值的名称为要注销的端口名，值的内容为该端口的 Revision，为 0 表示忽略 Revision 匹配，强行注销指定端口。 注意：“#”和“$”开头的端口名为系统保留。若用户指定的端口名由上述符号开头，则 HAC Manager 将会安静地忽略该请求。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（1）。

4.3.3 消息发送（SendMsg，单播和广播）

SendMsg 命令向指定的端口发送消息，可通过一次指定多个请求来实现批量发送。此方法不保证消息可达性和有序性，仅保证消息在每个指定端口上最多被消费一次。

除非用户自行实现了消息的批量收集和打包机制（例如：用户自行实现了专门的消息集中收集和打包线程），否则不建议使用此方法进行消息发送。**换句话说，除非用户根据自身业务特性实现更优的消息批量收集和打包机制，否则强烈建议使用 4.3.8 异步消息发送（AsyncSendMsg）方法来进行日常消息发送，以获得更好的性能。**

注意：发送消息前，请先判定目标端口的属主节点是否就是自身，以避免不必要的消息投递开销（HAC Manager 不会跟踪和记录应用程序具体持有哪些端口，因此不会进行此类优化）。

此外，对于消息密集型集群，建议与 BYPSS 同时部署 BYDMQ 强一致高性能分布式消息中间件。部署后，HAC Manager 会优先使用 BYDMQ 服务来发送消息，显著降低 BYPSS 的负载。

另外，单条消息和批量 MSP 消息包的最大尺寸均可在 conf.ini 中配置（参考：3.2 conf.ini 配

置文件）。

SendMsg 命令请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 2。
<SEQ>	KEY	每个要发送的消息一个子键，子键的名称为要发送的消息序号，如“000”,“001”等。当存在多个消息时，HAC Manager 将按照其字母顺序执行消息批量发送。
<SEQ>\ToPort	STRING	指定要发送的消息端口。为空表示将消息广播到当前 BYPSS 集群中的所有在线节点。若指定的端口不存在，则 BYPSS 服务将安静地忽略该请求。 特别地，若端口名称为“#”加十进制数字，则表明向指定的服务器节点发送消息。例如：“#123”表示向 Node ID 为 123 的服务器节点发送消息。
<SEQ>\Body	KEY	指定要发送的消息体，其中内容对 BYPSS 服务透明。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（2）。

4.3.4 消息组播（MulticastMsg）

MulticastMsg 命令向指定的端口集合发送消息（组播），可通过一次指定多个请求来实现批量发送。请注意此方法不保证消息可达性和有序性，仅保证消息在每个指定端口上最多被消费一次。

除非用户自行实现了消息的批量收集和打包机制（例如：用户自行实现了专门的消息集中收集和打包线程），否则不建议使用此方法进行消息组播。**换句话说，除非用户根据自身业务特性实现更优的消息批量收集和打包机制，否则强烈建议使用 4.3.9 异步消息组播（AsyncMulticastMsg）方法来进行日常消息组播，以获得更好的性能。**

注意：发送消息前，请先判定目标端口的属主节点是否就是自身，以避免不必要的消息投递开销（HAC Manager 不会跟踪和记录应用程序具体持有哪些端口，因此不会进行此类优化）。

此外，对于消息密集型集群，建议与 BYPSS 同时部署 BYDMQ 强一致高性能分布式消息中间件。部署后，HAC Manager 会优先使用 BYDMQ 服务来发送消息，显著降低 BYPSS 的负载。

另外，单条消息和批量 MSP 消息包的最大尺寸均可在 conf.ini 中配置（参考：3.2 conf.ini 配置文件）。

MulticastMsg 命令请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 3。
<SEQ>	KEY	每个要发送的消息一个子键，子键的名称为要发送的消息序号，如“000”,“001”等。当存在多个消息时，HAC Manager 将按照其字母顺序执行消息批量发送。
<SEQ>\ToPorts	STRING_SET	指定要组播的消息端口集合。 特别地，若端口名称为“#”加十进制数字，则表明向指定的服务器节点发送消息。例如：“#123”表示向 Node ID 为 123 的服务器节点发送消息。
<SEQ>\Body	KEY	指定要发送的消息体，其中内容对 BYPSS 服务透明。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（3）。

4.3.5 端口查询（QueryPort）

QueryPort 命令向 BYPSS 服务查询指定端口的所有者（属主）相关信息，可通过一次指定多个查询来实现批量请求。其请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 4。
Ports	STRING_SET	指定要查询的端口列表。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（4）。
<PortName>	KEY	存放已完成查询的端口信息。每个端口一个子键，子键的名称即为端口名。
<PortName>\NID	DWORD	指定端口当前的属主节点 ID。-1（0xFFFFFFFF）表示该端口目前无属主（尚未被注册或已被释放）。
<PortName>\Address	STRING	属主节点的网络地址（通常为 IP 地址）。为空表示该端口目前无属主（尚未被注册或已被释放）。

4.3.6 节点查询（QueryNode）

QueryNode 命令向 BYPSS 服务查询指定节点的相关信息，可通过一次指定多个查询来实现批量请求。其请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 5。
Nodes	STRING	指定要查询的节点列表。格式为逗号分割的节点 ID（DIGITAL STRING）。例如：“0,3,25”等等。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（5）。
<NID>	STRING	每个节点一个值，值的名称为该节点的 ID（DIGITAL STRING），值的内容为该节点的网络地址（通常为 IP 地址），内容为空表示该的节点不存在。

4.3.7 异步端口注销（AsyncUnRegPort）

此方法与 4.3.2 端口注销（UnRegPort）功能相同，区别在于请求到达 HAC Manager 后将会被加入到一个内部请求队列中，并在专门的工作线程中发送。

与 UnRegPort 相比，此方法几乎不会引入额外的延迟，并且由于工作线程可以自动对连续到达的消息进行自动批量化打包（Mini Batch）优化，因此除用户已自行实施了批量处理优化等特殊情况以外，我们均鼓励用户尽量使用此方法来处理所有需要即刻生效的请求，以尽可能优化底层网络，保证网络连接的高载荷比和高吞度量。

AsyncUnRegPort 请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 101。
Ports	KEY	存放要注销的端口列表。
Ports\<Name>	DWORD	每个要注销的端口一个值，值的名称为要注销的端口名，值的内容为该端口的 Revision，为 0 表示忽略 Revision 匹配，强行注销指定端口。 注意：“#”和“$”开头的端口名为系统保留。若用户指定的端口名由上述符号开头，则 HAC Manager 将会安静地忽略该请求。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（101）。

4.3.8 异步消息发送（AsyncSendMsg）

此方法与 4.3.3 消息发送（SendMsg，单播和广播）功能相同，区别在于请求到达 HAC Manager 后将会被加入到一个内部请求队列中，并在专门的工作线程中发送。

与 SendMsg 相比，此方法几乎不会引入额外的延迟，并且由于工作线程可以自动对连续到达的消息进行自动批量化打包（Mini Batch）优化，因此除用户已自行实施了批量处理优化等特殊情况以外，我们均鼓励用户尽量使用此方法来处理所有需要即刻生效的请求，以尽可能优化底层网络，保证网络连接的高载荷比和高吞度量。

对于实时性要求不高的消息投递，可使用 4.3.11 批量消息发送（BatchSendMsg）来进一步提高整体吞吐。

注意：发送消息前，请先判定目标端口的属主节点是否就是自身，以避免不必要的消息投递开销（HAC Manager 不会跟踪和记录应用程序具体持有哪些端口，因此不会进行此类优化）。

此外，对于消息密集型集群，建议与 BYPSS 同时部署 BYDMQ 强一致高性能分布式消息中间件。部署后，HAC Manager 会优先使用 BYDMQ 服务来发送消息，显著降低 BYPSS 的负载。

AsyncSendMsg 命令请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 102。
<SEQ>	KEY	每个要发送的消息一个子键，子键的名称为要发送的消息序号，如“000”,“001”等。当存在多个消息时，HAC Manager 将按照其字母顺序执行消息批量发送。
<SEQ>\ToPort	STRING	指定要发送的消息端口。为空表示将消息广播到当前 BYPSS 集群中的所有在线节点。若指定的端口不存在，则 BYPSS 服务将安静地忽略该请求。 特别地，若端口名称为“#”加十进制数字，则表明向指定的服务器节点发送消息。例如：“#123”表示向 Node ID 为 123 的服务器节点发送消息。
<SEQ>\Body	KEY	指定要发送的消息体，其中内容对 BYPSS 服务透明。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（102）。

4.3.9 异步消息组播（AsyncMulticastMsg）

此方法与 4.3.4 消息组播（MulticastMsg）功能相同，区别在于请求到达 HAC Manager 后将会被加入到一个内部请求队列中，并在专门的工作线程中发送。

与 MulticastMsg 相比，此方法几乎不会引入额外的延迟，并且由于工作线程可以自动对连续到达的消息进行自动批量化打包（Mini Batch）优化，因此除用户已自行实施了批量处理优化等特殊情况以外，我们均鼓励用户尽量使用此方法来处理所有需要即刻生效的请求，以尽可能优化底层网络，保证网络连接的高载荷比和高吞度量。

对于实时性要求不高的消息投递，可使用 4.3.12 批量消息组播（BatchMulticastMsg）来进一步提高整体吞吐。

注意：发送消息前，请先判定目标端口的属主节点是否就是自身，以避免不必要的消息投递开销（HAC Manager 不会跟踪和记录应用程序具体持有哪些端口，因此不会进行此类优化）。

此外，对于消息密集型集群，建议与 BYPSS 同时部署 BYDMQ 强一致高性能分布式消息中间件。部署后，HAC Manager 会优先使用 BYDMQ 服务来发送消息，显著降低 BYPSS 的负载。

AsyncMulticastMsg 命令请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 103。
<SEQ>	KEY	每个要发送的消息一个子键，子键的名称为要发送的消息序号，如“000”,“001”等。当存在多个消息时，HAC Manager 将按照其字母顺序执行消息批量发送。
<SEQ>\ToPorts	STRING_SET	指定要组播的消息端口集合。 特别地，若端口名称为“#”加十进制数字，则表明向指定的服务器节点发送消息。例如：“#123”表示向 Node ID 为 123 的服务器节点发送消息。
<SEQ>\Body	KEY	指定要发送的消息体，其中内容对 BYPSS 服务透明。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（103）。

4.3.10 批量端口注销（BatchUnRegPort）

此方法与 4.3.2 端口注销（UnRegPort）功能相同，区别在于请求到达 HAC Manager 后，不会被立刻处理。相反，HAC Manager 会将该请求加入到一个内部请求队列中，并在专门的工作线程中，周期性（通常为 1、2 秒一次，具体由“Async Batch Slice”配置项决定，详见：3.2 conf.ini 配置文件）地从该队列中批量收集和批量处理请求，以期进一步提升网络利用率。

具体的异步批量处理周期可以在 conf.ini 中设置（默认为 2 秒，详见：3.2 conf.ini 配置文件）。强烈推荐用户尽量使用此方法来处理所有无需即刻生效的请求，以尽可能优化底层网络，保证网络连接的高载荷比和高吞度量。

AsyncUnRegPort 请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 201。
Ports	KEY	存放要注销的端口列表。
Ports\<Name>	DWORD	每个要注销的端口一个值，值的名称为要注销的端口名，值的内容为该端口的 Revision，为 0 表示忽略 Revision 匹配，强行注销指定端口。 注意：“#”和“$”开头的端口名为系统保留。若用户指定的端口名由上述符号开头，则 HAC Manager 将会安静地忽略该请求。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（201）。

4.3.11 批量消息发送（BatchSendMsg）

此方法与 4.3.3 消息发送（SendMsg，单播和广播）功能相同，区别在于请求到达 HAC Manager 后，不会被立刻处理。相反，HAC Manager 会将该请求加入到一个内部请求队列中，并在专门的工作线程中，周期性地从该队列中批量收集和处理请求，以期进一步提升网络利用率。

具体的异步批量处理周期值可以在 conf.ini 中设置（默认为 2 秒，详见：3.2 conf.ini 配置文件）。**强烈推荐用户尽量使用此方法来投递所有无需即刻发送的消息，以尽可能优化底层网络，保证网络连接的高载荷比和高吞度量**。

注意：发送消息前，请先判定目标端口的属主节点是否就是自身，以避免不必要的消息投递开销（HAC Manager 不会跟踪和记录应用程序具体持有哪些端口，因此不会进行此类优化）。

此外，对于消息密集型集群，建议与 BYPSS 同时部署 BYDMQ 强一致高性能分布式消息中间件。部署后，HAC Manager 会优先使用 BYDMQ 服务来发送消息，显著降低 BYPSS 的负载。

AsyncSendMsg 命令请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 202。
<SEQ>	KEY	每个要发送的消息一个子键，子键的名称为要发送的消息序号，如“000”,“001”等。当存在多个消息时，HAC Manager 将按照其字母顺序执行消息批量发送。
<SEQ>\ToPort	STRING	指定要发送的消息端口。为空表示将消息广播到当前 BYPSS 集群中的所有在线节点。若指定的端口不存在，则 BYPSS 服务将安静地忽略该请求。 特别地，若端口名称为“#”加十进制数字，则表明向指定的服务器节点发送消息。例如：“#123”表示向 Node ID 为 123 的服务器节点发送消息。
<SEQ>\Body	KEY	指定要发送的消息体，其中内容对 BYPSS 服务透明。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（202）。

4.3.12 批量消息组播（BatchMulticastMsg）

此方法与 4.3.4 消息组播（MulticastMsg）功能相同，区别在于请求到达 HAC Manager 后，不会被立刻处理。相反，HAC Manager 会将该请求加入到一个内部请求队列中，并在专门的工作线程中，周期性地从该队列中批量收集和处理请求，以期进一步提升网络利用率。

具体的异步批量处理周期值可以在 conf.ini 中设置（默认为 2 秒，详见：3.2 conf.ini 配置文件）。**强烈推荐用户尽量使用此方法来投递所有无需即刻发送的消息，以尽可能优化底层网络，保证网络连接的高载荷比和高吞度量**。

此外，对于消息密集型集群，建议与 BYPSS 同时部署 BYDMQ 强一致高性能分布式消息中间件。部署后，HAC Manager 会优先使用 BYDMQ 服务来发送消息，显著降低 BYPSS 的负载。

注意：发送消息前，请先判定目标端口的属主节点是否就是自身，以避免不必要的消息投递开销（HAC Manager 不会跟踪和记录应用程序具体持有哪些端口，因此不会进行此类优化）。

AsyncMulticastMsg 命令请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 203。
<SEQ>	KEY	每个要发送的消息一个子键，子键的名称为要发送的消息序号，如“000”,“001”等。当存在多个消息时，HAC Manager 将按照其字母顺序执行消息批量发送。
<SEQ>\ToPorts	STRING_SET	指定要组播的消息端口集合。 特别地，若端口名称为“#”加十进制数字，则表明向指定的服务器节点发送消息。例如：“#123”表示向 Node ID 为 123 的服务器节点发送消息。
<SEQ>\Body	KEY	指定要发送的消息体，其中内容对 BYPSS 服务透明。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（203）。

4.3.13 获取节点状态（GetStatus）

GetStatus 命令向当前 HAC Manager 查询当前节点的相关信息。其请求格式如下：

名称	类型	说明
CMD	DWORD	[必选] 指定要执行的命令，必须为 301。

若命令执行成功，则返回的响应中可包含如下内容：

名称	类型	说明
CMD	DWORD	已执行的命令（301）。
NodeID	DWORD	当前节点 ID（在当前 BYPSS 集群中唯一）。
Status	DWORD	当前节点状态，可以是： - S_MASTER = 2, // 在线（主模式）：BYPSS 成功连接，并获得了服务所有权 - S_SLAVE = 1, // 在线（从模式）：BYPSS 成功连接，但未获得服务所有权。 注意：仅在启用了“Allow Slave Online”配置项后，才有可能出现 S_SLAVE 状态。
SrvPort	STRING	当前服务的注册端口，仅在 Sataus 为 S_MASTER 时存在。