# BaiY Application Platform

# Technical White Paper

**Version: 2.43**

**http://baiy.cn**

Copyright © 2001-2023 BaiYang. All rights reserved.

# Introduction

It has been 16 years since the publication of the first edition of BaiY Application Platform. During the years, I have written more than one million lines of codes for the platform using Assembly language and C/C++ languages. The most fundamental and important part of the platform, the libutilitis library, is wholly implemented by myself. In addition to the Windows platform that I had already been familiar with, I have gradually become familiar with and become fond of some other operating systems like Linux, FreeBSD/NetBSD/OpenBSD and OpenSolaris, during the process of encapsulating a number of cross-platform* functions into libutilitis.

My main approach for implementing the other libraries within the platform is to design a set of interfaces and frameworks on the basis of the libutilitis library. The specific features, such as cryptographic and compression algorithms, the audio encoder/decoder, and cross platform UI components are implemented using stable third-party codes that are freely available. After taking into account third-party open source code, the application platform contains more than five million lines of code.

A variety of products that are built upon the application platform have been widely deployed in different production environments, such as:

- State Grid Corporation of China (SGCC, Top 2 in the world)
- China National Petroleum Corp (CNPC, Top 5 in the world)

- General Electric Company (GE, Top 5 in the world)
- Agricultural Bank of China (ABC, Global 500)
- China Industrial Bank (CIB, Global 500)
- China Everbright Bank (CEB)
- Sinosafe Insurance
- taobao.com (The largest e-commerce platform in China)
- SOCIETE GENERALE (SOCIETE GENERALE, The second largest bank in Europe, Global 500)
- Delphi (Global 500)
- United Airlines (Global 500)
- China Unicom (Global 500)
- China Mobile (Top 50 in the world)
- Bertelsmann (Global 500, The world's largest outsourcing call center)
- Teleperformance SE (TPC, French telecommunications company, 300,000 employees worldwide)
- Accenture (Global 500)
- Eldman (The world's largest public relations company)
- China Soong Ching Ling Foundation (National Fund founded by Deng Xiaoping)
- BMW Group (Global 500)
- Shaanxi Automobile Group
- One Foundation
- yiguo.com (China's leading fresh food e-commerce)
- Yantai Wanhua Group

And etc. The wide deployment in real production environments not only provides a reliable and platform-independent infrastructure for the high-level applications, but also has verified the reliability, stability, portability and efficiency of the application platform.

Copyright of the Application Platform belongs to BaiYang, wherein a number of technologies are subject to a number of national and international patents protections.

The application platform currently supports the following operating systems:

- The full range of Windows operating systems including Win98/ME, WinNT4/2000/XP/2k3/Vista/2k8/Win7/2k8r2/Win8/8.1/2012/2012r2, etc.

- Linux, FreeBSD, NetBSD, IBM, AIX, HP-UX, Solaris, MAC OS X, and a variety of Un*x/POSIX systems
- vxWorks, QNX, SMX, DOS, WinCE (Windows Mobile), NanoGUI, eCos, RTEMS, Android, iOS and other embedded systems.

The currently supported hardware platforms include x86/x64, ARM, RISC-V, IA64, MIPS, POWER, SPARC and etc.

# A Quick Brief

As previously described, BaiY Application Platform contains millions of lines of assembly, C / C++ code and thousands of mature general-purpose components. It has been tested in the real production environment of numerous Fortune 500 companies. It has been used in multiple high-load telecommunications, Internet and distributed computing environments for more than a decade.

Thousands of mature and reliable high-quality functional components can greatly enhance the quality of software products in terms of performance, functionality, and stability. It also brought unimaginable convenience for the development of the product. For example:

## High-performance IO server components

The Application Platform uses assembly and asynchronous IO to optimize the network service components. These components enable high performance network services through the memory zero-copy and asynchronous IO mechanisms via DMA + hardware interrupts. On an entry-level 1U PC Server (with dual-socket Intel Xeon 56xx) manufactured in 2011 (at that time, the price of the machine

was less than 20,000 CNY), a single node can permit tens of millions of TCP / HTTP concurrent connections. Correspondingly, with the same machine, a general server development by Java or .NET can only support up to 3000 to 5000 concurrent connections, PHP is even lower (See: 3.2.1 High performance I/O Framework, 3.3.1 Web Framework, and 3.3.2 Typical Web Use Cases).

## Consistent HAC and HPC across multiple active IDC

Distributed high availability and high performance cluster with strong consistency assurance (anti split-brain): Thanks for the patented nano-SOA large-scale distributed architecture. We could maintain a high cohesion, low coupling design under the premise of keeping the single node performance to far beyond the traditional SOA architecture, while simplifying the cluster deployment, and improve the cluster maintainability (See: 5.4 nSOA - libapidbc, 5.4.1 SOA vs. AIO, and 5.4.2 nSOA Architecture).

BaiY Port Switch Service (BYPSS): BYPSS is designed for providing a high available, strongly consistent and high performance distributed coordination and message dispatching service based on the quorum algorithm. It can be used to provide services such as fault detection, service election, service discovery, distributed lock, and other distributed coordination functionalities, it also integrates a message routing service.

Thanks for our patented algorithm, we eliminate the network broadcast, disk IO and other major costs within the traditional Paxos / Raft algorithms. We have also done a lot of other optimizations, such as: support for batch mode, use the concurrent hash table and high performance IO component. These optimizations allow BYPSS to support ultra-large-scale computing clusters consist with millions nodes and trillions ports in a limited (both for throughput and latency) cross-IDC network environment (See: 5.4.3 Port Switch Service).

Scaling out nodes across multiple active IDC and keeping strong consistency guarantee is the key technology of modern high-performance and high-availability cluster, which is also recognized as the main difficulty in the industry. As examples: September 4, 2018, the cooling system failure of a Microsoft data center in South Central US caused Office, Active Directory, Visual Studio and other services to be offline for nearly 10 hours; August 20, 2015 Google GCE service interrupted for 12 hours and permanently lost part of data; May 27, 2015, July 22, 2016 and Dec 5, 2019 Alipay interrupted for several hours; As well as the July 22, 2013 and Mar 29, 2023 WeChat service interruption for several hours, and etc. These major accidents are due to product not implement the multiple active IDC architecture correctly, so a single IDC failure led to full service off-line.

We have over 10 years of experience in the distributed computing field. We hold the related distributed architecture and algorithms which protected by a number of national and international patents. Thanks to these leading distributed clustering algorithms and architectures, we can deploy multiple active IDC cluster with strong consistent, high availability, and high-performance guarantee easily. We have been implemented the truly multiple active IDC cluster on full range of our products,

providing our customers with unparalleled data reliability and service availability assurance.

## Distributed coordination service

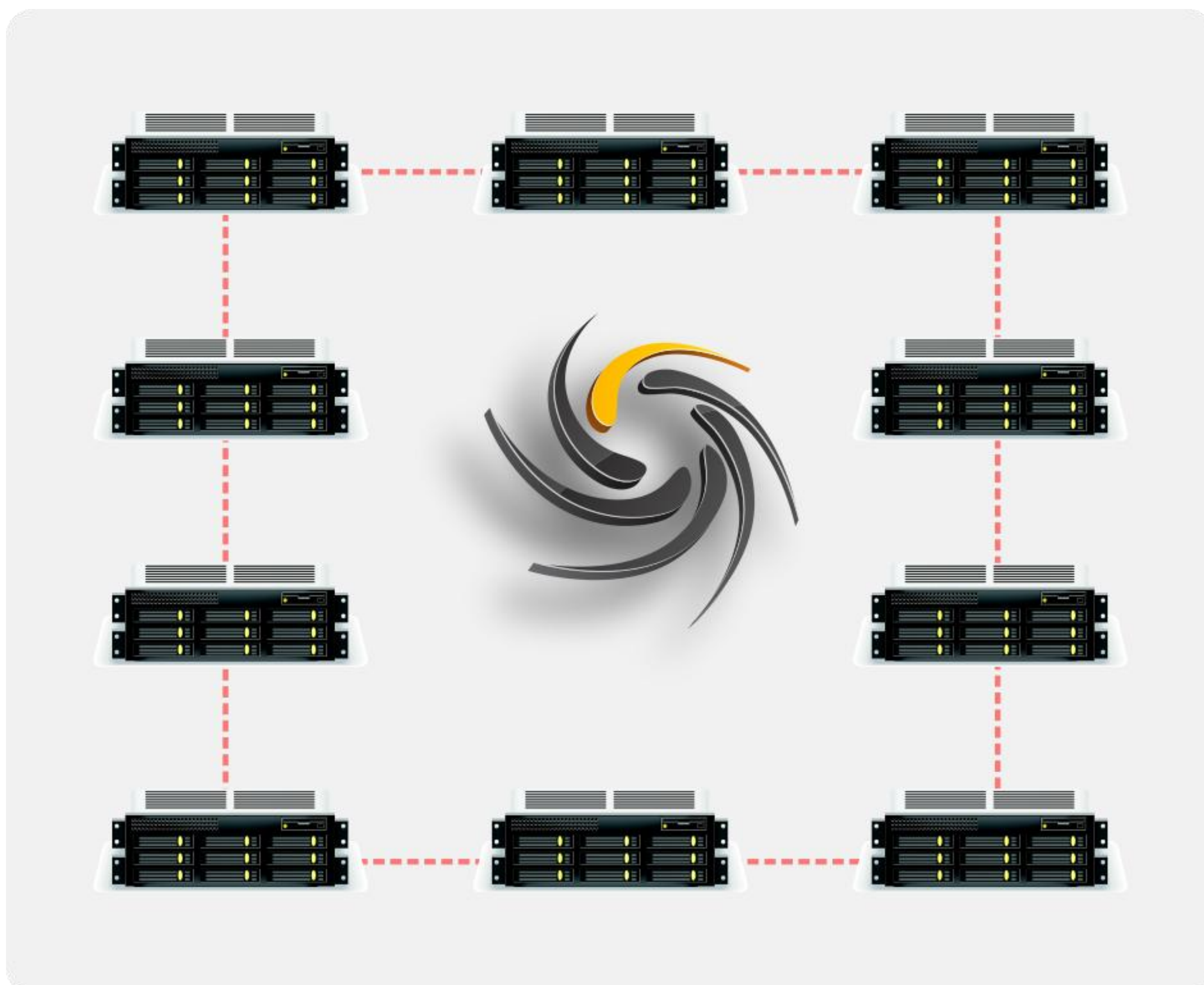


Figure 1

Distributed coordination services provide functions such as service discovery, service election, fault detection, failover, failback, distributed lock, task scheduling, message routing and message dispatching.

The distributed coordination service is the brain of a distributed cluster that is responsible for coordinating all the server nodes in the cluster. Make distributed clusters into an organic whole that works effectively and consistently, making it a linear scalable high performance (HPC) and high availability (HAC) distributed clustering system.

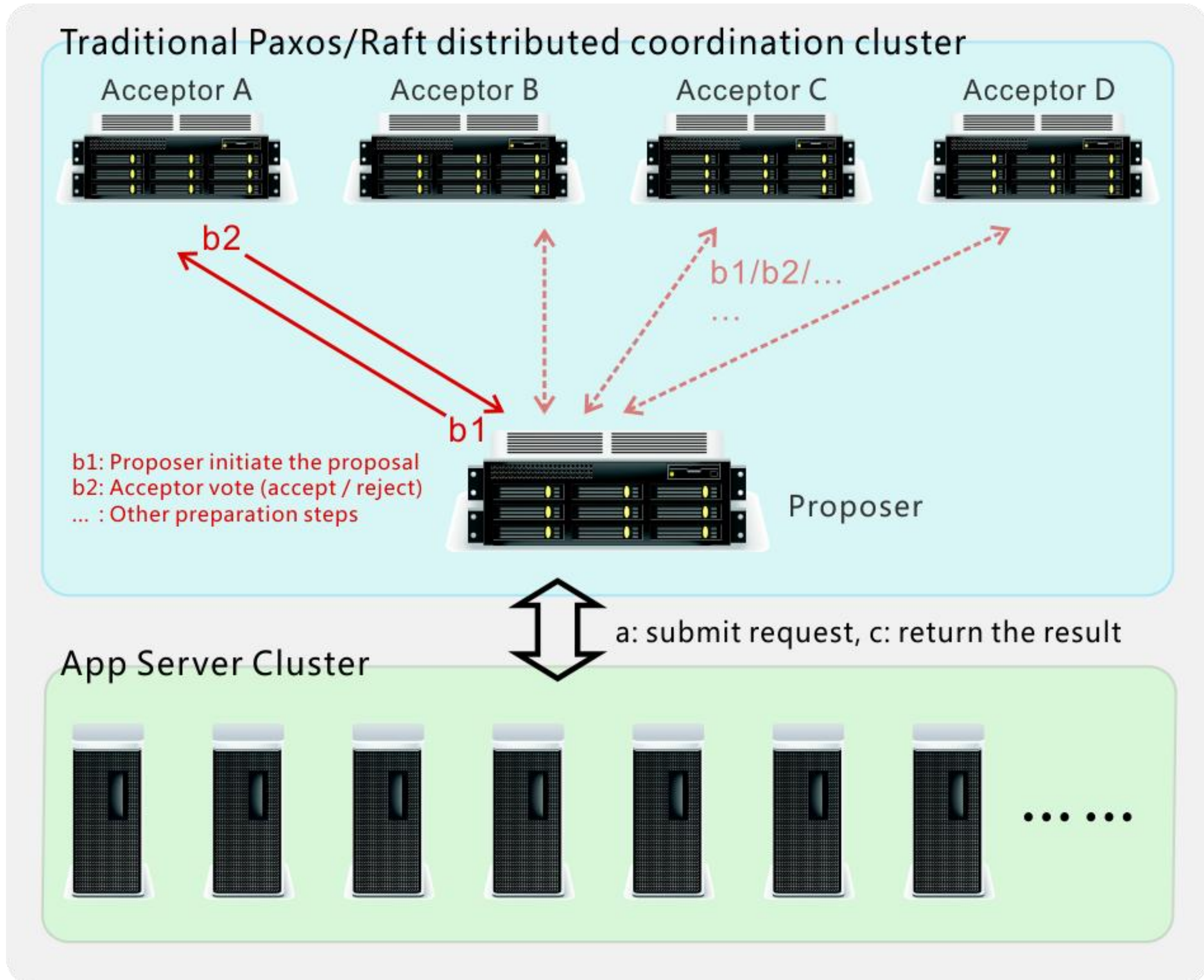



Figure 2

The traditional Paxos / Raft distributed coordination algorithm initiates voting for each request, generating at least 2 to 4 broadcasts (b1, b2...) and multiple disk IO. Making it highly demanding on network throughput and communication latency, and cannot be deployed across multiple data centers.

Our patent algorithm completely eliminated these overheads. Thus greatly reducing the network load, significantly improve the overall efficiency. And makes it easy to deploy clusters across multiple data centers.

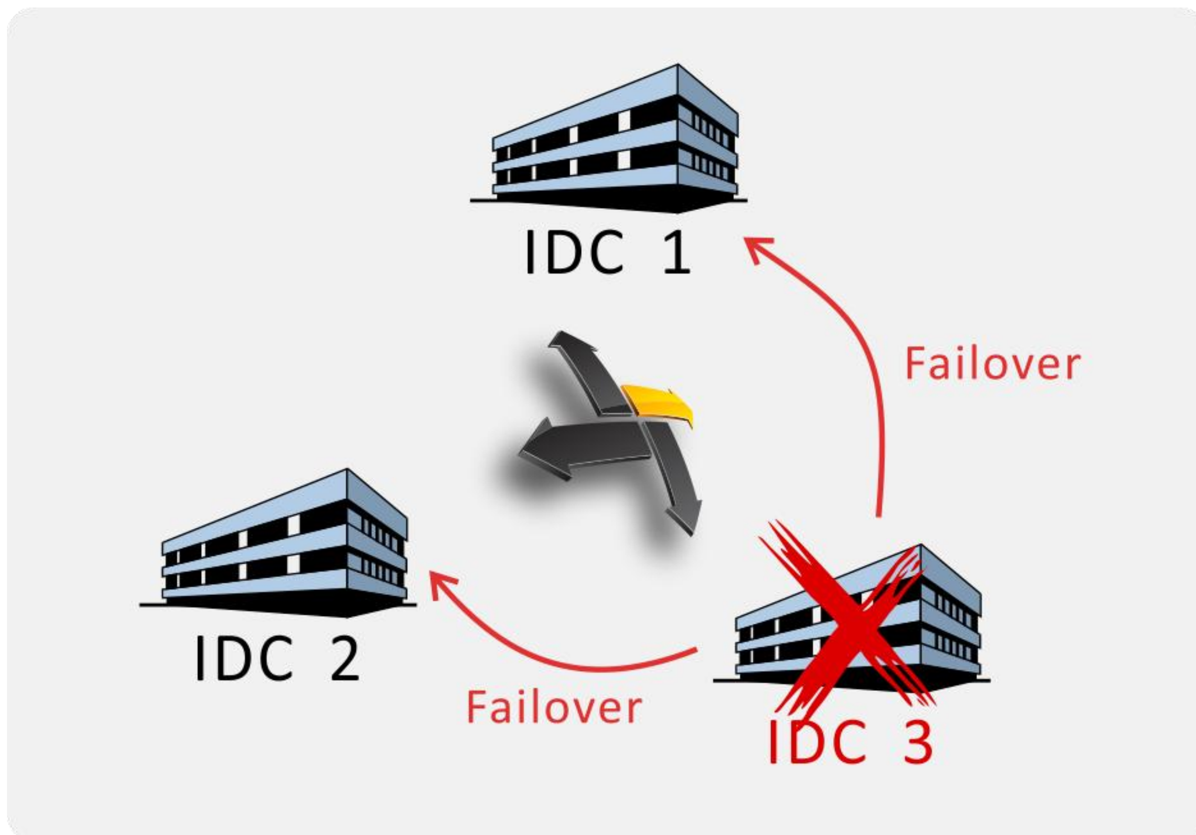



Figure 3

Based on our unique distributed coordination technology, the high performance, strong consistency cluster across multiple data centers can be implemented easily. Fault detection and failover can be done in milliseconds. The system is still available even if the entire data center is offline. We also providing a strong consistency guarantee: even if there is a network partition, it will not appear split brain and other data inconsistencies. For example:

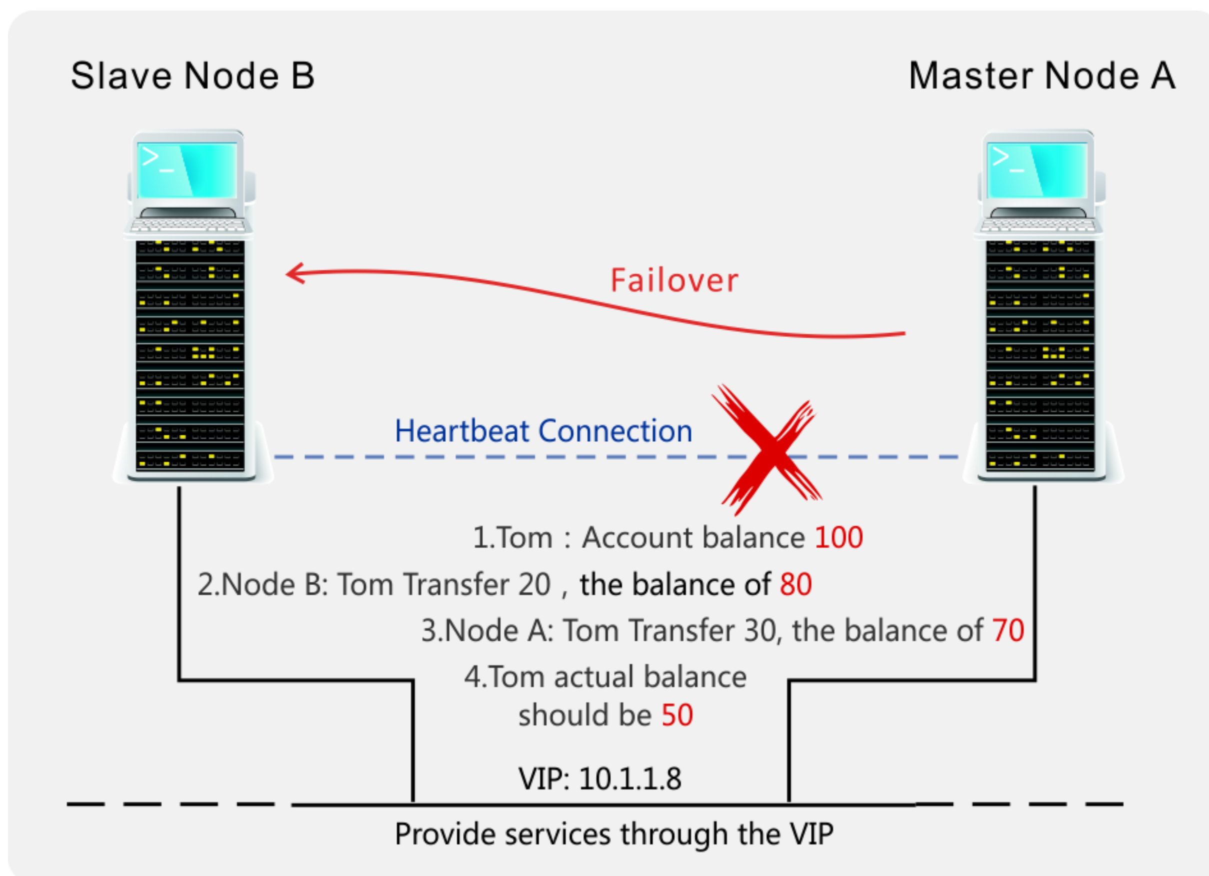



Figure 4

In the traditional dual fault tolerance scheme, the slave node automatically promotes itself as the master node after losing the heartbeat signal and continues to provide services to achieve high availability. In this case, split brain problem occurs when both the master and slave nodes are normal, but the heartbeat connection is accidentally disconnected (network partition). As shown in Figure 4: At this time, node A and B both think that the other party is offline. As a result, both nodes upgrade themselves to the master node and provide the same service, respectively. This will result in inconsistent data that is difficult to recover.

Our BYPSS service provides the same level of consistency as the traditional Paxos / Raft distributed algorithm, fundamentally eliminates the occurrence of inconsistencies such as split brain.

Similarly: ICBC, Alipay and other services are also have its own remote disaster recovery solutions (Alipay: Hangzhou → Shenzhen, ICBC: Shanghai → Beijing). However, in their remote disaster recovery schemes, there is no paxos and other distributed coordination algorithms between the two data centers, so strong consistency cannot be achieved.

For example, a transfer transaction that has been successfully completed at Alipay may take several minutes or even hours to be synchronized from the Hangzhou main IDC to the disaster recovery center in Shenzhen. When the Hangzhou main data center offline, all of these non-synchronized

transactions are lost if they switch to the disaster recovery center, leads a large number of inconsistencies. Therefore, ICBC, Alipay and other institutions would rather stop the service for hours or even longer, and would not be willing to switch to the disaster recovery center in the major accidents of the main IDC. Operators will consider turning their business into a disaster recovery center only after a devastating accident such as a fire in the main data center.

Therefore, the remote disaster recovery schemes and our strong consistency, high availability, anti-split brain multi-IDC solution is essentially different.

In addition, Paxos / Raft cannot guarantee the strong consistency of data during the process of simultaneous failure and recovery of more than half of the nodes, and may cause inconsistencies such as phantom reading (For example, in a three-node cluster, node A goes offline due to power failure, and after one hour, nodes B and C go offline because of disk failure. At this point, node A resumes power supply and goes online again, and then the administrator replaces the disks of nodes B and C and restores them to go online. At this point, the data modification of the entire cluster in the last hour will be lost, and the cluster will fall back to the state before the A node goes offline at 1 hour ago). BYPSS fundamentally avoids such problems, so BYPSS has a stronger consistency guarantee than Paxos / Raft.

Due to the elimination of a large number of broadcast and distributed disk IO and other high-cost operation brought by the Paxos / Raft algorithm. Making BYPSS distributed coordination component also provides more excellent features in addition to the above advantages:

**Bulk operation**: Allows each network packet to contain a large number of distributed coordination requests at the same time. Network utilization greatly improved, from the previous less than 5% to more than 99%. Similar to the difference between a flight only can transport one passenger each time, and another one can transport full of passengers. In the actual test, in a single Gigabit network card, BYPSS can achieve 4 million requests per second performance. In the dual-port 10 Gigabit network card (currently the mainstream data center configuration), the throughput of 80 million requests per second can be reached. There is a huge improvement compared to the Paxos / Raft cluster which performance is usually less than 200 requests per second (restricted by its large number of disk IO and network broadcast).

**Large capacity**: usually every 10GB of memory can support at least 100 million ports. In a 1U-size entry-level PC Server with 64 DIMM slots (8TB), it can support at least 80 billion objects at the same time. In a 32U large PC server (96TB), it can support about 1 trillion distributed coordinating objects. In contrast, traditional Paxos / Raft algorithms can only effectively manage and schedule hundreds of thousands of objects due to their limitations.

The essence of the problem is that in algorithms such as Paxos / Raft, more than 99.99% of the cost is spent on broadcast (voting) and disk writes. The purpose of these behaviors is to ensure the reliability of the data (data needs to be stored on persistent devices on most nodes simultaneously).

However, distributed coordination functions such as service discovery, service election, fault detection, failover, failback, distributed lock, and task scheduling are all temporary data that have no long-term preservation value. So it makes no sense to spend more than 99.99% of your effort to persist multiple copies of them - even if there is a rare disaster that causes the main node to go offline, we can regenerate the data in an instant with great efficiency.

It’s as if tom bought a car that has an additional insurance service. The terms are: In the event of a fatal traffic accident, it provides a back in time mechanism that takes tom back to the moment before the accident, so he can avoid this accident. Of course, such a powerful service is certainly expensive, and it probably needs to prepay all the wealth tom’s family can get in the next three generations. And these prepaid service fees are not deductible even if he has never had a fatal traffic accident with this car. Such an expensive service that is unlikely to be used in a lifetime (what percentage of people will have fatal traffic accidents? Not to mention that it can only happen on the specific car), even if it does happen, this huge price is hard to say is worth it?

And we offer a different kind of additional service for our cars: Although there is no back in time function, our service can instant and intact resurrect all the victims in the same place after the fatal accident. The most important thing is that the service will not charge any fees in advance. Tom only needs to pay a regenerative technology service fee equivalent to his half-month salary after each such disaster.

In summary, our patented distributed coordination algorithm providing strong consistency and high availability assurance at the same level as the traditional Paxos / Raft algorithm. At the same time, it also greatly reduces the system's dependence on the network and disk IO, and significantly improves the overall system performance and capacity. This is a significant improvement in the high availability (HAC) and high performance (HPC), large-scale, strongly consistent distributed clusters.

For a further description of the BYPSS service, see: 5.4.3 Port Switch Service.

## Efficient high-strength cryptographic components

This includes basic functions such as public-key algorithms, symmetric encryption algorithms, data encoding and decoding, hash and message authentication algorithms, data compression algorithms, and etc (See: 4. Cross-platform Cryptographic Library - libcrypto, and 4.1 The Cryptographic Algorithm Module - algorithm). In addition, the application platform also provides a number of highly abstract advanced components, such as:

The Virtual File System (VFS) supports data encryption and compression on-the-fly. VFS supports dozens of strong encryption algorithms, including AES (128/256), TwoFish, etc., optimized using AES-NI, SSE4 and other assembly instruction set, with high efficiency. We use this component to provide on-the-fly data compression and strong encryption protection for the whole database and

configuration categories in our products like BlueWhale, WhiteDolphin, ZhiYeJing.com and so on. It also includes strong cryptographic communication protection components based on Public Key Infrastructure (PKI) and etc. (See: 4.2 The Common Facilities Module - facility)

In recent years, security issues frequently occur. Well-known enterprises such as Amazon, Wal-Mart, Yahoo, Linkedin, OpenAI (ChatGPT), Sony, JP Morgan Chase, UPS, eBay, dj.com, Alipay, ctrip.com, 12306, Netease, CSDN, China Life Insurance, as well as major hotel groups (such as HOME INNS, HANTING INNS, Jinjiang Hotels, InterContinental, Sheraton, Marriott, etc.) are frequently reported a large number of users information disclosure and others serious security incidents, security protection demand immediate attention.

All of our databases (the entire dataset) and configuration data are stored in our self-developed VFS which supports on-the-fly data compression and strong encryption, provides comprehensive protection for our customers.

In addition, our unique High Performance Network Security Tunnel (BYST) component provides high-performance, high-throughput and high-network utilization VPN services while maintaining communication security, further help customs improve the performance and security of network communication in local, metro and wide area networks (see:5.6 Secure Tunnel Service (BYST)).

Strong encryption algorithms based on industry standards ensure that even if a supercomputer with one trillion trillions of key cracking attempts per second is made in the future, it will still take an average of 540 billion years to crack a key.

## Data query engine

The application platform also includes a query engine. Its ability is better than SQL language. Having own query engine gives us the flexibility to switch between RDBMSs such as MySQL, MS SQL Server, Oracle, DB2, SQLite, and NoSQL databases like MongoDB and Cassandra. In addition to making applications database-independent, the query engine also provides a variety of advanced characteristics that are not supported by SQL language, such as ARE (Advanced Regular Expressions) query with support for Unicode charset, join query with support for nested tables, mix query of business data and configuration data, virtual field query, and other customized queries.

The query engine was implemented using C/C++, and its hotspot codes were optimized using assembly language for mainstream hardware platforms. 13 million times of evaluation of expressions per second can be achieved on a ThinkPad W510 notebook (having 4 cores and 8 threads @1.6GHz) produced in 2010, using a single core and a single thread only (See: 3.3 Common Facilities Module - facility, and 5.4 nSOA - libapidbc).

# More...

The above only refers to a few highlights within the thousands of components in BaiY Application Platform. A more complete description is given below.

# Revision History

| Version | Date | Description | Changed by | Reviewed by |
|---|---|---|---|---|
| 1.0 | 2007-07-21 | First edition migrated from the old introduction document | Bai Yang | |
| 1.1 | 2007-08-09 | Updated document structure (changed section 6.3.3 to section 6.4) and corrected minor wording errors | Bai Yang | |
| 1.2 | 2008-01-04 | Added description about Web application extensions | Bai Yang | |
| 1.3 | 2008-03-19 | Added support for bz2 algorithm | Bai Yang | |
| 1.4 | 2009-12-02 | Updated AIO framework and added support for SCGI | Bai Yang | |
| 1.5 | 2010-04-27 | Restructured the document; added support for HTTP | Bai Yang | |
| 1.6 | 2010-06-13 | Added Web framework comparison table | Bai Yang | |
| 1.7 | 2010-07-06 | Added description about HTTP Pipelining | Bai Yang | |
| 1.8 | 2010-08-25 | Added new components such as LRU Cache, according to recent adjustment to the low-level library | Bai Yang | |
| 1.9 | 2010-08-27 | Added description about a typical working model of Web application nodes | Bai Yang | |
| 1.10 | 2010-12-10 | Added support for /dev/poll and pollset to the AIO framework | Bai Yang | |
| 1.11 | 2011-06-07 | Annual update; added description about the development kit for the CConfig component | Bai Yang | |
| 1.12 | 2012-03-15 | Annual update; added description about components such as variant data type | Bai Yang | |
| 1.13 | 2012-04-12 | Added LZ4 data compression algorithm | Bai Yang | |
| 2.0 | 2012-04-17 | Restructured the document; moved description about the cross-platform GUI framework and the cross-platform audio processing library to chapter 6. Interface, Media and Other Tools; added chapter 5. Data Processing Tools | Bai Yang | |
| 2.1 | 2012-04-19 | Added new sections such as 6.3 CConfig Language Binding Component and 6.4 JavaScript Tools Library - libbaiy | Bai Yang | |
| 2.2 | 2012-05-07 | Added description about generic query conditions and QLI (Query Language Interpreter) | Bai Yang | |
| 2.3 | 2012-05-19 | Added description about Search Helper | Bai Yang | |
| 2.4 | 2012-06-09 | Added JavaScript Keyword Tree container into | Bai Yang | |

| Version | Date | Description | Changed by | Reviewed by |
|---|---|---|---|---|
| | | libbaiy | | |
| 2.5 | 2012-12-02 | Updated figures and page layout | Bai Yang | |
| 2.6 | 2013-01-16 | Annual update; corrected an invalid reference | Bai Yang | |
| 2.7 | 2013-03-11 | Updated description about atomic and Memory Barrier | Bai Yang | |
| 2.8 | 2013-03-16 | Added support for SHA-3 | Bai Yang | Huasong Liu |
| 2.9 | 2013-05-22 | Corrected a typo | Bai Yang | |
| 2.10 | 2014-01-12 | Annual update; added the cross-platform mechanism used for tracing function call stack | Bai Yang | |
| 2.11 | 2014-02-14 | Added new components (e.g. Message Dispatcher) to libbaiy.js | Bai Yang | |
| 2.12 | 2014-06-07 | Added the Task Queue component to libbaiy.js | Bai Yang | |
| 2.13 | 2015-02-07 | Added description about the libapidbc library | Bai Yang | |
| 2.14 | 2015-03-23 | Added further discussions about distributed caching, NoSQL and NewSQL into the section Database and memcached Services | Bai Yang | |
| 2.15 | 2015-05-06 | Rewording; corrected some typos; updated some data | Bai Yang | |
| 2.16 | 2015-05-30 | Small updates | Bai Yang | |
| 2.17 | 2015-07-24 | Updated some sections | Bai Yang | |
| 2.18 | 2015-11-04 | Small updates | Bai Yang | |
| 2.19 | 2015-12-21 | Updated some data | Bai Yang | |
| 2.20 | 2016-1-22 | Small updates | Bai Yang | |
| 2.21 | 2016-4-13 | Updated some sections | Bai Yang | |
| 2.22 | 2016-7-15 | Updated some data | Bai Yang | |
| 2.23 | 2016-10-04 | Add support for CRC32-C, ChaCha, and BLAKE2 algorithms | Bai Yang | |
| 2.24 | 2016-12-06 | Add “A Quick Brief” section | Bai Yang | |
| 2.25 | 2016-12-16 | Add “BYPSS based High performance cluster” section | Bai Yang | |
| 2.26 | 2017-04-03 | Added description about the diff function of the CConfig component | Bai Yang | |
| 2.27 | 2017-08-06 | Added “Distributed coordination service” section | Bai Yang | |
| 2.28 | 2017-10-07 | Added “Distributed FTS Service” section | Bai Yang | |
| 2.29 | 2018-01-13 | Small updates | Bai Yang | |
| 2.30 | 2018-03-26 | Change "µSOA" to "nano-SOA" to avoid confusion with "micro-SOA" | Bai Yang | |
| 2.31 | 2018-05-22 | Add support for ARIA, Kalyna, Simon, Speck, SM4, ThreeFish, SipHash, Poly1305, SM3 algorithms | Bai Yang | |

| Version | Date | Description | Changed by | Reviewed by |
|---|---|---|---|---|
| 2.32 | 2019-01-12 | Added description for BYDMQ distributed message queue component | Bai Yang | |
| 2.33 | 2019-02-13 | Add support for CHAM, HIGHT, LEA, SIMECK, Rabbit, and HC algorithms | Bai Yang | |
| 2.34 | 2019-03-23 | Add support for SHAKE algorithm | Bai Yang | |
| 2.35 | 2019-04-27 | Added description for BYST secure tunnel component | Bai Yang | |
| 2.36 | 2020-08-18 | Added the glance of our international patents | Bai Yang | |
| 2.37 | 2021-03-03 | Added some supplementary descriptions related to BYST | Bai Yang | |
| 2.38 | 2021-10-16 | Added support for LSH algorithm | Bai Yang | |
| 2.39 | 2021-10-26 | Small updates | Bai Yang | |
| 2.40 | 2021-10-31 | Add a summary description for 5.6 Secure Tunnel Service (BYST) | Bai Yang | |
| 2.41 | 2022-02-20 | Clarified several BYPSS/BYDMQ related details; fixed individual typos | Bai Yang | |
| 2.42 | 2023-02-25 | Update typical customer list and patent list | Bai Yang | |
| 2.43 | 2023-02-38 | Add millisecond level failover support for BYPSS and BYDMQ | Bai Yang | |

# Contents

INTRODUCTION ..... I
A QUICK BRIEF ..... III
High-performance IO server components ..... III
Consistent HAC and HPC across multiple active IDC ..... IV
Efficient high-strength cryptographic components ..... X
Data query engine ..... XI
More... ..... XII

REVISION HISTORY ..... XIII

CONTENTS ..... XVI

1. OVERVIEW OF THE APPLICATION PLATFORM ..... 1

2. ARCHITECTURE OF THE PLATFORM ..... 2

3. CROSS-PLATFORM INFRASTRUCTURE - LIBUTILITIS ..... 4
3.1 THE BASE MODULE - BASE ..... 5
3.1.1 Bottom Layer of the Base Module ..... 5
3.1.2 Interface Layer of the Base Module ..... 6
3.2 SYSTEM UTILITIES MODULE—SYSUTIL ..... 10
3.2.1 High performance I/O Framework ..... 14
3.3 COMMON FACILITIES MODULE - FACILITY ..... 16
3.3.1 Web Framework ..... 24
3.3.2 Typical Web Use Cases ..... 30
3.3.3 FastCGI? SCGI? HTTP! ..... 41

4. CROSS-PLATFORM CRYPTOGRAPHIC LIBRARY - LIBCRYPTO ..... 42
4.1 THE CRYPTOGRAPHIC ALGORITHM MODULE - ALGORITHM ..... 42
4.1.1 Block Cipher Algorithms ..... 43
4.1.2 Stream Cipher Algorithms ..... 45
4.1.3 Public Key Algorithms ..... 46
4.1.4 Hash Algorithms ..... 46
4.1.5 Message Authentication Algorithms ..... 47
4.1.6 Data Compression Algorithms ..... 47
4.1.7 Data Encode/Decode Algorithms ..... 47
4.1.8 Random Number Generator Algorithm ..... 47
4.2 THE COMMON FACILITIES MODULE - FACILITY ..... 48

5. DATA PROCESSING TOOLS ..... 49

5.1 REPORT GENERATION LIBRARY - LIBREPORT ........ 49
5.2 ODBC ENCAPSULATION LIBRARY - LIBODBC_CPP ........ 49
5.3 SQLITE ENCAPSULATION LIBRARY - LIBSQLITE_CPP ........ 51
5.4 NSOA - LIBAPIDBC ........ 52
5.4.1 SOA vs. AIO ........ 54
5.4.2 nSOA Architecture ........ 56
5.4.3 Port Switch Service (BYPSS) ........ 57
5.4.4 Distributed Message Queue Service (BYDMQ) ........ 79
5.5 DISTRIBUTED FULL TEXT SEARCH (FTS) SERVICE ........ 85
5.6 SECURE TUNNEL SERVICE (BYST) ........ 86

**6. INTERFACE, MEDIA AND OTHER TOOLS ........ 90**
6.1 CROSS-PLATFORM AUDIO I/O LIBRARY - LIBAUDIOIO ........ 90
6.2 CROSS-PLATFORM I18N GUI FRAMEWORK - LIBMLGUI ........ 92
6.2.1 File System Extension ........ 94
6.2.2 I18N Components Library ........ 94
6.2.3 Quick Help Framework ........ 96
6.2.4 Universal Graphic Controls ........ 98
6.3 CCONFIG LANGUAGE BINDING COMPONENT ........ 102
6.4 JAVASCRIPT TOOLS LIBRARY - LIBBAIY ........ 102
6.4.1 Functions Library ........ 103
6.4.2 User Interface Library ........ 104

**7. ERROR PROCESSING MECHANISM ........ 106**

# 1. Overview of the Application Platform

The application platform is the foundation on which products are built as well as the interface for communication between a product and the operating system. It not only encapsulates all the functions associated with the operating system but also provides a collection of common tools. As an important generic component, the application platform plays a key role in quick development of high-quality and cross-platform applications.

The application platform provides a number of common features for the other components. These features include:

- **Cross-platform and low-level support:** encapsulates all operations associated with the operating system, such as Semaphore, atomic operations, shared memory/file mapping, thread, network operations (Socket), file management, service control, registry access, Inter-process communication (IPC), server framework and etc. This is the key component for achieving a cross-platform and multi-platform system.
- **Common features**: include user authentication and authorization, strong encryption algorithms based on the PKI infrastructure, common network protocols, binary and charset encoding conversion, automated script engine, form handling, data compression, task management, log, audio I/O, audio encoder/decoder, audio effects, HTTP protocol, Web application extensions and etc.
- **Cross-platform data processing functionalities:** include a cross-platform report generation library with support for Excel and HTML formats, a database component with support for ODBC and ISO SQL/CLI interfaces, and the SQLite database engine encapsulation.
- **Distributed Computing**: Came up with the nano-SOA architecture and corresponding supporting components, include: a cross-platform API registration and dispatching framework, a generic plugin interface. Also, it offers database connectors (DBC) for implementing strong encryption, data sharding and CAS-based optimistic locking algorithm, and has implemented common DBC plugins. In addition, it has defined a high available, strongly consistent and high performance distributed coordination and message dispatch service.
- **Cross-platform GUI framework:** encapsulates system functions such as windows, controls and the system message mechanism, and provides a unified and platform-independent framework for GUI applications.
- **Platform-independent support for Internationalization (I18N):** provides a platform-independent multi-language environment for components such as the report generator and GUI framework.

# 2. Architecture of the Platform

The application platform is the foundation on which all the other components depend. It offers platform-independent abstraction between software developers and the runtime environment (hardware platform, compiler environment and the operating system), and also provides developers with a set of cross-platform components and frameworks that are reliable, efficient and easy-to-use.

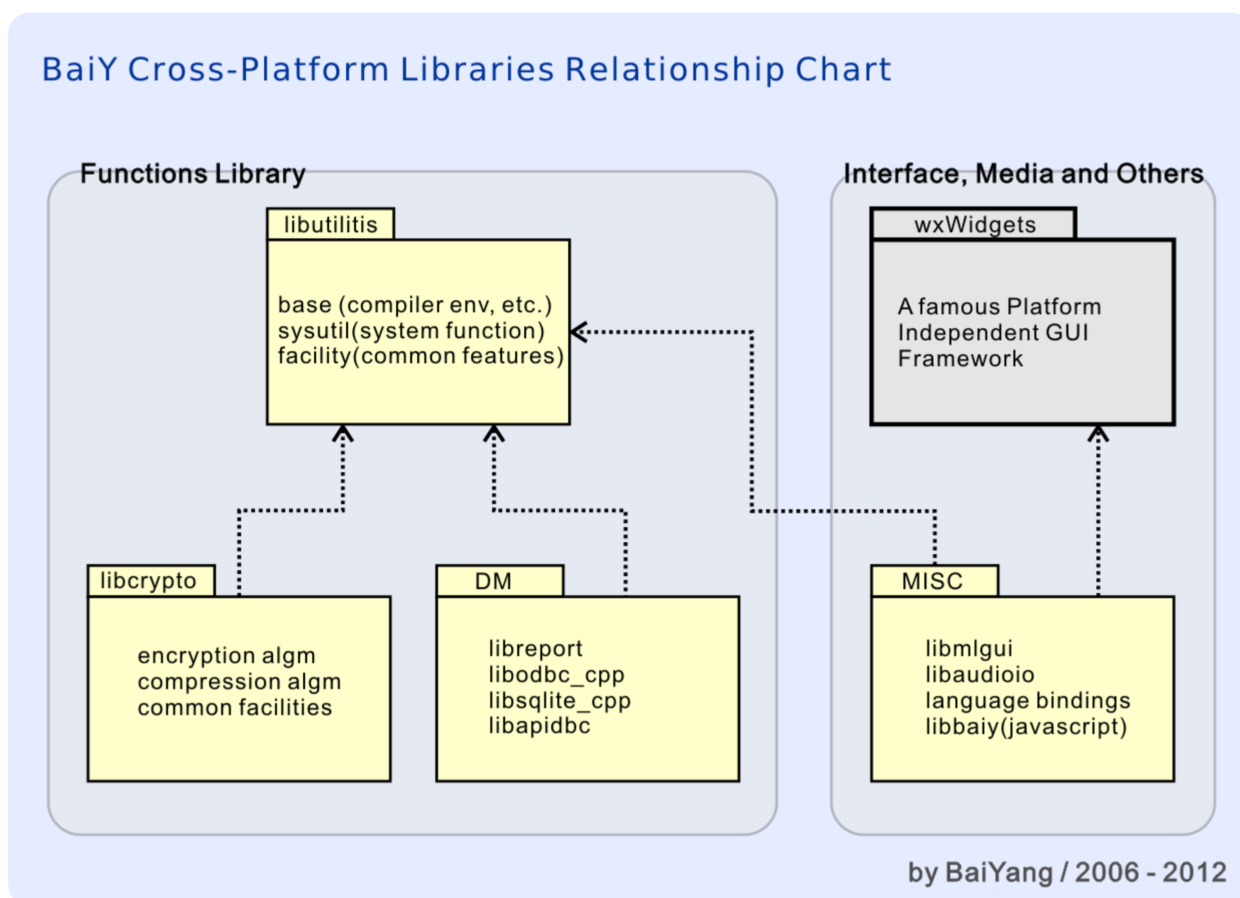



Figure 5

As illustrated in Figure 5, the application platform consists of the following correlated components:

* **libutilitis:** encapsulates all fundamental functions associated with the hardware platform, compiler environment and the operating system, and provides common features and general frameworks.
* **libcrypto:** this component was implemented based on libutilitis and third-party cryptographic and compression libraries. It encapsulates all cryptographic, compression and data encoding algorithms. Relying on this encapsulation, the libcrypto component has implemented a collection of common features.

* Data processing functions (DM):

  - **libreport:** this is the cross-platform report generation library implemented on the basis of libutilitis. It supports a range of file formats like Excel 2.0 (BIFF), Excel XP (ExcelML), Excel 2007 (xlsx) and HTML, and offers features including customizable templates and variables, chart generator, and I18N capabilities.

  - **libodbc_cpp:** this library was implemented based on libutilitis. It is a C++ encapsulation of ODBC/ISO CLI interfaces, and supports features like prepared statement, parameter binding, zero-copy result set retrieval (result set field pre-binding), and etc.

  - **libsqlite_cpp:** this library was implemented based on libutilitis, libcrypto and the SQLite engine. It is a C++ encapsulation of SQLite database engine, and supports features like prepared statement, parameter binding, and VFS-based whole database encryption using strong cryptographic algorithms.

  - **libapidbc**: this library was implemented based on libutilitis, libcrypto, libodbc_cpp and libsqlite_cpp. It has defined a set of cross-platform interfaces for common plugins, and has further implemented a collection of common database connectors. Furthermore, it has defined a complete set of tools used for managing API registration/dispatching and requests queuing for communications among the functional plugins.

* User Interface and multi-media libraries:

  - **libaudioio:** this library was implemented based on libutilitis. It provides a platform-independent audio I/O mechanism, encoder and decoder for various audio formats, and some general filters. Also, the library provides some common tools like the audio playing/recording tool.

  - **libmlgui:** this library was implemented based on libutilitis and the wxWidgets framework. It provides a complete set of platform-independent I18N GUI frameworks and related common features.

The above mentioned components are discussed in more details in the following sections.

# 3. Cross-platform Infrastructure - libutilitis

As illustrated in the system architecture, the application platform is at the bottom layer of the entire product, and libutilitis is the infrastructure for the platform. The main function of libutilitis is to encapsulate all the details associated with the hardware platform, compiler environment and the operating system, and to offer a set of easy-to-use, consistent and platform-independent APIs. Based on these functionalities, the libutilitis library also provides some common tools and frameworks.

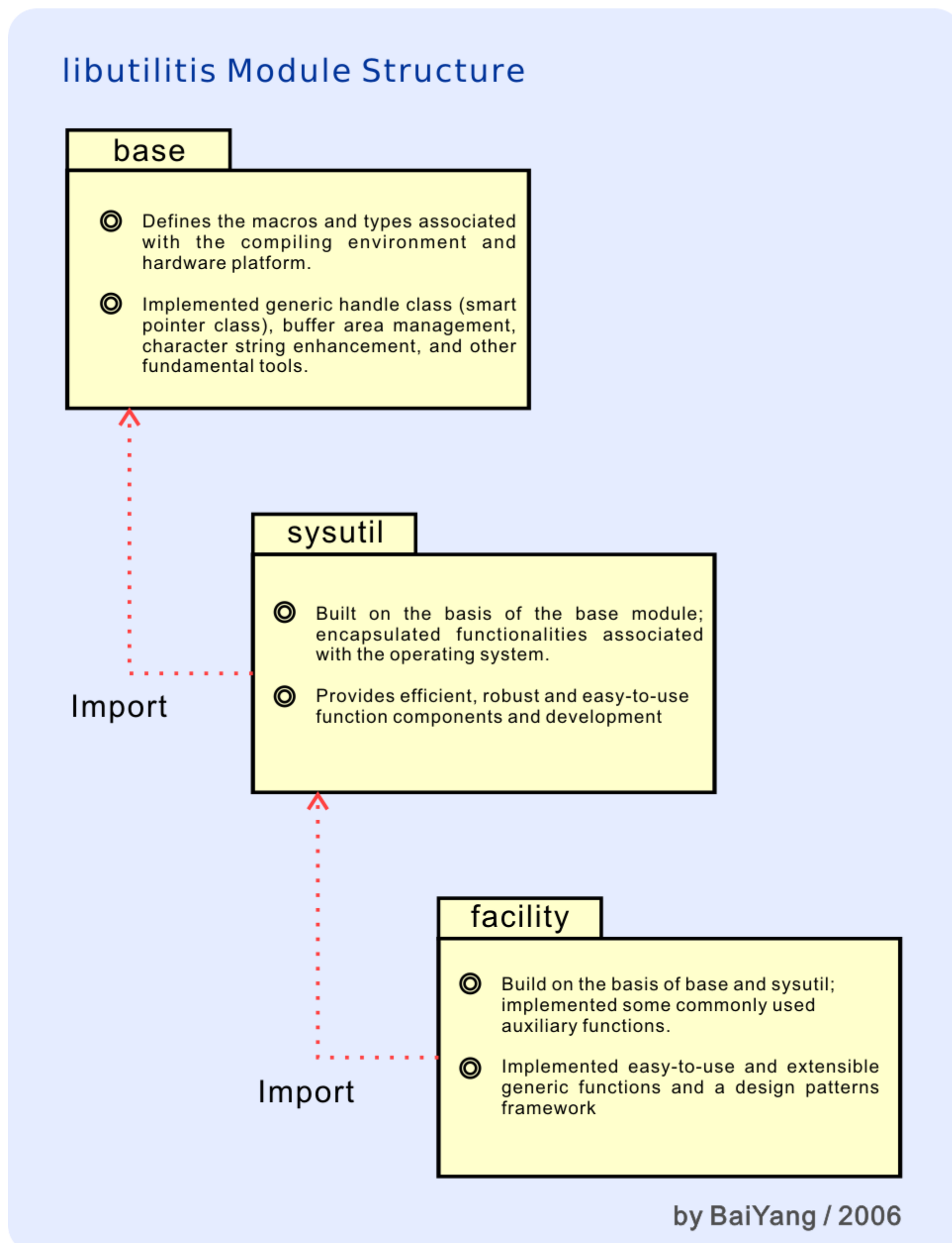



Figure 6

As illustrated in Figure 6, libutilitis is composed of three interdependent modules, which are discussed in the following sections.

## 3.1 The Base Module - base

The base module encapsulates the details associated with the hardware platform and the compiler environment, and provides universal fundamental tools for the sysutil and the facilities module.

When we start to develop a software module, we always want to provide users with the fullest feature set, the easiest to use interface, and robust, agile and elegant components without compromising efficiency. Unfortunately, something that can satisfy all these conditions is out of the current human capabilities. In most cases, we have to painfully compromise some aspects.

Therefore, it is necessary to determine the importance level of each factor before we start to design. This can help to build consistent and easy-to-understand interfaces. In terms of the base module, the factors being considered before designing are as follows (in descending order of importance):

1. **Reliability/robustness and correctness**: to either execute a task or to inform the user with an error message explicitly.
2. **Efficiency**: to improve efficiency to the greatest degree possible, on the premise of ensuring reliability and correctness.
3. **Usability:** to make interfaces easy-to-use and easy-to-understand as possible; to provide obvious prompt in places where unexpected results may occur.
4. **Portability**: to minimize, to the greatest degree possible, the effort required for porting the software across underlayer platforms.
5. **Maintainability and extensibility:** to define a clear inner structure, and to keep system architecture as easy to extend and maintain as possible, on the premise of guaranteeing the above factors.

The base module can also be divided into two parts according to their relevance with the implementation details: bottom layer and interface layer.

### 3.1.1 Bottom Layer of the Base Module

The bottom layer of the base module handles issues related to hardware characteristics and the

compiler environment. To ensure maximum execution efficiency, the bottom layer is completely composed of complex macro magics and dozens of typedef.

The bottom layer is the fundamental part of the entire library. All logic judgments are achieved via plenty of macro magic, which are difficult to use and maintain, though they have completely eliminated runtime consumption. Users rarely need to use these macros directly, and also should avoid using them (except for ideographic macros) when possible.

Similar with many of configurable libraries, users can adjust various function and behaviour options available in libutilitis by defining or changing some on/off macros before compiling it.

### 3.1.2 Interface Layer of the Base Module

The interface layer is an encapsulation of the low-level implementations, and provides users with consistent and easy-to-use interfaces. For example,

* Provides transparent INT64 Integer simulation in compiler environments that do not support 64-bit Integer.

* Provides acquire, release and no barrier semantics atomic operations for 32-bit, 64-bit and pointer data types. It is preferred to implement atomic operations via intrinsic/built-in methods provided by the compiler and the inline assembly language. Atomic operations are currently supported on platforms like x86/x64, IA64, ARM, RISC-V, POWER, MIPS, and SPARC. For platforms that do not support hardware-level atomic operations, libutilitis can offer atomic support via operating system API or by simulating it using a set of mutex with hash collections optimization. If the target platform is an embedded environment without thread support, all atomic operations will be degraded as the most efficient and unprotected dummy implementation.

* Provides read & write, read only, and writer only memory barrier operations. Similar with atomic operations support, it is preferred to implement memory barrier operations via intrinsic/built-in methods provided by the compiler and the inline assembly language. The platforms that support atomic operations also support hardware-level memory barrier operations. For platforms that do not support the latter, libutilitis provides a simulation using mutex.

  For more details about atomic and memory barrier operations, refer to section "Atomic Operations and volatile Keywords" in my document C++ Coding Guidelines (Chinese only).

* Provides plenty of identifier macros associated with the platform or compiler. For

example, force inline instructions, DLL symbol export instructions, inline assembly instructions, the current hardware platform, compiler type, and compiler supported characteristics (e.g., whether the compiler supports template embedding and hash container), etc.

* Provides a series of optimization instructions associated with the low-level platform, such as branch prediction optimization, pre-fetch optimization, and register usage optimization related to hardware platform type.

* Provides a mechanism to guarantee the initialization order of the global objects. C++ language can only guarantee that global objects within the same compiling unit are initialized in the defined order. There is no guarantee about the order in which global objects from different compiling units are initialized. For compilers (e.g. GCC) that do not support customized order for global objects initialization, libutilitis also offers a compiler-independent mechanism to guarantee the order of global objects initialization. For further discussions on this topic, refer to section "Threads Safety and Interdependence Issues with Global Objects Initialization" in my document C++ Coding Guidelines (Chinese only).

* Implemented a collection of platform-independent call stack back tracing tools, which can be used to obtain call stack information under the current context or under specified context. These information include module name, source code file name, line number, function/method name (supports MSVC/GCC name mangling) and etc.

In addition to the encapsulation of low-level details, a number of fundamental tools are also implemented within the interface layer. For example,

* Generic handle template (a smart pointer class with reference counting support). In most cases, the generic handle is designed to replace the traditional C pointer. Its major characteristics are as follows:

  - Automatic management: users do not need to worry about when the resources should be destroyed and who should destroy them.
  - Exception safety guarantees: satisfies RAII (Resource Acquisition Is Initialization) semantics, and ensures that exceptions will not result in any memory leak or program error.
  - High efficiency: the generic handle has the same efficiency with a pointer for performing all operations, except creation, destroy, and copy operations. Even while performing the latter operations, only maintenance of reference counting is added.
  - Error prevention: the generic handle can effectively avoid memory leak and other program errors, and has dramatically simplified program design associated with pointers.

- Customized destroy strategy and NIL value: programmers can customize the destroy strategy (the default is to use delete operation) and the NIL value (set to NULL by default). For example, for templates that deal with file handles, programmers can set the destroy strategy as calling the file close function, and specify the NIL value as something like INVALID_HANDLE. Customized NIL value and destroy strategy are introduced in as template parameters and are bound with instances while compiling, so they will not add any processor and memory consumption at runtime.

- Support for static handle (binding without ownership)

- Support for construct with DontInit indicator, which helps to create efficient local static object with multi-thread safety. For more details about this topic, refer to section "Thread Safety Issue with Local Static Objects Initialization" in my document C++ Coding Guidelines (Chinese only).

- Users can specify reference counting variable type by the template parameter. This helps to ensure multi-thread safety when using atomic variable type (the default value) to completes reference counting. When thread safety is not required, users can choose to implement a reference counting mechanism using primitive Integer type that has better performance.

* Temporary handle template: similar with the generic handle template, temporary handle template also obeys RAII semantics, customized destroy strategy and NIL value, and other characteristics. The only difference is that the temporary handle does not support reference counting, so users need to explicitly release ownership in order to pass pointers. Different with the generic handle which is often used to pass objects between functions or threads, the temporary handle usually guarantees RAII semantics and security (when an exception occurs) for a single function or code block. Because reference counting is not needed, the temporary handle has exactly the same efficiency as primitive pointers with respect to all operations.

* basic_buffer: the basic_buffer template is an efficient buffer management tool compatible with the basic_string template within the C++ standard library. It is fully compatible with STL basic_string, but offers higher space and time efficiency and a more fine-grained storage management mechanism. Thanks to the support for a collection of technologies like reference counting, copy-on-write, memory reallocation, buffer pre-allocation and static (no ownership) buffer, basic_buffer can offer much higher efficiency than basic_string. Furthermore, there is a specialised template class which is specifically optimized for BLOB (basic_buffer<BYTE>) objects.

* String extension class: provides extension capabilities for basic_buffer or basic_string, such as streaming operations, type conversion, common string parsing tasks, various inverse operations, BRE/ERE/ARE (TCL 8.2) regular expressions with Unicode charset support, and escape operations on the basis of callback or symbol table, and etc.

* High-efficiency linked list node template: the CListNode template has encapsulated the node operations associated with doubly linked list. Compared to std::list, CListNode provides O(1) time complexity and more flexible linked list usage like node separation, node exchange and node moving without the need for memory reallocation. When there is a need to use linked list, users should first try to use the std::list container. Only when the std::list container cannot satisfy the requirements, users can consider using CListNode to implement dedicated linked list.

* LRU Cache template: the template is a buffer manager powered by the LRU (Least Recently Used) algorithm. It supports a complete collection of operations like settings, delete, match, traversal and management. Users can choose to perform key-value indexing and matching using hash table (hash_map/unordered_map), B tree (std::map) or any STL compliant containers. This buffer manager utilizes CListNode for maintaining an efficient LRU list.

* Other extensions of standard library, including: the fixed_vector template which utilizes static buffer area and is compatible with std::vector, a circular buffer container that is compatible with std::deque, wrapper class of standard C library's file operations, universal pointer and subscript based iterator encapsulation, various member function adapters, and etc.

* Encapsulation of exceptions processing: this encapsulation obeys the RAII semantics and is used to handle unexpected exceptions as well as exceptions occurred with operator new and operator delete. For more details about this topic, refer to sections "Exceptions" and "C++ Exceptions Mechanism Implementation and Consumptions Analysis" in my document C++ Coding Guidelines.

* Error handling mechanism: libutilitis can capture all unprocessed fatal errors within applications, and output them to the global logger object. These errors include:

  - C++ runtime errors, such as unexpected exceptions or exceptions that are within an exception;
  - Errors reported by the operating system, such as memory access violation.

  Meanwhile, the current function call stack under the problematic context will also be output to the global logger object together with the errors.

In conclusion, the base module has encapsulated all the fundamental features associated with the low-level platform and the compiler environment. The libutilitis library and all other modules within the application platform highly rely on the fundamental tools defined by the base module.

## 3.2 System Utilities Module—sysutil

The system utilities module is built on the basis of the base module, and has encapsulated all functionalities associated with the operating system. It offers a platform-independent, easy-to-use and reliable interface for users to interact with system functions. The factors being considered before designing this module and their importance levels are the same as those for the base module.

The design goal of the system utilities module was to encapsulate the great majority of common services provided by the operating system and hardware platform. This module provides corresponding interfaces for almost all the features and functionalities that can be found in traditional operating system textbooks. For example:

- Process control, including:
  - Create process (e.g., sub-process creation with user impressing, input/output re-directing, hidden process creation);
  - Terminate process and wait for process being terminated;
  - Preemptive settings as well as settings like priority level, scheduling algorithm, and CPU affinity;
  - Set limits for resources like memory and file handle;
  - Query process information, include: resource usage such as CPU time and memory size; modules loaded by the processes; processes loaded by the system; and memory mapping information, etc.
  - Look up belonged module using a given address. For example, look up the dll/so module that provides function calling according to a function pointer).
- Thread and TLS: support operations like create, run, suspend, continue, stop, kill and etc; preemptive settings as well as settings like priority level, scheduling algorithm, CPU affinity, and the ideal processor; retrieve status and statistics information of a thread; relinquish remaining time slice of current thread, or forcibly switch to another thread; create and access TLS storage.
- Coroutine: also known as fiber, co-process, and user thread, coroutine is a concurrency mechanism more lightweight than thread. The libutilitis library supports a complete set of co-routine operations, and also offers a runtime environment based on thread pool with basic FIFO scheduling algorithm supported. By deriving a new class, users can easily specify runtime environment and scheduling algorithm according to their own requirements.
- Synchronization mechanisms like semaphore, mutex, and event (condition variable). Moreover, libutilitis offers high-speed synchronization mechanisms like Futex, fast semaphore and spinlock for platforms that support hardware-level atomic operations. Futex has

implemented full user-mode mutex that supports recursive calls, thus most of the user mode and kernel mode switch of the lock/unlock operation could be eliminated. This has substantially improved its working efficiency. Compared with Windows Critical Section, Futex provides a broader range of features (such as timeout waiting) and slightly higher efficiency. Fast semaphore has similar advantages over semaphore. Both Futex and fast semaphore within libutilitis support spinlock operations, and can automatically detect the amount of online processors in the current environment and fallback to the standard mode in a single-processor environment. On platforms that do not support hardware-level atomic operations, fast semaphore performs equally to normal semaphore, and Futex is the same as normal mutex. So users can always retain the most efficient synchronization method (platform-independent) without further code changes.

- Dynamic library (dll/so) loading tool: a platform-independent tool used for loading dynamic library and locating API entry.
- Synchronous & asynchronous I/O operations on files, network, and communication devices: libutilitis has encapsulated I/O operations associated with files, network (socket, support for IPv4 and IPv6), and communication devices like serial port, parallel port and pipeline. It also offers a set of platform-independent asynchronous I/O frameworks (see the following sections).
- File mapping and shared memory: supports access control like read, write, execute and Copy-on-write (COW), and allows users to build file mapping or shared memory at specified base address.
- Directory management: contains a complete set of tools used for disk volume and directory management. It supports:
  - Traversal/copy/move/deletion of directories, files and sub-directories, manipulate properties and authority settings of them.
  - Retrieve of disk volume topology and file system information, as well as detailed information of all currently mounted volume devices.
- System clock, time zone, DST rules and time span operations: libutilitis provides a complete set of operations associated with time and calendar, and supports high precision performance counter operations.
- High precision timer: this has encapsulated a high-precision and periodic timer mechanism provided by the operating system.
- System log: send log messages into syslogd (unix) or System Event Service (Windows).
- Service manager: add/delete/manage services and drivers within the current platform or within the specified computer (currently windows only).
- Service (Daemon) framework: a platform-independent framework used for developing Windows Service or Unix Daemon.

* Charset encoding conversion: supports Windows API, POSIX libiconv, IBM libicu and ISO C locale API; can automatically select the best encoding converter according to charset encoding and the current settings of the specified platform.
* Acquisition of platform information: different with the macros that are predefined in the base module, libutilitis offers a tool used for dynamically acquiring information of the current platform during runtime. The information include operating system type, product series, version number, Service Pack/Patch number, system uptime, memory page size, CPU type/width, CPU byte order, the number of processors, and etc.
* Registry access, terminal (textual user interface) control and other commonly used features.
* Memory validation (for read/write/execute), system management (log off, shut down, restart), environment variable expanding and other miscellaneous features.

On platforms that do not support some specific features, the sysutil module also provides a transparent simulation layer for users. For example, a virtual registry implementation (fully compatible with the Windows registry) is provided on platforms that do not support registry operations. These features can be compiled across platforms and can automatically switch to the implementation that is best suitable for the current platform. For example, system offered registry service is preferred for use on Windows platform, and the virtual registry service provided by libutilitis is used for other platforms.

In addition, the system utilities module has also supplied some application frameworks that are closely related to the low-level platform. For example:

* The system services framework has encapsulated the standard workflow and working model for service programs. Applications that are built upon this framework behave as a Daemon in POSIX environment. However, in Windows, they act as a service and co-work with the System Service Manager.
* Efficient I/O framework which will be discussed in the next section.

System-level frameworks and tools have provided big help for building some critical applications. They have considerably reduced the cost for cross-platform transplanting, and also have improved development efficiency as well as code quality through high-density code reuse.

Though libutilitis should offer features as consistent as possible across platforms, but apparently there are still some differences cannot be avoided. The service manager is the most typical example. WinNT series platforms provide a service manager to manage all the background services and drivers within the current system. Similar mechanism does not exist in most POSIX environments (such as un*x/linux) and DOS environment. Obviously, it is hard to implement simulation of similar features without operating system support, because this involves interactions with the other system components.

One key principle for designing the libutilitis library is to achieve reliability, correctness and

completeness. We can choose to not include some features in libutilitis, but once we provide a feature to users, we must guarantee this component can perform correctly and stably. For some specific features, libutilitis can either not to include them, or provide a complete set of clear interfaces. For example, libutilitis will never provide a directory access class that does not support file/sub-directory traversal. This guarantees users will never be forced to bypass a component within libutilitis and implement the features again by themselves because that component is lacking of some basic functionalities.

Based on the above principle, few components within the libutilitis library may be unable to implement fully transparent cross-platform capabilities. See the User Guide for the libutilitis library for more details.

The base module and system utilities module together have encapsulated most of the services associated with the platform. In real projects, however, there are chances that users need to directly access operating system features and hardware resources. For example, when the project relies on a third-party COM component, or when hotspot codes needs to be optimized using inline assembly language.

One of the most attractive characteristic of C/C++ languages is, they have simultaneously provided easy-to-use high-level language, standard libraries with a broad function list, premium efficiency, and the capability to directly access the low-level hardware. The design goal of libutilitis is never to set obstacles in executing these tasks, on the contrast, libutilitis is dedicated to provide a set of tools that can help users to achieve their design goals in a more elegant and portable way.

The libutilitis library is intended to offer users a set of platform-independent implementations that are complete, efficient and reliable. We truly understand that lacking of any of these conditions will force users to bypass libutilitis, and turn to implement some functions (that has key importance to them or their projects) by themselves. While substantially reducing direct interactions with the low-level platform, libutilitis can also help users to complete those inevitable interactions in a more structuralized and controllable way.

For example, the library contains macros that can be used to identify:

- compiler manufacturer, version, and whether the compiler and standard library supports specific functionalities;
- the operating system on the target platform;
- CPU type/width and CPU byte order, and etc.

The library also contains other macros that can be used encapsulate different inline assembly syntax in various compilers, and the tools class that is used to dynamically acquire platform type and version information during the runtime.

The libutilitis library can perform a great majority of common tasks on behalf of users, and help users to achieve those inevitable interactions with the low-level platform in a more convenient, elegant and portable way. These have finally resulted in a more concise, robust and easier to maintain product.

### 3.2.1 High performance I/O Framework

High performance I/O framework has encapsulated a high-concurrency, high-load and multi-threaded I/O server model. In general, the current I/O models can be classified into the following major types:

* **Model 1**: multi-threaded and synchronous blocking I/O model. Use “one connection per thread/process” design. As the most basic, easiest to implement and least efficient I/O servo model, it is utilized by the famous apache web server. It has the following major problems:
  - Creating a thread/process for each connection results in high consumption. When there is high-concurrency, a great majority of server resources are mainly wasted on frequently creating and switching threads/processes.
  - Weak defence against DDoS attacks targeting high-concurrency and slow requests.
  - Lack support for applications that need to maintain many keep-alived connections concurrently (every connection will occupy a thread or process for a long time).
* **Model 2:** high-efficiency poll (epoll/kqueue/event ports…) mechanism with synchronous non-blocking I/O model. Multi-threaded and “one ready connection per thread” design. It utilizes the efficient polling interface provided by the operating system to periodically wait for some connections within a connections collection to become usable, and then performs non-blocking read and write on the usable connections. That is, read data from the receive buffer of the low-level protocol stack or copy data to the send buffer of the protocol stack. Finally, it enters waiting status again using the polling interface. The advantage of this servo model is: it can use a few threads to process a large amount of concurrent connections, achieving high space and time efficiency. Its disadvantage is, the programming model is complex and relies on specific API provided by the operating system.
* **Model 3:** this model is characterized as asynchronous I/O, multi-threaded, and the “one active connection per thread” design. In this servo model, applications submit required I/O operations to the operating system and after the operations are complete, the operating system will notify applications through a callback mechanism. Theoretically, this is the most efficient I/O servo model. The reason is that applications can submit the memory address to be transmitted to the low-level hardware, which will complete the I/O operations directly at this memory location using DMA. This has implemented zero-copy. After the operations are complete, the hardware will trigger an interrupt request to the operating system, which will then callback the application. This mechanism can avoid polling waiting and connection

collection maintenance operations in model 2. Furthermore, by submit multiple I/O requests to the underlayer driver simultaneously, there is possibility for the operating system and the low-level hardware to merge operations (merge several messages into a single network frame or disk I/O request to complete read and write) and to optimize request order (the disk head begins read/write requests from the nearest track). The major disadvantage of this model is the complex programming model. Besides, its actual performance depends on the implementation method of the operating system.

Theoretically, the asynchronous I/O architecture utilized by model 3 offers the highest I/O efficiency. In practice, its actual efficiency highly depends on how the operating system implements the AIO mechanism. For example, both Linux and Solaris do not support real kernel-level socket AIO operations. All asynchronous I/O operations on these systems are user-mode simulations using multi-threaded synchronous blocking I/O operations (i.e., model 1). We can predict that using the AIO services provided by these systems can only result in serious performance degradation.

On the other hand, high-efficiency polling interfaces like epoll, kqueue, port_get, /dev/poll and pollset have achieved the O(1) level constant time complexity in their corresponding systems. Moreover, a majority of modern operating systems have implemented (partially) zero-copy non-blocking I/O operation using reference counting and Copy-on-write (COW) mechanisms of memory page. Thus, in real production environments, we need to conduct extensive performance tests and kernel source codes analysis, in order to determine which I/O model can offer highest efficiency for the current platform.

The high performance I/O framework implemented by libutilitis offers a platform-independent I/O mechanism, and always attempts to choose the most efficient I/O servo model for the current platform. To be specific:

- Use overlapped I/O + IOCP on WinNT series platforms (NT/2k/xp/2k3/Vista/2k8/Win7 and the like).
- Use overlapped I/O + Event on WinCE series platforms (WinCE/WinMobile).
- Use POSIX AIO + Realtime Signal on posix platforms that support kernel-level AIO, such as FreeBSD/Apple Mac OS X/HP-UX/IBM AIX. However, there are exceptions to network AIO (Socket AIO), because different operating systems have different implementation for high-concurrency I/O. These exceptions are:
  - On FreeBSD, Socket AIO is implemented using kqueue.
  - On HP-UX v11, Socket AIO is implemented using /dev/poll.
  - On IBM AIX v6.1 and above, Socket AIO is implemented using pollset.
  - On all other platforms, AIO is implemented using POSIX AIO + Realtime Signal.
- Use non-blocking I/O and epoll on Linux.

* Use non-blocking IO and kqueue on NetBSD/OpenBSD/DragonFly.
* Use non-blocking I/O and Event Completion Framework on (Open)Solaris.
* Use thread pool and blocking I/O simulation in environments (such as RTEMS/eCos/DOS) that do not support any of the high performance I/O models.

## 3.3 Common Facilities Module - facility

The common facilities module is built on the basis of the base and sysutil modules, thus it has naturally achieved platform independence. It provides the fundamental algorithms, functionalities, design patterns and frameworks, which are mainly used to simplify project building and to improve code re-usage. The following is a list of facilities offered by this module:

* Various commonly used synchronization algorithms: a RAII confirmed encapsulation of various synchronization algorithms like critical section, full synchronization locks, Reader/Writer locks and Producer/Consumer locks, and their corresponding optimized variants like fast Semaphore, Futex and spinlock optimizations.

* Time, time span and calendar tools with time zone and Daylight Saving Time (DST) rules, and corresponding time zone and DST rules interpreter.

* Command line interpreter with support for complex syntax.

* Message queue with multi-thread safety: an encapsulation of a high-efficiency message queue mechanism used for inter-thread communication. This can be implemented using std::deque, std::list, std::priority_queue (priority queue) and circular queue which are defined in the base module, or several other containers (specified as template arguments). The queue uses Producer-Consumer algorithm, and offers a list of variants that are optimized for different usage cases (specified as template arguments, such as variants using futex and spinlock).

  In addition to the traditional message queue, libutilitis has also implemented a message queue that allows unlimited writing by producers. However, if generation speed exceeds the speed of consumption, which has caused the queue to be full, then the newly generated elements will replace the oldest unconsumed elements from the queue. This kind of message queue is mainly used in situations when there is high requirement for responsiveness but no requirement for reliable message delivery.

* Logging mechanism: libutilitis provides a graded logging method, which allows users to set the lowest level (lowest urgency) allowed for recording events for logger objects. Each logger object can be bound to several loggers simultaneously. A logger represents a class of data targets used for storing logs, such as windows, files, system logging service, and etc. When the user writes

logs into a logger object, these logs will be distributed to all the loggers that are bound with this logger object. The libutilitis library has implemented various types of loggers including files, terminal windows, standard output device, periodic files, memory buffer, network connection, system logging service (Windows Event Log service and UNIX syslogd), and standard log servers talking syslog protocol (RFC 3164). Users can also easily implement their own loggers through simple derivation.

Logger objects also support a tool called filter, which provides a callback mechanism, monitors and filters logs for the current object, and decides whether a log message is allowed to be recorded. The libutilitis library has provided log filters based on wildcard and regular expression. Users can also easily implement their own filter mechanism.

Logger objects support logging messages in a non-blocking manner, to improve concurrency and to mitigate the delay and performance degradation caused by log message surge. In non-blocking model, applications submit log messages to a message queue, and the logger object will complete all filtering and recording tasks within a separate thread. Users can continue working with no need to wait for the logs to be written into the recording device (e.g., disk, network, and screen).

* Modem control (AT commands on serial communication): supports a full range of AT commands, time-out operations, and dial-up and back-to-back connections (used for long-distance and narrowband transmission at a low cost).

* Reliable Session Protocol (RSP, based on TCP and serial communication) defined by Bell Labs is widely used in Avaya switches and other high reliability areas. RSP protocol maintains its own receive & send windows, and has implemented re-send upon timeout, heartbeat detection, and flow control algorithm based on message window and real-time RTT auto adapting.

* Efficient and message-based session-layer protocol: this is implemented via two methods. The AIO version is implemented on the basis of the High Performance I/O Framework provided by libutilitis, and is ideal for high-concurrency and high-load environments like large-scale servers. The synchronous I/O version is easy to use, and is applicable to client and low-load servers.

* HTTP and FTP clients: supports passive FTP mode, HTTP Keep-Alive Connection, SSL/TLS, and HTTP/FTP/SOCKS agents.

* A Web framework that is based on the efficient I/O framework as well as HTTP/FastCGI/SCGI protocols (see the following sections for more details).

* Keyword tree and keyword tree with matching rules: keyword tree is a common container that is usually used for hierarchical prefix match for certain type of key value information, such as automatic completion and area codes matching. Keyword tree with matching rules has the

same behaviours with normal keyword tree. The exception is that the token added into the tree can be divided into two halves by the specified delimiter. The first half uses standard keyword tree match. After the first half is matched, several times of matching by user-defined rule will be performed on the second half (e.g., rules of regular expression). The item can be deemed to be a real match only when the rules are satisfied.

* Timer: different with the high precision timer that is within the sysutil module and is based on operating system related services, the timer here is implemented using timing threads that are maintained by libutilitis itself. The reason for implementing this timer is that the high precision timer provided by the operating system usually causes high consumption of resources and the total number of triggers is limited. For example, each process on Windows platform can create a maximum of 16 high precision triggers simultaneously. Moreover, creating high precision trigger will also change the hardware clock interrupt frequency, thus decrease the overall system performance.

  The timer provided by libutilitis can support a lot of timer tasks triggered periodically. It also supports timer groups: namely, all timer-triggered tasks are divided into several groups by type, and each group of tasks run on a dedicated timing thread without interference other groups. The advantages for doing this include:

  - Running different timers on different threads can eliminate mutual blocking between timers.

  - Users are allowed to specify different timer resolution and priority level for different threads.

  - Users can enable the timer calibration feature for timer groups that require high precision (millisecond level). When this feature is enabled, the triggering interval will be calculated based on factors like actual blocking interval and actual consumptions of executing the periodicity tasks.

* Task manager (based on the timer): the task scheduling component consists of two parts: the task manager and planned tasks. They provided very similar functionalities with the timer. In fact, the task manager itself is implemented on the basis of the timer. Each application can contain any number of timer threads (but most applications need only one timer thread). In each timer thread, there can be any number of timers. As a special type of timer, each task manager can contain any number of planned tasks to be executed.

  Offering the above three levels of timer mechanism is not something done on a whim, but is the result of observation on real use cases. The reason for offering several timer threads is already described in details in the descriptions about the timer. Here we will discuss the difference between using task scheduler and using the timer directly.

- Timer can be triggered only at specified intervals. However, the triggering condition for scheduled tasks can be quite complex.

- It is hard to achieve co-working among several timers. However, scheduled tasks can be grouped by type and the groups can easily co-work with each other (usually, each group of tasks are managed by a dedicated task manager).

- Scheduled tasks can be executed by the order of priority, but timers do not support this.

- Scheduled tasks are usually created in heap and are maintained using smart handle. They support “fire and forget” semantic. Users just need to create tasks, and do not need to worry about when they should be destroyed and who should destroy them.

* Message processing framework: this has defined a common message processing framework that supports the Chain of Responsibility and command patterns, and has implemented message pre-processing and message dispatching mechanisms.

* Prototype factory: this has defined an efficient prototype factory framework that is implemented using balanced binary trees or hash tables.

* Persistence framework: this defines an object persistence (serialization) framework, and has implemented two storage formats for collection serialization, one supports random access, and the other is mainly used for long-term archiving. Persistent data can be written into any data sink and read from any data source.

* Virtual registry (CConfig): it provides Windows registry simulation service. The main characteristic of virtual registry is, it is implemented on the basis of ISXF format, which is a platform-independent binary format. This guarantees:

  - Platform-independence capability and superior read & write performance of data accessing.

  - I18N capability: strings within the virtual registry are all saved in Unicode (UTF-8). This guarantees that issues like garbled texts displayed/saved and incorrect data will never occur in any language environment.

  - High efficiency: after the virtual registry is loaded, all subkeys and values are stored in the balanced binary tree. This can ensure high efficiency even when searching in or accessing a large data set.

  - The CConfig component also provides the ability to calculate the difference between any two objects. The difference data (including records such as addition, modification, and

deletion of values and subkeys) are stored in a compact and efficient binary format. The difference data can be applied to the specified CConfig object to implement version control functions such as version rollback and multi-version management. The CConfig component also supports the conversion of binary differential data into human-readable summary text. This makes it easy for applications to implement administrative functions like revision review and change auditing.

- Support for importing/exporting data from/to CSV, INI, JSON, and XML files, and import and export operations between the virtual registry and Windows registry.

  CConfig Schema has been widely used for communication between different systems (subsystems), because it has a bunch of advantages: high efficiency, platform-independence, I18N, self-explanation, extensibility, flexibility, and the support for CSV, INI, JSON, XML and other common data formats. In addition to the CConfig configurations editor tool, we also provide CConfig development kit for commonly used languages like JavaScript, C/C++, Java, .NET (C#, VB.NET, J#, and etc.) and PHP, for the purpose of reducing cost for partners and third-party developers.

  For more details about CConfig, see sections 6.2.4 Universal Graphic Controls, 6.3 CConfig Language Binding Component, and 6.4 JavaScript Tools Library - libbaiy .

* String-matching rules: each string-matching rules table can contain any number of string-matching rules. Users can specify the string to be matched based on the rules collection. The matching rules currently supported by libutilitis include: range, wildcard, regular expression, enumeration, and etc.

* Thread pool: to create and management thread pool objects that can be adjusted dynamically.

* Common data processing framework: libutilitis has defined a high performance data processing framework that supports zero-copy and provides a broad range of data sources and data sinks, such as source and sink that are based on files, network, serial port, object queue, and memory buffer. The framework has also implemented various filters such as container filter and T filter. The libaudioio library, which we will discuss in the following sections, is implemented using this framework.

* Virtual File System (VFS): it is an abstraction framework. Anything like a folder that contains files can be encapsulated as a virtual volume. There are two types of VFS virtual volume: encapsulated virtual volume and file-based virtual volume.

  The encapsulated virtual volume implemented by libutilitis contains standard disk directory, FTP-based VFS and HTTP-based virtual files. The libutilitis library has also defined a basic file-based virtual volume, which can pack a folder containing any number of files and

subdirectories into a single file to be accessed as a VFS. It is also allowed to add metadata of any complexity for each file and directory (by attaching a virtual registry on it). Furthermore, libutilitis provides a virtual volume implementation that is fully based on memory, which is specifically useful for creating temporary data and implementing memory cache.

A virtual volume can contain other virtual volumes, and allows nested access with zero performance loss. Additionally, libcrypto has defined a file-based virtual volume that supports compression and strong encryption on-the-fly (see following sections for more details).

- Runtime environment manager: it provides the following common runtime functionalities:
  - Environment variables management: maintains an internal environment variables system own by the application itself, and provides traversal and string expanding services. The variables manager supports recursive resolution of environment variables and referencing of system environment variables.
  - File deletion reservation: users can reserve a deletion before creating a temporary file. This can ensure this file is deleted the next time the relevant application is launched (at the latest), even when the system has got a serious problem such as power down. The file deletion reservation service guarantees transaction-based completeness.
  - Temporary file creation: to create and open a temporary file atomically.
- IP network list based on IP address and mask: IPv4 and IPv6. Supported operations include matching, traversal, add, delete, serialization, and etc. This can be used for implementing whitelist and blacklist.
- Inter-System eXchangeable Format (ISXF) read/write operations: Inter-system exchangeable format is a platform-independent, self-explanatory binary data encoding. It is mainly used for data serialization and information exchange over network. The ISXF encoding is inspired from data exchange schemes like SUN XDR (NDR/RPC) and ISO/IEC/ITU ASN.1 2002 DER/BER. ISXF is specifically optimized for Intel and recent ARM, RISC-V and MIPS processor architectures (LE byte order). ISXF format message can be read from any data source and be written to any data sink.
- Variant data type (CVarType): it is mainly used for situations that require use of dynamic types, such as highly abstracted data interfaces. Like the traditional VARIANT and _variant_t, CVarType has implemented dynamic type operations that are similar to those in JavaScript. The difference between CVarType and other implementations (like _variant_t) is, CVarType eliminates the need for resolving issues like inter-process or inter-language message passing, thus it has implemented efficient zero-copy message passing using reference counting and Copy-on-write technologies. This can dramatically improve space and time efficiency.
- Platform-independent language resource pack and a multi-language control framework based

on Observer (publish/subscribe) pattern. The matching of any language resource is O(1) complexity, and each item in the language pack can contain any number of titles, tips, help information, and additional data resource. The language pack can automatically complete tasks like encoding conversion and font matching according to the current runtime environment.

* Efficient CSV/JSON generator and parser: uses iterative algorithms and a manually optimized lexer, which allow big files to be generated and parsed efficiently at low overhead.

* Stack operations based on files and data blocks.

* Periodic file operation: provides a file handling class that can automatically perform periodic file creation (daily, weekly, monthly, and etc.). It also allows users to specify the maximum number of files that can be retained. This component is often used as the log facility for long-term running services.

* Encapsulation of file-like read/write operations from/to dynamic and static memory buffers.

* Generic data query object: this has defined a generic query statement, namely, (restrictions) AND (query condition) + Sort Criteria + Limit/Offset limitation + advanced options. The “restrictions” and “query condition” can be composed of any number of sub expressions, each of which can contain operations like equal to, not equal to, belong to, more than, more than or equal to, less than, less than or equal to, wildcard matching, and regular expression matching. The data query object is responsible for completing validity check on the operations.

  Additionally, several sub expressions can be concatenated using conjunctions like AND, OR and predicates like NOT. AND operations have higher priority over OR operations, but parenthesized expressions are supported for the purpose of re-defining priority for operations. In the Sort Criteria, users can specify any number of fields to sort them in ascending or descending order.

  The purpose for adding a separate “restriction” is to help users to implement functionalities like data access control.

* Data query engine: the query engine will perform syntax analysis, semantics analysis, and Intermediate Code generation and optimization, and will execute query on the final abstract tree. It uses “parenthesized expression > AND > OR” priority order to perform parsing and evaluation on the expressions, and also supports short-circuit expressions (short-circuit evaluation). To guarantee generality and flexibility, the evaluation of sub expressions can be completed by user-supplied visitor.

  In addition to making applications database-independent, the query engine also provides a variety of advanced characteristics that are not supported by SQL language, such as ARE (Advanced Regular Expressions) query with support for Unicode charset, join query with

support for nested tables, mixed query of business data and configuration data, virtual field query, and other customized queries.

Similar with the other components within the platform, the query engine was implemented using C/C++, and its hotspot codes were optimized using assembly language for mainstream hardware platforms. Its high efficiency and reliability have been verified in the real production environments of many Fortune 500 companies. Even when the optimization methods of short-circuit expressions is disabled, 13 millions times of evaluation of logic expressions (A and B or C and D) per second can be achieved on a ThinkPad W510 notebook (having 4 cores and 8 threads) produced in 2010, using single core and single thread only (Intel Core i7 1.6GHz).

* Search Helper: converts the specified string into a format that can be easily searched, including removing all punctuations, converting characters to lowercase, and creating abbreviations. For example, the string "_Steven.Jobs_" will be converted to 'steven jobs' and 'sj'.

  In addition, the Search Helper can convert hieroglyphics to its Latin expressions, and can support various languages. For example, it can convert specified Chinese characters into different Latin expressions like Hanyu Pinyin (PRC), Taiwan Pinyin, Japanese Romanization, and Korean Romanization, so that they can be easily searched. For example, "Calvin・赵" will be either converted to 'calvin zhao' and 'cz' using Hanyu Pinyin (PRC), and or converted to 'calvin jhao' and 'cj' using Taiwan Pinyin. Similarly, "13 叔" will be convert to '13 shu' and '13s'.

  For characters that have several pronunciations, the Search Helper will output all possible combinations. For example, the string "单田芳" will have results 'chan tian fang', 'dan tian fang', 'shan tian fang', ‘dtf’, ‘ctf’, and ‘stf’.

The design goal of the common facilities module was to achieve the followings by improving component-level reuse:

- Help users to further simplify and complete regular tasks.
- Reduce code bugs.
- Reduce the difficulty of code writing and code maintenance.
- Improve the average expression ability of each line of the code.

All advanced characteristics of C++ other than the templates have been avoided intentionally in the base and sysutil modules, though they can be used appropriately in the facility module. The purpose is to eliminate additional consumption caused by the followings: virtual function “one pointer member per object” and “one base-offset reference per call”; type_info static linked lists traversal and comparison associated with RTTI; virtual base “one pointer member per object” and virtual base members indirect addressing. For detailed analysis of the advanced characteristics of C++, refer to section “Consumption analysis and usage guidelines for RTTI, virtual functions and virtual base class”

(Chinese only) in my document C++ Coding Guidelines (http://baiy.cn).

## 3.3.1 Web Framework

The libutilitis library supports multiple Web protocols, including HTTP (RFC2616), FastCGI (www.fastcgi.com), and SCGI (www.python.ca/scgi/). The following table compares the features and capabilities among these protocols.

| Protocol | High Perf. I/O Framework | Synchronous I/O + thread pool | Support for Keep-Alive |
|---|---|---|---|
| FastCGI | Yes | Yes | Yes |
| SCGI | Yes | Yes | No |
| HTTP | Yes | Yes | Yes |

As shown in the above table, libutilitis provides all the three protocols with two implementation methods, namely High Performance I/O Framework and "synchronous I/O + thread pool".

### Synchronous I/O + Thread Pool Architecture

The "synchronous I/O + thread pool" server model is mainly used to easily implement low-load Web applications. Figure 7 shows how this server model works.

Figure 7

As shown in Figure 7, the Web application framework, which is based on “synchronous I/O + thread pool”, comprises three main parts, namely listener threads pool, Web requests queue and worker threads pool.

Listener threads pool waits for a Web request that is sent from the reverse proxy or browser, and completes initialization tasks associated with this request. Then it puts the request to the end of the Web requests queue, and continues to listen for the next request. All these tasks are completed by a separate thread pool to improve concurrency under high-load situation. The maximum number of threads in the listener threads pool is customizable by the user, and can be dynamically adjusted

according to load pressure.

The Web requests queue arranges all pending requests in a queue. In segmented processing mode (see below), the Web requests queue also lines up intermediate requests that are not fully processed. The maximum size of Web requests queue is customizable by the user.

The worker threads pool continues to pick up pending requests from the front of the queue, read and process messages that are sent from the client, generate a response, and return results. The pool can be adjusted dynamically according to current load status and user configured parameters. When the pool is in low-load or idle status, the number of worker threads will be reduced to the minimum value specified by the user. When high-load status persists, the number of worker threads will increase gradually until the maximum value specified by the user to provide extra concurrency capability. When load pressure continues to decrease, the framework will gradually reclaim idle worker threads until the minimum value specified by the user. Worker threads reclaiming strategy is configurable by the user.

Depending on applications, the "Synchronous I/O and thread pool" based Web application framework supports two servo models:

* **Blocking I/O model**: in this model, once a Web request arrives, the worker thread will complete the whole process including read, analysis, calculation, results generation and response returning. The worker thread will not process the next request until the current request is fully processed. The Blocking I/O model is the simplest servo model and also the easiest one to implement, but it has the disadvantage of low concurrency in high-load complex applications.

* **Segmented handling model**: in this model, each request will be divided into several segments. Once a segment is processed, the worker threads pool will put that request and its related working status at the end of the queue, and then it will pick up the next request from the head of the queue. Compared with Blocking I/O model, this model can provide better balanced assignment of server resource and network bandwidth, but it needs to save users' intermediate state.

## High Performance Asynchronous Web Framework

Using the High Performace I/O Framework provided by the sysutil module, libutilitis has implemented an **efficient asynchronous Web framework** which is based on asynchronous I/O and callback. This framework can easily support tens of thousands of concurrent connections even on an outdated AMD AthlonXP 2600+ (single-core/single-thread @1.8GHz) machine manufactured in 2002. On an entry-level 1U PC Server (with dual-socket Intel Xeon 56xx) manufactured in 2011, a single node can permit tens of millions of concurrent connections. Compared with IIS+asp.net / Apache+php / Nginx+php and Java / Python / RoR schemes, the Web framework based on C/C++ has a number of

advantages over them in terms of performance.

Even if we put aside .NET / Java / PHP / RoR / Python which are the relatively less efficient Application Logic part, the Web framework still is directly comparable to the leading Web servers like Nginx, Lighttpd, Charokee, and IIS. Web servers like Apache that uses the low-efficiency “one connection per thread” model are completely surpassed. The best method to evaluate performance is to do real tests. As a reference, the following table compares the Requests Per Second (RPS) among the Web framework, IIS and Nginx running on different platforms:

| Web Servers | Platforms | Requests Per Second (RPS) |
|---|---|---|
| IIS | Windows 2k3 / ThinkPad T61 Core 2 Duo Mobile @2.0GHz | 33500 |
| libutilitis | Windows 2k3 / ThinkPad T61 Core 2 Duo Mobile @2.0GHz | 33200 |
| Apache | Windows 2k3 / ThinkPad T61 Core 2 Duo Mobile @2.0GHz | 5650 |
| IIS/ASP.NET | Windows 2k3 / ThinkPad T61 Core 2 Duo Mobile @2.0GHz | 5270 |
| Nginx | Ubuntu 8.04LTS / VMWare Single Core Guest @ ThinkPad T61 | 17000 |
| libutilitis | Ubuntu 8.04LTS / VMWare Single Core Guest @ ThinkPad T61 | 18900 |
| Nginx/PHP | Ubuntu 8.04LTS / VMWare Single Core Guest @ ThinkPad T61 | 160 |

Table 1. RPS comparison between Web frameworks

The test of Requests per Second is mainly focused on inspecting the expense the low-level application framework spend on each request. So we return only a simple “Hello World” page for the purpose of minimizing the interference caused by content generation and transmission. All the tests on Windows platform are completed on a single IBM ThinkPad T61 notebook manufactured in 2007 (Windows 2003 SP2, Intel Core 2 Duo Mobile Dual-Core Processor, 4GB DDR2 800 Dual-Channel Memory). And all the tests on the Linux platform are completed in a VMWare virtual machine under above mentioned host (the guest environment has single CPU and 768MB memory, with Ubuntu 8.04LTS installed). Configurations for Nginx have been optimized to the greatest degree possible. Otherwise, the test result is only 12400 RPS under the default configuration with gzip compression disabled.

All the results shown in the table above are the average result for continuous 10 tests. As shown in the table, libutilitis Web framework makes its rivals such as ASP.NET / PHP lag far behind, and is equally matched with the leading Web servers such as IIS and Nginx. The reason is, libutilitis uses the High Performance I/O Model supported by underlayer operating system and hardware. For example, libutilitis uses overlapped I/O+IOCP architecture on Windows platform (the same as IIS), and uses non-blocking I/O + epoll architecture on Linux platform (the same as Nginx).

Additionally, all the tests were executed under the pressure of 100 concurrent connections and 100,000 continuous requests. 100 concurrent connections will not expose the drawback (efficiency will decrease drastically as the number of concurrent connections increases) of architectures based on Apache/PHP/ASP.NET and the like, and also can maximize the efficiency of the resources such as CPU,

network adapter and memory.

Admittedly, the above tests can only reflect a single aspect of Web applications. In terms of a Web framework running on the specified platform, its performance is usually observed from the following three aspects:

* **Maximum Concurrent Connections:** the maximum number of concurrent HTTP connections that the Web framework can support on the given platform.

* **Maximum Request per Second:** the maximum number of requests that the Web framework can support per second on the given platform.

* **Dynamic Content Generation Performance:** measurement of algorithm performance for generating contents like graphics, reports, and pages in real time on the given platform.

Important: before starting with a test, make the test environment as clean as possible so the test result will not be affected by the other factors. For example, the Request per Second test that we mentioned earlier was executed under the condition with a tiny “Hello World” page and a moderate number of concurrent connections, in order to eliminate, to the greatest degree possible, the interference caused by other factors.

It is important to keep the testing environment as clean as possible, which is the basis of all modern science. For example, the preset conditions (zero resistance, absolute horizontal, 1 standard atmospheric pressure and zero centigrade) in physics, chemistry and other disciplines; and the prerequisites for testing the acceleration (0-100km) performance of a car: wind velocity (statical stability), road conditions (flat, no rain and snow), slope (horizontal). A clean testing environment is good for discovering rules and characteristics of things, and also facilitates performance comparison between different products.

Maximum Concurrent Connections is a hard indicator for the Web framework. It is closely related to reliability, robustness, and survivability under environments like high-load, DDoS attack, and slow connection attack. On the above mentioned T61 platform, libutilitis Web framework can support high-load situation with over 200,000 concurrent connections, which is completely unachievable for the other Web frameworks like PHP/Java/ASP.NET. With regard to the topic about Request per Second, the previous sections have provided detailed discussion and comparison.

Dynamic Content Generation Performance is all about performance comparison between programming languages and databases. There is no need to talk more about the performance advantages of C/C++ over other popular languages used for Web applications development, such as PHP, Java, C#, Ruby, Perl, and Python. Also, a lot of trustable benchmark comparisons can be found on the Internet. The performance comparison among database and memory cache products is out of the scope of this paper, because it has little relevance with how to choose a Web framework.

We fully understand that performance related topics are always controversial and it is hard to make a choice purely based on theory. Practice is the sole criterion for testing truth. This is especially true in relation to performance measurement. Thus we welcome any invitation for A/B testing and performance evaluation.

Not long ago, as the continuous performance improvement for hardware, efficiency of programs has become a topic that need to be considered only for operating system and a few software like database and large-scale applications. Nowadays, this topic is back to the spotlight due to environmental deterioration, increasing energy cost, and cloud computing and virtualization (separate a single physical server into several VPS) going mainstream. We are dedicated to help customers continuously improve product quality, application efficiency as well as Performance per Watt, for the purpose of reducing energy consumption and carbon emission and better adapting customers to cloud and virtualization environments.

## Keep-Alive and HTTP Pipelining Mode

For high-concurrency applications, the Keep-Alive mode can spare the operations that are repeated between requests, such as TCP connection establishment (3-way handshake), connection termination (4-way handshake), flow control and initialization. At the same time, it has dramatically saved system resource (TIME-WAIT pool). Thus it also plays a critical role in high-performance network applications. FastCGI enables the Keep-Alive mode using the FCGI_KEEP_CONN flag contained in the Begin Request message. HTTP turns the Keep-Alive mode on and off using the standard “Connection:” header. The libutilitis library provides Keep-Alive support for both HTTP (1.1 or 1.0) and FastCGI.

Under environments with Keep-Alive enabled, libutilitis can also support HTTP Pipelining that conforms to HTTP 1.1. In the HTTP Pipelining mode, clients can continuously send multiple requests without waiting for the responses returned from the server.

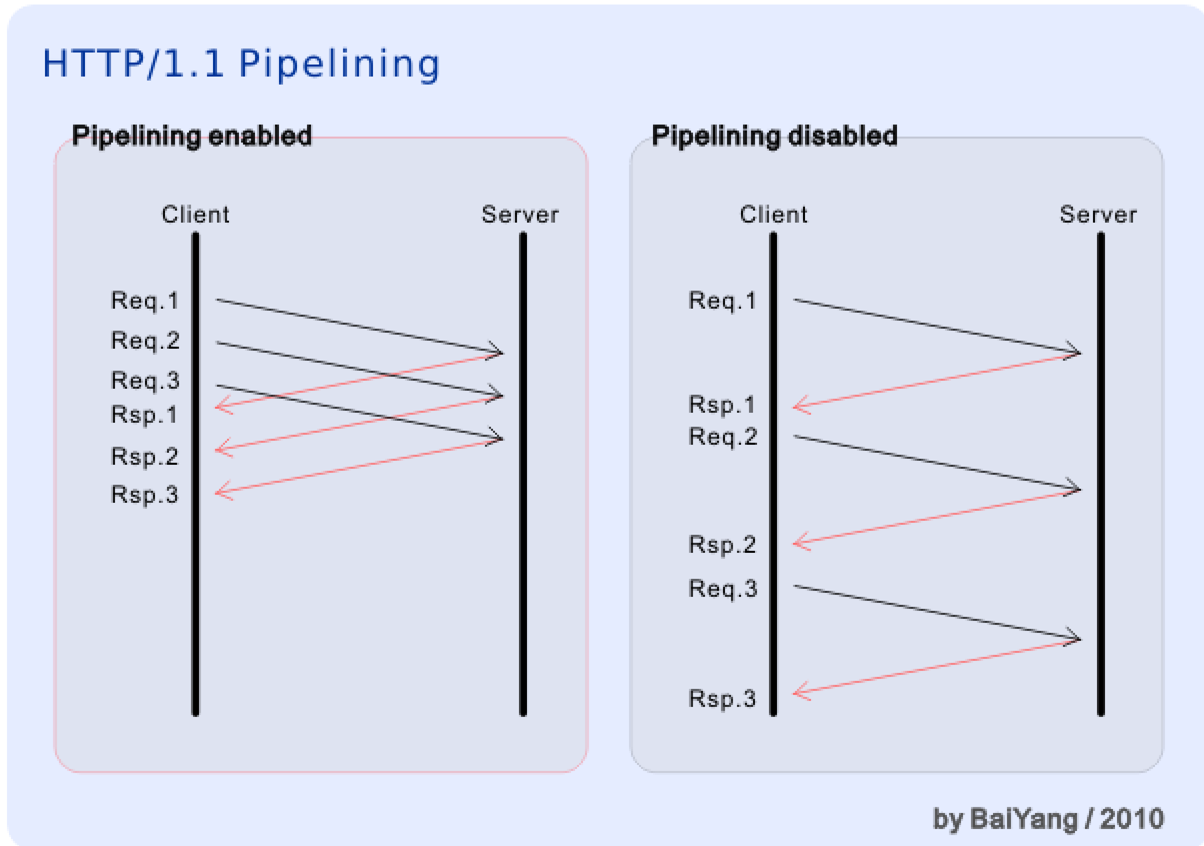



Figure 8

As shown in Figure 8, HTTP Pipelining technique has avoided the “stop-wait” protocol by continuously sending multiple requests. This has dramatically reduced delay in communication and processing and has enhanced network utilization and throughput. Thus the overall user experience has been considerably improved.

## 3.3.2 Typical Web Use Cases

The following figure shows a typical example of high-load Web applications.

Figure 9

Figure 9 shows a typical high-performance Web application with three-layer architecture. This is a proven architecture that has been widely deployed in many large-scale Web applications including Google, Yahoo, Facebook, Twitter, and Wikipedia.

Reverse Proxy

The reverse proxy server, which is in the outside layer of the architecture, accepts connection requests from users. In real use cases, the proxy server will also need to complete at least some the tasks listed below:

**Connection management**: maintains the connection pools on the client side and application server side, manages Keep-Alive connections, and terminates them after time out.

**Attack detection and isolation**: all requests associated with business logic will be sent to and processed by the back-end application server, because the reverse proxy service does not handle any dynamic content generation tasks. Thus, the reverse proxy service will almost not be affected by program or back-end service vulnerabilities. The reliability and security of reverse proxy service only depends on the product itself. Deploying a reverse proxy server at the front-end of the application server can effectively set up a reliable isolation and attack detection mechanism between the back-end applications and remote users.

When higher security is needed, users can add additional network isolation device like hardware firewall at boundary positions of external network, reverse proxy, back-end applications and database.

**Load balance**: use Round Robin or the "Least Connections First" service policy to achieve load balance based on user requests, or utilize SSI technology to divide a user request into several parallel parts and submit them to several application servers separately.

**Distributed cache acceleration**: Deploy reverse proxy servers in groups at network boundaries that are geographically close to hot areas, and accelerate network applications by providing cache service at locations close to clients. This has established a CDN network.

**Static file server**: when a static file request is received, the server directly returns the file without submitting the request to the back-end application server.

**Dynamic response cache:** caches the dynamically generated responses that will not change for a period, to prevent the background server from frequently executing repeated query and calculation.

**Data compression**: enables GZIP/ZLIB compression algorithms for returned data in order to save bandwidth.

**Data encryption (SSL Offloading)**: enables SSL/TLS encryption for communications with clients.

**Fault detection and Fault tolerance:** tracks the health status of back-end application servers, to avoid sending requests to a faulty server.

**User authentication**: completes tasks including user login and session establishment.

**URL alias**: establishes a uniform URL alias in order to hide the real location.

**Applications mixture**: mixes different Web applications together using SSI and URL mapping technology.

**Protocol conversion**: provides protocol conversion service for back-end applications that use protocols like SCGI and FastCGI.

The popular reverse proxy services include Apache httpd+mod_proxy, IIS+ARR, Squid, Apache Traffic Server, Nginx, Cherokee, Lighttpd, HAProxy, Varnish, and etc.

## Application Service

The application service layer is located between the back-end service layer (e.g., database) and the reverse proxy layer. It receives connection requests forwarded by the reverse proxy, and downwards accesses structured storage and data query services provided by the database.

This layer has implemented all business logic associated with Web applications, and usually needs to complete a lot of calculation and dynamic data generation tasks. The nodes within the application layer may not be fully equivalent, and may be separated into different service clusters with SOA or nano-SOA architecture. Working in combination with the asynchronous Web framework provided by libutilitis, it is realistic to use C/C++ to implement Web applications that leave its rivals far behind in terms of functionality and effectiveness.

Figure 10

Figure 10 shows a typical working model with high concurrency and high performance. Each Web application node (represented by boxes labelled as “App” in Figure 9) usually works on its own server (physical server or VPS), and several nodes can work in parallel in order to easily achieve horizontal scaling (scale-out).

In the above example, a Web application node comprises three key parts: I/O callback threads pool, Web requests queue, and back-end worker threads pool. The workflow is as follows:

1. When a Web request arrives, the operating system informs AIO callback thread to process this arrived Web request, through the I/O completion (or I/O ready) callback mechanisms which are closed related to the platform such as IOCP, epoll, kqueue, event ports, real time signal

(posix aio), /dev/poll, and pollset.

2. When a worker thread in the AIO callback pool receives an arrived Web request, it attempts to pre-process the request. During pre-processing, local high-speed cache will be used to avoid data query which requires relatively higher cost. If local cache is matched, it will directly return the result (still using asynchronous method) to the client and will complete this request.

3. If the queried data is not matched in local cache, or the Web request needs writing to the database, the AIO callback thread will put this request into the specified queue. The request will wait for an idle thread in the worker threads pool to further process it.

4. Each thread in the back-end worker threads pool maintains two Keep-Alive connections: one is connected to the bottom layer database service, and the other is connected to the distributed caching (memcached) system. The worker threads pool has implemented a connection pool mechanism for both the database and distributed cache, through the method that each worker thread maintains its own Keep-Alive connections. Keep-Alive connection has substantially improved application processing efficiency and network utilization by repeated use of a single network connection for different requests.

5. Back-end worker threads wait for new requests to arrive in the Web requests queue. Once getting a new request from the queue, the thread will first attempt to match the data being queried by the request with distributed cache, if there is no match or this request needs further processing such as database writing, this Web will be directly completed through database operations.

6. After a Web request is fully processed, the worker thread will return the result as a Web response to the specified client using asynchronous I/O method.

The above procedures are intended to give you a general understanding about how a typical Web application node works. It is worth noting that different Web applications may have very different working model and architecture because of different design concept and functions.

Note that the edge-triggered AIO event notification mechanisms like Windows IOCP and POSIX AIO Realtime Signal are different with level-triggered notification mechanisms like epoll, kqueue and event ports. In order to prevent the I/O completed events queue from being too long or overflow, causing the memory buffer being locked in the nonpaged pool for a long time, the above mentioned AIO callback mechanism is composed of two separate thread pools and one AIO completed events queue. One thread pool is responsible for continuously listening for events arrived at the AIO completed events queue, and then submit the events to an internal AIO completed events queue (this queue works under user mode and will never lock memory; the queue length is user-customizable.); and simultaneously, the other thread pool is waiting on this internal AIO queue, and processes AIO

completed events that arrives at the queue. This type of design can reduce workload for the operating system, and can avoid message loss, memory leak and memory exhaustion that may occur in extreme situations. Also, it can help the operating system to better manage and utilize its nonpaged pool.

As a typical use case, most of Google Web applications like search engine and Gmail are implemented using C/C++. Thanks to the high efficiency and powerfulness of C/C++ languages, Google provides global Internet users with the best Web experience, and also has achieved completing a Web search among its millions of distributed servers around the world at total consumption of 0.0003 kW·h only. For further discussion on Google Web application architecture and hardware scaling, refer to http://en.wikipedia.org/wiki/Google and http://en.wikipedia.org/wiki/Google_search.

## Database and memcached Services

Database service offers relational or structured data storage and query service for upper layer Web applications. Depending on specific use case, Web applications can provide access to different database services using plugin mechanisms like database connector. Under this architecture, users can flexibly choose or change to a database product which is most suitable for their needs. For example, users can use embedded engine like SQLite for quick deployment and functions verification at POC stage, and can switch to MySQL database solution which is cheaper at the preliminary stage. And when business needs increase and database workload becomes heavy, users can migrate to a more expensive and complex solution such as Clustrix, MongoDB, Cassandra, MySQL Cluster and Oracle.

Memcached is a distributed data objects caching service fully based on memory and <Key, Value> pair. It offers unbelievable performance and has a large distributed architecture which eliminates the need for inter-server communication. For high-load Web applications, memcached is an important service often used to speed up database access. It is not a mandatory component, so users can wait to deploy it till the time when performance bottleneck shows up in their database service. It is worth noting that though memcached is not a mandatory component, its deployments in large-scale Web applications (e.g., YouTube, Wikipedia, Amazon.com, SourceForge, Facebook, and Twitter) has proved that memcached not only can keep performing stably under high-load environments, but also can dramatically improve the overall performance of data query. For further discussion on memcached, refer to http://en.wikipedia.org/wiki/Memcached.

However, we should note that distributed caching systems like memcached are intrinsically a compromise solution that improves the average access performance at the cost of consistency. Caching service adds distributed replicas of some records in database. For multiple distributed replicas of the same piece of data, it is impossible to guarantee the strong consistency unless we employ consensus algorithms like Paxos and Raft.

Contradictorily, memory cache itself is meant to improve performance. Thus it is unrealistic to employ the above mentioned expensive consensus algorithms. These algorithms require each access

request to simultaneously access the majority replica including master and slave nodes in the background database. Obviously, this will make performance even lower than not using caching service.

Furthermore, the consensus algorithms like Paxos and Raft can only guarantee strong consistency at single record level. That means there is no guarantee for transaction-level consistency.

Distributed caching will add complexity to the program design and will increase access delay in unfavoured circumstances such as RTT delay upon unmatched, delay upon node offline or network communication issues.

Since 20 years ago, the mainstream database products have implemented proven multi-layer (e.g., disk block, data page and query result set) caching mechanism with high match rate. Now that distributed caching mechanisms have so many drawbacks while database products have excellent built-in caching mechanisms, why the former have become an important foundation for modern high-load Web App?

The intrinsic reason is, in the technology environment ten years ago, the RDBMS (SQL) system with poor scale-out capability had become the bottleneck for network applications like Web App to expand. Thus, NoSQL database products represented by Google BigTable, Facebook Cassandra, MongoDB and SequoiaDB, and distributed caching systems represented by memcached and redis emerged in succession, all playing an important role.

Compared with “traditional” SQL database products like MySQL, ORACLE, DB2, MS SQL Sever, and PostgreSQL, both NoSQL database and distributed caching systems has sacrificed strong consistency to get higher scale-out capability.

This kind of sacrifice was a painful choice under the technology conditions at that time. Systems have become complex: traditional RDBMS is used for places where ACID transaction and strong consistency are required and data volume is small; distributed caching systems are preferred for places where there is “more read and less write” but there is still some room for compromising consistency; NoSQL is used for big data with even lower requirement for consistency; if the data volume is large and there is strict requirement for consistency, sharding of RDBMS could be a solution, which requires various middleware to be developed for implementing complex operations such as request distribution and result set merging for the underlayer databases. There are many different cases which are mingled together making the systems even more complex.

In retrospect, that is an age when old rules were broken but new rules were still not established yet. The old RDBMS is poor in scale-out capability so it cannot satisfy the emerging requirements for big data processing. However, there was not a structured data management solution that can replace the old systems and can satisfy most of user requirements.

That is an age when requirements were not satisfied. Products like BigTable, Cassandra, and memcached are self-rescue results made by Google, Facebook and LiveJournal respectively. There is no doubt these products aimed at “satisfying business requirements at the lowest cost” are poor with generality.

In 2015, finally we are moving out of the predicament. As many of NewSQL solutions (e.g., Google F1, MySQL Cluster (NDB), Clustrix, VoltDB, MemSQL, NuoDB and MyCat) are getting mature and the technology is improving, horizontal scaling capability is no longer a bottleneck for RDBMS. Nowadays architectures can guarantee enough horizontal scaling capability for the system, and simultaneously can achieve strong consistency for distributed transactions (XA).

New High-Load Web Applications Architecture

Internet /Intranet

Reverse proxy cluster

Proxy | Proxy | Proxy | ……

HTTP/FastCGI/SCGI

Application nodes cluster

App | App | App | ……

App | App | App

Object Storage cluster

Node | Node | Node | ……

NewSQL cluster

NewSQL | NewSQL | NewSQL | NewSQL | NewSQL | NewSQL | ……

by BaiYang / 2015

Figure 11

As shown in the above figure, there is no longer a keen need for distributed caching systems or NoSQL products after NewSQL is equipped with good scale-out capabilities. This has made design and development of the architecture back to simplicity and clarity. Object Storage service offers the support for storing and accessing unstructured BLOB data like audios, videos, graphics and files.

This kind of simple, clear and plain architecture makes everything seemly reverted back to years

ago. Object Storage service looks like disk file systems such as FAT, NTFS and Ext3, and NewSQL service looks like the old single-machine database such as MySQL and SQL Server. However, everything is different. Business logic, database and file storage have evolved to be high-performance and high-availability clusters that support scale-out capabilities. Performance, capacity, reliability and flexibility have grown with leaps and bounds. Human beings have always evolved in a spiralling course. Every change that looks like a return represents intrinsic development.

As the distributed file systems (e.g., GlusterFS, Ceph and Lustre) that are mountable and support Native File API are becoming more mature and complete, it is expected to replace existing object storage services for most use cases in a phased manner. This is a major milestone in the evolution of the Web App architecture, of which a real revolution will come when we can implement a high-efficiency and high-availability general Single System Image system. Once such system happens, writing a distributed application will be nothing different from writing a standalone multi-thread application nowadays. It will be nature that processes are distributed and highly available.

### Scalability of the Three-tier architecture

The three-tier Web application architecture has demonstrated incredible scalability. It can be scaled down for deployment within a single physical server or VPS, and also can be scaled up for deployment in Google's distributed application which comprises millions of physical servers around the world.

Specifically, during project verification and application deployment and at the early stage of service operation, users can deploy the three-layer service component into a single physical server or VPS. Simultaneously, by cancelling the memcached service and by using embedded database products that consume less resource and are easier to deploy, users can further reduce both the difficulty level for deployment and the overall system overhead.

As business expands and system workload keeps increasing, the single-server solution and simple scale-up will no longer be able to satisfy the operation needs. Users can achieve a scale-out solution by distributing components to run on several servers.

For example, a reverse proxy can achieve distributed load balancing by using DNS CNAME records or some layer-3/layer-4 relay mechanisms (such as LVS and HAProxy). It can also use Round Robin or the "Least Load First" strategy to make distributed load balancing for application services. Additionally, a server cluster solution based on shared virtual IP can also implement load balancing and the fault tolerance mechanism.

Similarly, both memcached and database products have their own distributed computing, load balancing and fault tolerance mechanisms. Furthermore, the performance bottleneck of database access can be resolved by changing to NoSQL/NewSQL database product or by using methods such as

master-slave replication. Query performance of the traditional SQL database can be dramatically improved by deploying memcached or similar services.

### 3.3.3 FastCGI? SCGI? HTTP!

Though libutilitis supports three types of protocols, it is recommended to use HTTP as the preferred protocol for building Web applications. The first reason is, there is no need for protocol conversion, and HTTP can perfectly support Keep-Alive so it can offer the highest efficiency. The second reason is, as the most commonly used protocol today, HTTP is supported by the widest array of products and offers the most stable implementations.

In case it is impossible to use HTTP for certain reasons (e.g., there is a need for deploying reverse proxy service on the basis of IIS 6.0 or earlier), SCGI usually comes to be the second choice. Compared with FastCGI, the biggest advantage of the SCGI protocol is simplicity. A simpler protocol makes implementation easier and is less prone to defects. As a complex and message-based network protocol which supports multiplexing and Keep-Alive, FastCGI has not been correctly implemented on many Web servers. For example, all the remote FastCGI plugins on the current version of IIS (version 7.5) and Apache (version 2.2) have defects to some degree.

Furthermore, avoiding the redundant message packaging mechanism in FastCGI makes it even easier to implement SCGI efficiently. Compared to SCGI, the sole advantage of FastCGI is the support for Keep-Alive connection. Unfortunately, among all the mainstream Web servers that support FastCGI (such as IIS, Apache, Nginx, Lighttpd, Zeus and Cherokee), only Apache plans to include support for Keep-Alive connection in the mod_proxy_fcgi module of its upcoming release 2.3. Though the other servers have provided efficient and accurate FastCGI extensions without support for Keep-Alive connection, there is no advantage over SCGI. Because the FastCGI protocol is complex and the support of it varies from different servers, SCGI should be treated as the second choice for developing Web applications.

# 4. Cross-platform Cryptographic Library - libcrypto

The cryptographic library encapsulates all the algorithms and facilities that are provided by the application platform and are associated with cryptography. Because this library is implemented based on libutilitis, it can avoid almost all platform related operations other than the algorithm optimization part.

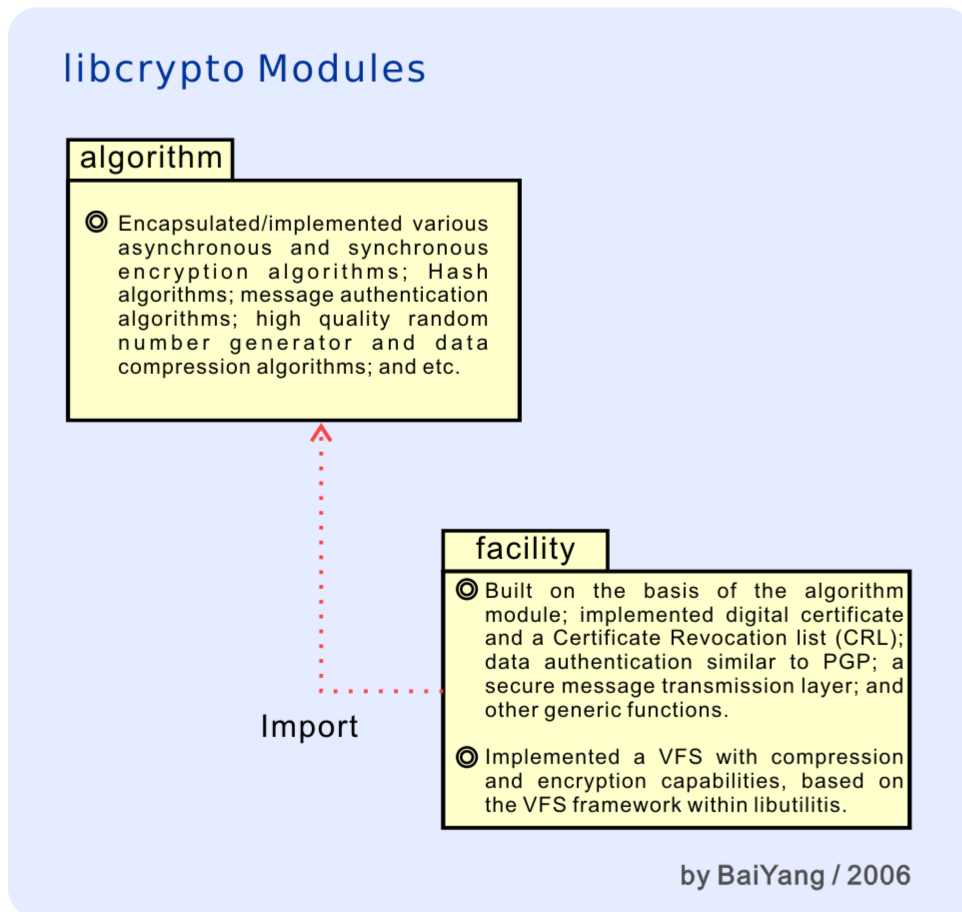



Figure 12

As shown in Figure 12, the libcrypto library is implemented using two-layer structure. The following sections provide more details about these layers.

## 4.1 The Cryptographic Algorithm Module - algorithm

This module encapsulates all fundamental algorithms. Because almost all the popular algorithms can be obtained from the Internet for free, we just need to categorize them and encapsulate them altogether. This has dramatically simplified our implementation and debug efforts.

The currently supported algorithms are described in the following sections.

### 4.1.1 Block Cipher Algorithms

| Algorithms | Key length |
|---|---|
| AES | 128bit, 192bit, 256bit |
| BlowFish | 32bit, 64bit, 96bit, 128bit, 192bit, 256bit, 384bit, 448bit |
| IDEA | 128bit |
| MARS | 128bit, 192bit, 256bit, 384bit, 448bit |
| DES | 56bit |
| DES-EDE2 | 128bit |
| DES-EDE3 | 192bit |
| CAST5 | 40bit, 64bit, 96bit, 128bit |
| CAST6 | 128bit, 160bit, 192bit, 224bit, 256bit |
| SAFER-K | 64bit, 128bit |
| SAFER-SK | 64bit, 128bit (an enhanced version of SAFER-K, has corrected a vulnerable within the key schedule) |
| TwoFish | 32bit, 64bit, 96bit, 128bit, 192bit, 256bit |
| Serpent | 32bit, 64bit, 96bit, 128bit, 192bit, 256bit |
| ARIA | 128bit, 192bit, 256bit |
| Kalyna | 128bit, 256bit, 512bit |
| Simon | 64bit, 128bit, 192bit, 256bit |
| Speck | 96bit, 128bit, 192bit, 256bit |
| SM4 | 128bit |
| ThreeFish | 256bit, 512bit, 1024bit |
| CHAM | 128bit, 256bit |
| HIGHT | 128bit |
| LEA | 128bit, 192bit, 256bit |
| SIMECK | 64bit, 128bit |

All the above algorithms support the following encryption modes:

```
Abbreviations:
  IN     - input vector
  OUT    - output vector (not for use in plaintext encryption)
  ENC    - encryption algorithm
  K      - encryption key
  P      - plaintext
  C      - ciphertext
  XOR    - exclusive or
  <<     - shift left
```

```
BSIZE – cipher block size
COUNT - counter
```

* **Counter (CTR) Mode:** IN(N) = ENC(K, COUNT++), C(N) = IN(N) XOR P(N); CTR mode is widely used in ATM network and IPSec applications. It is distinguished from the other modes by the following characteristics:
  - Hardware efficiency: allows multiple blocks of plaintext/ciphertext to be processed simultaneously.
  - Software efficiency: allows parallel computing which can make good use of parallel technologies like CPU pipeline.
  - Pre-processing: the output of the encryption box is independent of the input of plaintext or cipher text. If there are sufficient storage devices, the algorithm is just about a series of XOR operations, which will greatly increase the throughput.
  - Random access: decryption of block “i" ciphertext is independent of the ciphertext block i-1, providing high capability of random access. When encrypting a large block of data but random access to the data is required (e.g., virtual file system), random access capability can effectively increase encryption strength. This can save us from the need to reinitialize the algorithm to start a new round of encryption for each data block. In an operation mode without support for random access, the algorithm can only operate in a mode similar to electronic codebook (ECB). This is because the algorithm needs to be reset before every data block is going to be encrypted.
  - Provable security: can prove that CTR is at least as secure as other modes like CBC, CFB, and OFB.
  - Simplicity: different with other modes, CTR requires implementation of the encryption algorithm only (It does not require the decryption algorithm). This is a huge simplification for algorithms like AES.
  - No fill: can replace stream cipher.
* **Cipher Block Chaining (CBC) Mode:** IN(N) = P(N) XOR C(N-1), C(N) = ENC(K, IN(N)); this block cipher mode was widely employed before CTR appears. It was designed for grouped (iterated) encryption and authentication.
* **Cipher Feedback (CFB) Mode:** IN(N) = C(N-1) << (BSIZE-j), C(N) = ENC(K, IN(N)) <<(BSIZE-j) XOR P(N), in which j represents the number of bits for each encryption. CFB is similar to CBC, but it processes only the j bits of data at once, and discards the remaining BLOCKSIZE – j bits. From this point, CFB mode can change block cipher to stream cipher without compromising security. Nevertheless, the CFB mode is considered to be wasteful, because most of the results in each round are discarded (usually j is one byte, 8 bits. But typically the size of each cipher block could be 64, 128 or 256 bits).
* **Output Feedback (OFB) Mode:** IN(N) = OUT(N-1) << (BSIZE-j), C(N) = ENC(K, IN(N)) <<(BSIZE-j)

XOR P(N), OUT(N) = ENC(K, IN(N)) << (BSIZE-j). This mode is very close to CFB. The only difference is that output (not XORed with plaintext) from the previous iteration is used as the input of the current round. Similar to CFB mode, OFB mode can also be used as a stream cipher mode. In addition, input of each iteration not being the ciphertext from the previous iteration brings good fault tolerance. That means error propagation of a ciphertext block (one byte) will not affect subsequent ciphertext blocks. Usually this mode needs to work with message authentication and digital signature algorithms together, because not adding ciphertext to the input will result in poor anti-tamper capability. OFB mode is often used in communications with high noise level as well as in common stream cipher scenarios.

* **Electronic Codebook (ECB) Mode:** IN(N) = P(N), C(N) = ENC(K, IN(N)). It is the simplest and also the most insecure encryption mode. It always uses the same key to directly encrypt the input of each iteration. For identical plaintext blocks, it will always generate identical corresponding ciphertext blocks. This will result in poor performance with repetition statistics and structural analysis resistant. ECB mode is the worst scenario (i.e., every input plaintext block is less than BSIZE) of one-time encryption. ECB mode should be considered only for one-time pad or when a very small amount of data is being propagated.

### 4.1.2 Stream Cipher Algorithms

| Algorithms | Key length |
|---|---|
| SEAL | 160bit (the fastest symmetric algorithm until now, is 8 times faster than AES128. Though the source code is freely available, SEAL is the proprietary property of IBM in the United States.) |
| MARC4 | 32bit, 64bit, 96bit, 128bit, 192bit, 256bit, 384bit, 448bit, 512bit, 768bit, 1024bit, 1536bit, 2048bit (enhanced version of RSA RC4, has removed the insecure 256-byte header in ARC4) |
| Panama | 256bit |
| Salsa20 | 128bit, 256bit |
| XSalsa20 | 128bit, 256bit (increased the unpredictability in Salsa20) |
| Sosemanuk | 128bit, 192bit, 256bit |
| ChaCha8 | 128bit, 256bit |
| ChaCha12 | 128bit, 256bit |
| ChaCha20 | 128bit, 256bit |
| Rabbit | 128bit |
| HC-128 | 128bit, 256bit |

All of the above block cipher and stream cipher algorithms are implemented by the synchronous algorithm object within libcrypto.

### 4.1.3 Public Key Algorithms

Currently libcrypto supports RSA algorithm with a key length of 512, 768, 1024, 2048, 4096, 8192, or 16384 bits. Also, it fully supports public key encryption and signature.

### 4.1.4 Hash Algorithms

| Algorithms | Hash length |
|---|---|
| CRC32 | 32bit 4Bytes |
| CRC32-C | 32bit 4Bytes |
| ADLER32 | 32bit 4Bytes |
| MD5 | 128bit 16Bytes |
| MD2 | 128bit 16Bytes |
| SHA1 | 160bit 20Bytes |
| SHA224 | 224bit 28Bytes |
| SHA256 | 256bit 32Bytes |
| SHA384 | 384bit 48Bytes |
| SHA512 | 512bit 64Bytes |
| Panama | 256bit 32Bytes |
| Whirlpool | 512bit 64Bytes |
| TIGER | 192bit 24Bytes |
| RIPEMD128 | 128bit 16Bytes |
| RIPEMD256 | 256bit 32Bytes |
| RIPEMD160 | 160bit 20Bytes |
| RIPEMD320 | 320bit 40Bytes |
| SHA3-224 | 224bit 28Bytes |
| SHA3-256 | 256bit 32Bytes |
| SHA3-384 | 384bit 48Bytes |
| SHA3-512 | 512bit 64Bytes |
| BLAKE2-256 | 256bit 32Bytes |
| BLAKE2-512 | 512bit 64Bytes |
| SM3 | 256bit 32Bytes |
| SHAKE128 (256) | 256bit 32Bytes |
| SHAKE256 (512) | 512bit 64Bytes |
| LSH224 | 224bit 28Bytes |
| LSH256 | 256bit 32Bytes |
| LSH384 | 384bit 48Bytes |
| LSH512 | 512bit 64Bytes |

### 4.1.5 Message Authentication Algorithms

The algorithm module supports HMAC algorithms that correspond to all the above listed hash algorithms except for CRC32, ADLER32 and SHA-3.

In addition, non-HMAC family message authentication algorithms such as SipHash64, SipHash128, and Poly1305<AES> are also supported.

It should be noted that new hash algorithms such as SHA-3, Blake2, and SHAKE are not affected by length-extension attacks, so there is no need to use complex HMAC transforms.

### 4.1.6 Data Compression Algorithms

The algorithm module currently supports GZIP (RFC 1952), ZLIB (RFC 1950), BZ2, LZO, LZ4 (for real-time data compression), and etc. All these algorithms support iterated and one-time compression.

ZIP and BZIP have been well known for a long time, so I will not describe them in this document. LZO is famous for high-efficiency and lossless compression. Owing to its stable and exceptional performance, it is widely used in various areas including NASA's Mars rovers Spirit and Opportunity, the famous executable file compression utility UPX, and etc. LZO has surprising efficiency. It needs only 64 KB space for compression, and needs no extra space for decompression. On a Intel Pentium 133 with only 60MB/sec memory (memcpy) bandwidth, LZO can achieve 20MB/sec and 5MB/sec for decompression and compression respectively. With similar compression ratio, LZO can be several times faster than other famous algorithms like ZIP and RAR.

LZ4 is a newly emerged real-time compression algorithm. Compared with the famous LZO, LZ4 can provide higher efficiency: 1 - 4 times faster than LZO, under the condition of similar compression ratio.

### 4.1.7 Data Encode/Decode Algorithms

The libcrypto library currently supports the most popular data encode/decode algorithms such as HEX, BASE64 and BASE64-URL.

### 4.1.8 Random Number Generator Algorithm

As per the current environment, libcrypto can obtain a good random seed through interfaces like

CryptoAPI and /dev/random, and can work with secure hash algorithms and synchronous algorithms to generate a high quality random sequence with specified length.

## 4.2 The Common Facilities Module - facility

The common facilities module is built upon the cryptographic algorithm module. As a standard component within the application platform, it offers software designers with a universal and delicate tool set associated with cryptography. This module contains the followings:

- **Certificate**: uses public-key signature and hash algorithms to implement digital certificate and a Certificate Revocation list (CRL) that can satisfy PKI requirements.
- **Encryption algorithms similar to PGP:** uses public-key cryptography, Digital Signature Algorithm (DSA) and synchronous algorithms to implement a data encryption and signature mechanism that is similar to PGP.
- **Secure message transmission layer:** uses public-key algorithms, synchronous algorithms, message authentication algorithms, and data compression algorithms to implement a message-oriented secure and transparent transmission layer.
- **VFS supporting compression and encryption options on-the-fly**: uses hash algorithms, data compression algorithms, and synchronous algorithms to implement a file-based VFS tool, which is compatible with the VFS framework within libutilitis and supports real-time access. Iterative transformation protect of the encryption key is also supported. If the encryption option is enabled, the same encryption strength could be applied to directory list information and the meta data of each file and each directory.
- **License agreement**: uses compression, digital signature and obfuscation algorithms to provide uses with a generic tool used for license agreement authentication and protection.

# 5. Data Processing Tools

Includes data processing related components like report generator, embedded database engine, database access interface and etc.

## 5.1 Report Generation Library - libreport

The report generation library is implemented on the basis of libutilitis. It can generate reports in specified format using customized templates and specified dataset. The library has the following functions:

* Generate reports in formats such as Excel 2.0 (BIFF), Excel XP (ExcelML), Excel 2007 (xlsx), and HTML.
* Is independent of third-party components like Microsoft Excel, thus will never bring License issues.
* Supports all kinds of charts (except for Excel 2.0 and Excel XP) including line chart, bar chart, pie chart, and Gantt chart.
* Supports customizable variables and constants, formulas, as well as all field types such as date, time, numbers and text.
* Supports I18N and customizable themes including font, graphics, texts, and color.
* High performance and low consumption: load data and generate the report iteratively (one by one), one pass scan.

The libreport library offers cross-platform report generation tools with a rich set of functions. Its independence of any third-party component not only eliminates license issues, but also maintains convenience for deployment and usage, high efficiency as well as cross-platform capability. Its functionality, performance and stability have been proved after many years of usage in production environments of large enterprises.

## 5.2 ODBC Encapsulation Library - libodbc_cpp

The ODBC encapsulation library is implemented on the basis of libutilitis. As part of the ISO standard, the universal ODBC interface is widely supported by major platforms and almost all SQL/NewSQL database products. It is implemented as Native Client API in many database products like MS SQL Server, DB2, MySQL, Firebird, PostgreSQL, MySQL Cluster, Clutrix, OceanBase, InfiniDB,

MemSQL, Greenplum, and Teradata.

**Note:** Both psqlODBC and libpq are Native Client Library for PostgreSQL, and there is no dependence between them. However, psqlODBC will use libpq to complement some public operations that are not performance critical under the default compilation options.

The ODBC encapsulation library has implemented the following functions:

* ODBC interfaces are grouped and encapsulated with the concept of Connection, Statement and Result set.
* Supports prepared statement and dynamic parameter binding. Zero-copy binding is supported for all types of parameters.
* Supports zero-copy pre-binding or post-binding for result set fields.
* Thanks for the high efficiency BLOB and string type with reference counting and copy on write mechanisms which is provided by libutilitis, all zero-copy data binding operations are transparent to the user. Users do not need to pay any extra effort for it.
* Uses BLOB type and strings with reference counting and zero-copy, thus all zero-copy and binding operations are transparent to users, which spares users any additional efforts.
* Supports ODBC connection pool.
* Can easily configure commonly used parameters at connection and statement level. These parameters include time-out values, maximum result set size, maximum field size, and other common properties. It also can send and retrieve customized or advanced options with DM and Driver directly.
* Supports administrative operations like table/index existence checking, acquisition of table/database information, and termination of current operation.
* Compatible with Microsoft 32-bit/64-bit ODBC library, unixODBC, iODBC, and common ODBC or SQL/CLI applications like DB2 CLI.

ODBC not only is widely supported by major platforms and products, but also is the most effective universal database interface. Zero-copy capability of ODBC can avoid high consumptions of memory copy and format conversion, which are resulted from interfaces like OLEDB, ADO, JDBC, and ADO.NET. All the above mentioned database products use ODBC interface to implement their Native Client Library, which reflects that this interface is of superior efficiency.

In addition, ODBC is also appropriate for preventing CopyLeft restrictions with Client Library of database products like MySQL, without compromising efficiency.

## 5.3 SQLite Encapsulation Library - libsqlite_cpp

SQLite (http://sqlite.org) is a very famous embedded database engine. It has been 10 years since its first publication. There is no doubt with its stability and functionality, because it has been widely employed into the key products of famed companies like Microsoft, Google, GE (General Electric), Apple, Oracle (Sun), Nokia (Symbian), Mozilla (Firefox), Adobe, Toshiba, and McAfee. The key characteristics of SQLite are as follows:

- Free and open source under a very loose agreement.
- Supports various operating systems and hardware platforms.
- Provides stable support for large database with Terabytes of data and hundreds of millions of records.
- Offers ACID assurance; supports most of the standard SQL92 functions including primary key, foreign key, composite index, transactions, view, and trigger; supports standard SQL language.
- Single database file with cross-platform formats.
- Online backup with uninterrupted service.
- The database engine (only 300KB) is fully embedded into the application, with no need for installing any database service separately (server less). No configuration or management intervention is required (zero-configuration).
- High efficiency – the embedded engine can avoid consumptions of data transportation and encoding/decoding, and supports automatic query evaluation and optimization.

Implemented on the basis of libutilitis and libcrypto, the libsqlite_cpp library provides a C++ encapsulation of SQLite. It offers the following functions:

- Grouped and encapsulated SQLite interface with the concepts of DB (Connection), Statement and Result set.
- Prepared statement, dynamic parameter binding, and zero-copy parameter binding.
- Zero-copy binding for result set fields.
- Convenient configure options such as timeouts, WAL mode, and Shared Cache mode.
- Administrative operations such as table/index existence check, acquisition of table/database information, terminating ongoing operations.
- SQLite VFS Driver (EncVFS) provides on-the-fly strong encryption for all types of data including primary database, temporary database, attached database and all log files, to ensure no information leakage. It can employ all block cipher algorithms and stream cipher algorithms that are supported by libcrypto (Refer to 4.1.1 Block Cipher Algorithms and 4.1.2 Stream

Cipher Algorithms). Enabling EncVFS will make libsqlite_cpp depend on libcrypto.

On the premise that efficiency and functionality not being affected, the user interface and semantics for libsqlite_cpp are as consistent as possible with libodbc_cpp, so users can easily migrate between both libraries.

## 5.4 nSOA - libapidbc

The libapidbc library can be divided into three correlated parts. It defines a cross-platform plugin interfaces (IPlugin) which have the following characteristics:

* Plugins are usually provided in the form of dynamic-link library (DLL), exposing a single interface like “extern "C" void* CreateInstance(void);”. Plugins can also be embedded into projects in the form of static library or source codes, without exposing any interface.
* Plugins can carry or accept any complex CConfig information. These information are divided into several parts, such as general configurations, advanced configurations, and internal configurations. Each part can be customized according to plugin category or the specific implementations.
* Each plugin can carry two VFS which contains any type of resources. These VFS are used as an external virtual volume (usually contains resources like pages, graphics, and language pack), as well as a private virtual volume for internal use (e.g., report templates, data fields mapping table and etc.).
* Automatic plugins matching is implemented based on current environment factors such as processor, operating system, and release version (MBCS / UNICODE).

IPlugin defines a complete, self-descriptive, flexible and manageable interface. Based on it, libapidbc defines DBC (database connector) plugin types. DBC offers the following functions:

* As a middleware, DBC is easier to use than libsqlite_cpp and libodbc_cpp. It can directly use CConfig that is independent of database products to define table, index and data sharding rules, without the need for any SQL or NoSQL statement.
* Supports CAS (Compare and Swap) atomic updates that are based on the Revision field. This algorithm can resolve the competition issue that several nodes update the same record simultaneously.
* Provides data encryption service transparent to the user, and adds reliable strong encryption for data transmission and data storage for underlayer database. In situations where underlayer products or services do not support strong encryption, middleware will be employed for achieving this purpose transparently.

* Provides data compression service transparent to the user, and adds on-the-fly data compression support for data transmission and data storage for underlayer database. Users can set compression options for each table or each collection separately. Transparent data compression can be enabled simultaneously with other services like data encryption.
* Users can use the generic query object (provided by libutilitis) to describe complex query. There is no need for users to create any query statement or to consider the compatibility of underlayer DB products. Create complex query expressions independent of database products using graphic UI or the query object interface.
* Provides the following capabilities by making use of the query engine component (within libutilitis) intelligently: UNICODE and ARE regular expression matching, advanced query functions like associated query with embedded tables, virtual fields and user customizable query (e.g., users can define BELONGS_TO for department and location). Simultaneously, DBC utilizes capabilities like index provided by underlayer database to improve performance.
* Portability – libapidbc provides DBC plugins for common database products. Users can easily expand new plugins that are compliant with DBC interface regulations. Because exposed interfaces for configuration, query, update, insert and transactions are all independent of specific products, users can easily switch between database products by simply changing the DBC plugin. This has greatly reduced the dependence on a specific database product.
* User can configure any number of database server addresses for one DBC instance at the same time. The DBC plug-in can automate failover (high availability) and load balancing by these configurations.

API Nexus is another component provided by libapidbc. By registering API Dispatcher in API Nexus, a functional module can dynamically expose its services to other modules in the form of API. The dynamic API registration mechanism is a match with hot-plugging capability of the plugin.

To solve the dependence issue with plugins initialization order, API Nexus provides API calling capability based on asynchronous (message queue) semantics. When an asynchronous call issued, if the callee module (module called by the caller) is not loaded yet, API Nexus will put the request in a queue temporarily, and will process these queued requests in the order of being called once this module is loaded.

In addition, libapidbc also offers ETL mapping tools, customizable advanced query tools, locale adapter for API requests, and other auxiliary tools, as well as a set of message routing services (5.4.3 Port Switch Service) which integrates service election, service discovery, fault detection, distributed locking and other distributed coordination functions.

### 5.4.1 SOA vs. AIO

Since long ago, the high-layer architecture at server end has been categorized into two contradictory patterns: SOA (Service-oriented architecture) and AIO (All in one). SOA divides a complete application into several independent services, each of which provides a single function (such as session management, trade evaluation, user points, and etc.). These services expose their interfaces and communicate with each other through IPC mechanisms like RPC and WebAPI, altogether composing a complete application.

Conversely, AIO restricts an application within a separate unit. Different services within SOA behave as different components and modules. All components usually run within a single address space (the same process), and the codes of all components are usually maintained under the same project altogether.

The advantage of AIO is simplified deployment, eliminating the need for deploying multiple services and implementing high availability clustering for each service. AIO architecture has far higher efficiency than SOA, because it can avoid huge consumptions caused by IPC communications like network transmission and memory copy.

On the other hand, components within AIO are highly inter-dependent with poor reusability and replaceability, making maintenance and extension difficult. It is common that a rookie will spend a lot of effort and make many mistakes before getting the hang of a huge project which contains a large number of highly coupled components and modules. Even a veteran is prone to cause seemingly irrelevant functions being affected after modifying functions of a module, because of complicated inter-dependence among components.

The SOA architecture features complex deployment and configuration. In real cases, a large application is usually divided into hundreds of independent services. For example, a famous e-commerce website (among the top 5 in China) which fully embraces SOA has divided their Web application into tens of hundreds of services. We can imagine the huge amount of workload required to deploy hundreds of servers within high availability environment where multiple active data centers exist, and to configure these servers to establish coordination relationships among them. For example, the recent network outage with ctrip.com was followed by slow recovery due to its huge SOA architecture which comprises tens of hundreds of services.

Inefficient is another major disadvantage of SOA. From the logic flow perspective, almost every complete request from the client needs to flow through multiple services before the final result is generated and then returned to the client. Flowing through each service (through messaging middleware) is accompanied by multiple times of network and disk I/O operations. Thus several requests will cause long network delay accumulatively, resulting in bad user experience and high consumption of resources.

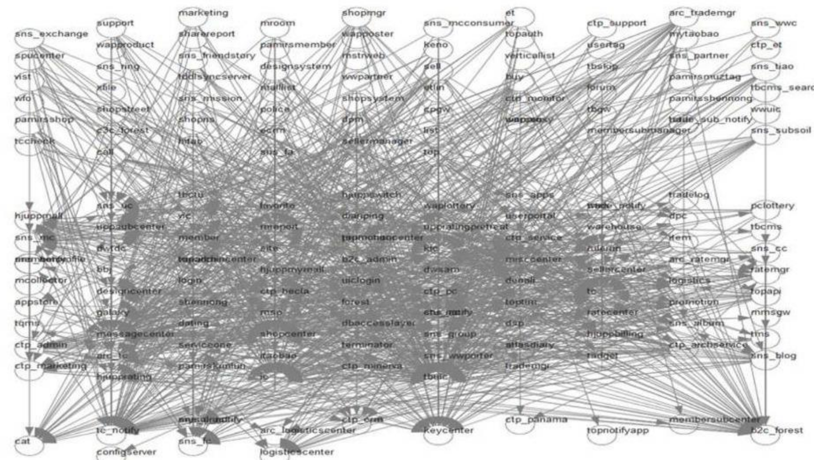



Figure 13 The Messy SOA Dependencies (Image from the Internet)

The responsibility to implement the support for cross-service distributed transaction will fall on the application developers, no matter each service is connected to a different DBMS or all services are connected to the same distributed DBMS system. The effort for implementing distributed transaction itself is more complex than most of common applications. Things will become more difficult when we try to add high availability and high reliability assurance to it, to achieve this goal, developers need to: utilize algorithms like Paxos/Raft or master/slave + arbiter for a single data shard; and employ algorithms like 2PC/3PC for transactions comprised of multiple data shards to achieve the ACID guarantee. Therefore, a compromise solution for implementing cross-service transactions within SOA applications is to guarantee the eventual consistency. This also requires extensive efforts, because it is not easy to implement consistency algorithms in a complex system.

Most of SOA systems usually need to utilize messaging middleware to implement message dispatching. This middleware can easily become a bottleneck if there are requirements for availability (part of nodes failed will not affect normal operation of the system), reliability (ensures messages are in order and never repeated/lost even when part of nodes failed), functionality (e.g., publish-subscribe pattern, distributing the tasks in a round-robin fashion) and etc.

The strength of SOA architecture lies with its high cohesion and low coupling characteristics. Services are provided through predefined IPC interface, and are running in an isolated way (usually in a separate node). SOA architecture has set a clear boundary for interfaces and functions, thus services can be easily reused and replaced (any new services that have compatible IPC interface can replace existing services).

From the point of view of software engineering and project management, each service itself has enough high cohesion, and its implemented functions are independent, SOA services are easier to maintain compared with interwoven AIO architecture. A developer only needs to take care of one specific service, and don't need to worry about any code modification or component replacement will

affect other consumers, as long as there is no incompatible change to the API.

An application composed of multiple independent services is easier to implement function modification or extension through the addition of new services or recombination of existing services.

### 5.4.2 nSOA Architecture

Through extensive exploration and practice with real projects, I have defined, implemented and improved the nano-SOA architecture which incorporates the strengths of both SOA and AIO. In nano-SOA, services that run independently are replaced by cross-platform plugins (IPlugin) that support hot-plugging. A plugin dynamically exposes (register) and hides (unregister) its function interfaces through (and only through) API Nexus, and consumes services provided by other plugins also through API Nexus.

nano-SOA fully inherits the high cohesion and low coupling characteristics of SOA architecture. Each plugin behaves like an independent service, has clear interface and boundary, and can be easily reused or replaced. It is comparable to SOA from the maintenance perspective. Each plugin can be developed and maintained separately, and a developer only needs to take care of his own plugin. By the addition of new plugins and recombination of existing plugins, nano-SOA makes things easier to modify or extend existing functions than SOA architecture.

nano-SOA is comparable to AIO with regard to performance and efficiency. All plugins run within the same process, thus calling another plugin through API Nexus does not need any I/O or memory copy or any other forms IPC consumption.

The deployment of nano-SOA is as simple as AIO. It can be deployed to a single node, and can achieve high availability and horizontal scaling by deploying only a single cluster. The configuration of nano-SOA is far simpler than SOA. Compared with AIO, configuring a list of modules to be loaded is the only thing added for nano-SOA. However, all the configurations for nano-SOA can be maintained in batch through utilizing a configuration management product. Streamlined deployment and configuration process can simplify operation and maintenance efforts, and also significantly facilitate establishing development and testing environments.

By using direct API calling through API Nexus, nano-SOA can avoid the dependence on messaging middleware to the maximum extent. We can also improve the parallel computing performance by plugging an inter-thread message queue (which is optimized through zero-copy and lock-free algorithms) on it. This has greatly increased throughput, reduced delay, and also eliminated huge efforts required for deploying and maintaining a high availability message dispatching cluster. nano-SOA has minimized the requirement for inter-node cooperation and communication, not imposing high demand for reliability, availability and functionality. In most cases, decentralized P2P protocol such as Gossip is adequate to meet these requirements. Sometimes, inter-node

communication can even be completely avoided.

From the nano-SOA perspective, DBC can be considered as a type of fundamental plugin for almost all server-end applications. It was implemented and added into libapidbc beforehand because of its wide use. libapidbc has established a firm foundation for the nano-SOA architecture, by offering the key components like IPlugin, API Nexus and DBC.

nano-SOA, SOA and AIO are not mutually exclusive options. In real use cases, users can work out the optimum design through combination of these three architecture patterns. For time-consuming asynchronous operations (like video transcoding) without the need to wait for a result to be returned, it is a preferred option to deploy it as an independent service to a dedicated server cluster with acceleration hardware installed, because most of the consumptions are used for video encoding and decoding. It is unnecessary to add it into an App Server as a plugin.

### 5.4.3 Port Switch Service (BYPSS)

BaiY Port Switch Service (BYPSS, pronounced "bypass") is designed for providing a high available, strongly consistent and high performance distributed coordination and message dispatching service which supports one trillion level ports, one million level nodes, and ten millions to billions of messages processed per second. The key concepts of the service include:

* Connection: Each client (a server within an application cluster) node maintains at least one TCP Keep-Alive connection with the port switch service.

* Port: Any number of ports can be registered for each connection. A port is described using a UTF-8 character string, and must be globally unique. Registering a port will fail if the port is already registered by another client node.

BYPSS offers the following API primitives:

* Waiting for Message (WaitMsg): Each node within the cluster should keep at least one TCP Keep-Alive connection with the BYPSS, and call this method for message pushing. This method upgrades the current connection from a message transmission connection to a message receiving connection.

  Each node number corresponds to only one message receiving connection. If a node attempts to generate two message receiving connections at the same time, the earlier connection will be disconnected, and all ports bound with that node will be unregistered.

* Relet: If BYPSS does not receive a relet request from a message receiving connection for a specified time period, it will treat the node as being offline, and will release all the ports

associated with this node. A relet operation is used for periodically providing heartbeat signals to BYPSS.

* Port Registration (RegPort): After a connection is established, the client should send request to BYPSS to register all the ports associated with the current node. A port registration request can contain any number of ports to be registered. BYPSS will return a list of ports (already occupied) that are failed to be registered. The caller can choose to subscribe port release notification for the ports failed to be registered.

  It is worth noting that each time a message receiving connection is re-established through calling WaitMsg, the server need to register all the relevant ports again.

* Port Un-registration (UnRegPort): To unregister the ports associated with the current node. A request can contain several ports for batch un-registration.

* Message Sending (SendMsg): To send a message (BLOB) to the specified port. The message format is transparent to BYPSS. If the specified port is an empty string, the message will be broadcasted to all nodes within BYPSS; sender can also specify multiple receiving ports to do a multicast. If the specified port does not exist, the message will be discarded quietly. The client can package multiple message sending commands within a single network request for batch sending, The BYPSS server will package messages sent to the same node automatically for batch message push.

* Port Query (QueryPort): To query node number and IP address associated with the node currently owns the specified port. This operation is used for service discovery with fault detection. This method is not needed for message sending (SendMsg) because the operation is automatically executed while delivering a message. A request can contain several ports for batch query.

* Node Query (QueryNode): To query information (e.g. IP address) associated with the specified node. This operation is mainly used for node resolving with fault detection. A request can contain several nodes for batch query.

Client connections within BYPSS are categorized into two types:

* **Message receiving connection (1:1):** It uses WaitMsg method for node registration and message pushing, and keeps occupying all ports belong to current node using Relet. Each node within the cluster should keep and only keep a single message receiving connection, which is a Keep-Alive connection. It is recommended to always keep the connection active and to complete Relet in a timely manner, because re-establishing a receiving connection will require service electing again (port registration).

- **Message sending connection (1:N):** All connections that are not upgraded using WaitMsg API are deemed as sending connections. They use primitives like RegPort, UnRegPort, SendMsg and QueryPort for non-pushing requests, without the need for using Relet to keep heartbeat. Each node within the cluster maintains a message sending connection pool, so that the worker threads can stay in communication with the port switch service.

Compared with traditional distribute coordination service and messaging middleware products, the port switch service has the following characteristics:

- Functionality: The port switch service integrates standard message routing function into distributed coordination services such as service electing (port registration), service discovery (send message and query port information), fault detection (relet timeout) and distribute locking (port registration and unregister notification). This high-performance message switch service has distributed coordination capabilities. Also, it can be purely used as a service electing and discovery service with fault detection, by using QueryPort and other interfaces.

- High-concurrency and high-performance: Implemented using C/C++/assembly languages; maintains a message buffer queue for each connection, and all port definitions and all messages to be forwarded are saved in memory (Full in-memory); there is no data replication or status synchronization between master node and slave node; message sending and receiving are implemented using pure async IO, enabling high-concurrency and high-throughput message dispatch performance.

- Scalability: When single-node performance gets a bottleneck, service can scaling out by cascading upper-level port switch service, similar to the three layers (access, aggregation, and core) switch architecture in IDC.

- Availability: High availability insurance by completing fault detection and master/slave switching within 5 milliseconds; quorum-based election algorithm, avoiding split brain due to network partition.

- Consistency: A port can be owned by only one client node at any given time. It is impossible that multiple nodes can succeed in registering and occupying the same port simultaneously.

- Reliability: All messages sent to an unregistered port (the port does not exist, or is unregistered or expired) are discarded quietly. The system ensures that all messages sent to registered ports are in order and unique, but messages may get lost in some extreme conditions:

  - Master/slave switching due to the port switch service is unavailable: All messages queued to be forwarded will be lost. All the already registered nodes need to register again, and all the already registered ports (services and locks) need election/acquirement

again (register).

- A node receiving connection is recovered from disconnection: After the message receiving connection was disconnected and then re-connected, all the ports that were ever registered for this node will become invalid and need to be registered again. During the time frame from disconnection to re-connection, all messages sent to the ports that are bound with this node and have not been registered by any other nodes will be discarded.

BYPSS itself is a message routing service that integrates fault detection, service election, service discovery, distributed lock, and other distributed coordination functionalities. It has achieved superior performance and concurrency at the premise of strong consistency, high availability and scalability (scale-out), by sacrificing reliability in extreme conditions.

BYPSS can be treated as a cluster coordination and message dispatching service customized for nano-SOA architecture. The major improvement of nano-SOA is, the model that each user request needs to involve multiple service nodes is improved so that most of user requests need to involve only different BMOD in the same process space.

In addition to making deployment and maintenance easier and the delay for request processing dramatically reduced, the above improvement also brings the following two benefits:

* In SOA, distributed transaction with multiple nodes involved and eventual consistency issues are simplified to a local ACID Transaction issue (from DBS perspective, transactions can still be distributed). This has greatly reduced complexity and enhanced consistency for distributed applications, and also has reduced inter-node communications (from inter-service IPC communications turned out to be inner-process pointer passing) and improved the overall efficiency.

* P2P node is not only easy to deploy and maintain, but also has simplified the distributed coordination algorithm. Communications among nodes are greatly reduced, because the tasks having high consistency requirements are completed within the same process space. Reliability of messaging middleware also becomes less demanding (the inconsistency due to message getting lost can be simply resolved by cache timeout or manual refreshing).

BYPSS allows for a few messages to be lost in extreme conditions, for the purpose of avoiding disk writing and master/slave copying and promoting efficiency. This is a reasonable choice for nano-SOA.

### Reliability Under Extreme Conditions

Traditional distributed coordination services are usually implemented using quorum-based

consensus algorithms like Paxos and Raft. Their main purpose is to provide applications with a high-availability service for accessing distributed metadata KV. The distributed coordination services such as distributed lock, message dispatching, configuration sharing, role election and fault detection are also offered. Common implementations of distributed coordination services include Google Chubby (Paxos), Apache ZooKeeper (Fast Paxos), etcd (Raft), Consul (Raft+Gossip), and etc.

Poor performance and high network consumption are the major problems with consensus algorithms like Paxos and Raft. For each access to these services, either write or read, it requires 2 to 4 times of network broadcasting within the cluster to confirm in voting manner that the current access is acknowledged by the quorum. This is because the master node needs to confirm it has the support from the majority while the operation is happening, and to confirm it remains to be the legal master node.

In real cases, the overall performance is still very low and has strong impact to network IO, though the read performance can be optimized by degradation the overall consistency of the system or adding a lease mechanism. If we look back at the major accidents happened in Google, Facebook or Twitter, many of them are caused by network partition or wrong configuration (human error). Those errors lead to algorithms like Paxos and Raft broadcasting messages in an uncontrollable way, thus driving the whole system crashed.

Furthermore, due to the high requirements of network IO (both throughput and latency), for Paxos and Raft algorithm, it is difficult (and expensive) to deploy a distributed cluster across multiple data centers with strong consistency (anti split brain) and high availability. For example: September 4, 2018, the cooling system failure of a Microsoft data center in South Central US caused Offi ce, Active Directory, Visual Studio and other services to be offline for nearly 10 hours; Google GCE service was disconnected for 12 hours and lost some data permanently on August 20, 2015; Alipay was interrupted for several hours on May 27, 2015, July 22, 2016 and Dec 5, 2019; July 22, 2013 and Mar 29, 2023 WeChat service interruption Hours; and May 2017 British Airways paralyzed for a few days and other major accidents both are due to the single IDC dependency.

Because most of the products that employ SOA architecture rely on messaging middleware to guarantee the overall consistency, they have strict requirements for availability (part of nodes failed will not affect normal operation of the system), reliability (ensures messages are in order and never repeated/lost even when part of nodes failed), and functionality (e.g., publish-subscribe pattern, distributing the tasks in a round-robin fashion). It is inevitable to use technologies that have low performance but require high maintenance cost, such as high availability cluster, synchronization and copy among nodes, and data persistence. Thus the message dispatching service often becomes a major bottleneck for a distributed system.

Compared with Paxos and Raft, BYPSS also provides distributed coordination services such as fault detection, service election, service discovery and distributed lock, as well as comparable consensus, high availability, and the capability of resisting split-brain. Moreover, by eliminates nearly all of the high

cost operations like network broadcast and disk IO, it has far higher performance and concurrency capability than Paxos and Raft. It can be used to build large-scale distributed cluster system across multiple data centers with no additional requirements of the network throughput and latency.

BYPSS allows for tens of millions of messages to be processed per second by a single node, and guarantees that messages are in order and never repeated, leaving common middleware far behind in terms of performance.

While having absolute advantages from performance perspective, BYPSS has to make a trade-off. The compromise is the reliability in extreme conditions (two times per year on average; mostly resulted from maintenance; controlled within low-load period; based on years of statistics in real production environments), which has the following two impacts to the system:

* For distributed coordination services, each time the master node offline due to a failure, all registered ports will forcibly become invalid, and all active ports need to be registered again.

  For example, if a distributed Web server cluster treat a user as the minimum schedule unit, and register a message port for each user who is logged in, after the master node of BYPSS is offline due to a failure, each node will know that all the ports it maintains have became invalid and it need to register all active (online) users again with the new BYPSS master.

  Fortunately, this operation can be completed in a batch. Through the batch registration interface, it is permitted to use a single request to register or unregister as much as millions of ports simultaneously, improving request processing efficiency and network utilization. On a Xeon processer (Haswell 2.0GHz) which was release in 2013, BYPSS is able to achieve a speed of 1 million ports per second and per core (per thread). Thanks to the concurrent hash table (each arena has its own full user mode reader/writer lock optimized by assembly) which was developed by us, we can implement linear extension by simply increasing the number of processor cores.

  Specifically, under an environment with 4-core CPU and Gigabit network adapter, BYPSS is capable of registering 4 millions of ports per second. Under an environment with 48-core CPU and 10G network adapter, BYPSS is able to support registering nearly 40 millions of ports per second (the name length of each of the ports is 16 bytes), almost reaching the limit for both throughput and payload ratio. There is almost no impact to system perforce, because the above scenarios rarely happen and re-registration can be done progressively as objects being loaded.

  To illustrate this, we consider the extreme condition when one billion users are online simultaneously. Though applications register a dedicated port (for determining user owner and for message distribution) for each of the users respectively, it is impossible that all these one billion users will press the refresh button simultaneously during the first second after

recovering from fault. Conversely, these online users will usually return to the server after minutes, hours or longer, which is determined by the intrinsic characteristics of Web applications (total number of online users = the number of concurrent requests per second × average user think time). Even we suppose all these users are returned within one minute (the average think time is one minute) which is a relatively tough situation, BYPSS only need to process 16 million registration requests per second, which means a 1U PC Server with 16-core Haswell and 10G network adapter is enough to satisfy the requirements.

As a real example, the official statistics show there were 180 million active users (DAU) in Taobao.com on Nov 11 (“double 11”), 2015, and the maximum number of concurrent online users is 45 million. We can make the conclusion that currently the peak number of concurrent users for huge sites is far less than the above mentioned extreme condition. BYPSS is able to support with ease even we increase this number tens of times.

* On the other hand, from message routing and dispatching perspective, all messages queuing to be forwarded will be lost permanently whenever the master node is offline due to a fault. Fortunately, the nano-SOA does not reply on messaging middleware to implement cross-service transaction consistency, thus does not have strict reliability requirements for message delivery.

  In the nSOA architecture, the worst consequence of the loss of messages is the corresponding user requests failed, but data consistency is still guaranteed and “half success” issue will never occur. This is enough for most use cases. Even Alipay and the china four largest banks’ E-bank applications occasionally have operation failures. This will not cause real problems only if there is no corruption with bank account data. User can just try again later one this case.

  Moreover, the BYPSS service has reduced the time that messages need to wait in the queue, through technologies such as optimized async IO and message batching. This message batching mechanism consists of message pushing and message sending:

  BYPSS offers a message batch sending interface, allowing for millions of messages to be submitted simultaneously within a single request. BYPSS also has a message batch pushing mechanism. If message surge occurs in a node and a large number of messages has arrived and are cumulated in the queue, BYPSS server will automatically enable message batch pushing mode, which packs plenty of messages into a single package, and pushes it to the destination node.

  The above mentioned batch processing mechanism has greatly improved message processing efficiency and network utilization. It guarantees the server-end message queue is almost empty in most cases, and thus has reduced the possibility of message loss when the master node is offline.

Although the probability of message loss is very low, and the nano-SOA architecture does not reply on messaging middleware to guarantee reliability, there are a few cases which have high requirements for message delivery. The following solutions can satisfy these requirements:

- Implement the acknowledgment and timeout retransmission mechanism by self: After sending a message to the specified port, the sender will wait for a receipt to be returned. If no receipt is received during the specified time period, it will send the request again.

- Directly send RPC request to the owner node of the port: The message sender obtains IP address of the owner node using port query commands, and then establishes direct connection with this owner node, sends a request and waits for the result to be returned. During the process, BYPSS is responsible for service election and discovery, and does not route messages directly. This solution is also recommended for inter-node communications with large volume of data stream exchanges (e.g., video streaming and video transcoding, deep learning), to avoid BYPSS becoming an IO bottleneck.

- Use third-party messaging middleware: If there is a large quantity of message delivery requests that have strict reliability requirements and using complex rules, it is suggested to deploy a third-party message dispatching cluster to process these requests.

  Of course, there is actually no completely reliable message queue service (which can ensure that messages are not lost, duplicated, or out of order). Therefore, when it is really necessary to implement distributed transactions across application server nodes, it is recommended to implement BYPSS and BYDMQ with algorithms such as SAGA. For details, see: 5.4.4 Distributed Message Queue Service (BYDMQ).

In brief, we can treat BYPSS as a cluster coordination and message dispatching service customised for the nano-SOA architecture. BYPSS and nano-SOA are mutually complementary. BYPSS is ideal for implementing a high performance, high availability, high reliability and strong consistency distributed system with nano-SOA architecture. It can substantially improve the overall performance of the system at the price of slightly affecting system performance under extreme conditions.

## BYPSS Characteristics

The following table gives characteristic comparisons between BYPSS and some distributed coordination products that utilize traditional consensus algorithms like Paxos and Raft.

| Item | BYPSS | ZooKeeper, Consul, etcd… |
|---|---|---|
| Availability | High availability; supports multiple-active IDC. | High availability, but difficult to support multi-active IDC. |

| Item | BYPSS | ZooKeeper, Consul, etcd... |
|---|---|---|
| Consistency | Strong consistency; the master node is elected by the quorum.<br>Both read and write operations provide strong consistency guarantees. | Strong write consistency; multi replica.<br>In order to improve performance, most implements sacrifice consistency when reading (only Consul supports configuring strong consistent read mode). |
| Concurrency | Tens of millions of concurrent connections; hundreds of thousands of concurrent nodes. | Up to 5,000 nodes. |
| Capacity | Each 10GB memory can hold at least 100 million message ports; each 1TB memory can hold at least ten billion message ports; two-level concurrent Hash table structure allows capacity to be linearly extended to PB level. | Usually supports up to hundreds of thousands of key-value pairs; this number is even smaller when change notification is enabled. |
| Delay | The delay per request within the same IDC is at sub-millisecond level (0.5ms in Aliyun.com); the delay per request for different IDCs within the same region is at millisecond level (2ms in Aliyun.com). | Because each request requires 2 to 4 times of network broadcasting and multiple times of disk I/O operations, the delay per operation within the same IDC is over 10 milliseconds; the delay per request for different IDCs is more longer (see the following paragraphs). |
| Performance | Each 1Gbps bandwidth can support nearly 4 million times of port registration and unregistration operations per second. On an entry-level Haswell processor (2013), each core can support 1 million times of the above mentioned operations per second. The performance can be linearly extended by increasing bandwidth and processor core. Up to 300 million operations per second on modern processors and dual-port 40 Gigabit NICs. | The characteristics of the algorithm itself make it impossible to support batch operations; less than 200 requests per second under the same test conditions. (Because each atomic operation requires 2 to 4 times of network broadcasting and multiple times of disk I/O operations, it is meaningless to add the batch operations supporting.) |
| Network utilization | High network utilization: both the server and client have batch packing capabilities for port registration, port unregistration, port query, node query and message sending; network payload ratio can be close to 100%. | Low network utilization: each request use a separate package (TCP Segment, IP Packet, Network Frame), Network payload ratio is typically less than 5%. |
| Scalability | Yes: can achieve horizontal scaling in cascading style. | No: more nodes the cluster contains (the range for broadcasting and disk I/O operations becomes wider), the worse the performance is. |
| Partition | The system goes offline when there is no | The system goes offline when there is |

| Item | BYPSS | ZooKeeper, Consul, etcd... |
|---|---|---|
| tolerance | quorum partition, but broadcast storm will not occur. | no quorum partition. It is possible to produce a broadcast storm aggravated the network failure. |
| Message dispatching | Yes and with high performance: both the server and client support automatic message batching. | None. |
| Configuration Management | No: BYPSS believes the configuration data should be managed by dedicate products like Redis, MySQL, MongoDB and etc. Of course the distribute coordination tasks of these CMDB products (e.g. master election) can still be done by the BYPSS. | Yes: Can be used as a simple CMDB. This confusion on the functions and responsibilities making capacity and performance worse. |
| Fault recovery | Need to re-generate a state machine, which can be completed at tens of millions of or hundreds of millions of ports per second; practically, this has no impact on performance. | There is no need to re-generate a state machine. |

Among the above comparisons, delay and performance mainly refers to write operations. This is because almost all of the meaningful operations associated with a typical distributed coordination tasks are write operations. For example:

| Operations | From service coordination perspective | From distributed lock perspective |
|---|---|---|
| Port registration | Success: service election succeeded; becomes the owner of the service.<br>Failed: successfully discover the current owner of the service. | Success: lock acquired successfully.<br>Failed: failed to acquire the lock, returning the current lock owner. |
| Port unregistration | Releases service ownership. | Releases lock. |
| Unregistration notification | The service has offline; can update local query cache or participate in service election. | Lock is released; can attempt to acquire the lock again. |

As shown in the above table, the port registration in BYPSS corresponds to “write/create KV pair” in traditional distributed coordination products. The port unregistration corresponds to “delete KV pair”, and the unregistration notification corresponds to “change notification”.

To achieve maximum performance, we will not use read-only operations like query in production environments. Instead, we hide query operations in write requests like port registration. If the request is successful, the current node will become the owner. If registration failed, the current owner of the requested service will be returned. This has also completed the read operations like owner query (service discovery / name resolution).

It is worth noting that even a write operation (e.g., port registration) failed, it is still accompanied by a successful write operation. The reason is: There is a need to add the current node that initiated the request into the change notification list of specified item, in order to push notification messages to all interested nodes when a change such as port unregistration happens. So the write performance differences greatly affect the performance of an actual application.

## HAC Manager Utility

BYPSS also includes a companion tool called HAC Manager. This utility use BYPSS to complete the service election and fault detection tasks. If multiple HAC manager instances competition the ownership of same service simultaneously, only the winner can start the service/daemon by executing a user specified command. Accordingly, the service/daemon will be stopped when it lose the ownership.

With the distributed storage technology such as DRBD, HAST, DataKeeper, DFS, Ceph, GlusterFS, Lustre and others , or shared storage solutions like SAN, HAC Manager can easily convert a single point service (e.g.: the traditional SQL DB, Full Test Search Engine, Report Generating Service, and etc.) to the high availability cluster (HAC) without split brain issues.

Note: The above nano-SOA architecture and the BYPSS distributed coordination algorithm are all subject to a number of national and international patents protections.

## BYPSS based High performance cluster

From the high-performance cluster (HPC) perspective, the biggest difference between BYPSS and the traditional distributed coordination products (described above) is mainly reflected in the following two aspects:

1. High performance: BYPSS eliminates the overhead of network broadcasting, disk IO, add the batch support operations and other optimizations. As a result, the overall performance of the distributed coordination service has been increased by tens of thousands of times.

2. High capacity: at least 100 million message ports per 10GB memory, due to the rational use of the data structure such as concurrent hash table, the capacity and processing performance can be linearly scaled with the memory capacity, the number of processor cores, the network card speed and other hardware upgrades.

Due to the performance and capacity limitations of traditional distributed coordination services, in a classical distributed cluster, the distributed coordination and scheduling unit is typically at the service

or node level. At the same time, the nodes in the cluster are required to operate in stateless mode as far as possible .The design of service node stateless has low requirement on distributed coordination service, but also brings the problem of low overall performance and so on.

BYPSS, on the other hand, can easily achieve the processing performance of tens of millions of requests per second, and trillions of message ports capacity. This provides a good foundation for the fine coordination of distributed clusters. Compared with the traditional stateless cluster, BYPSS-based fine collaborative clusters can bring a huge overall performance improvement.

User and session management is the most common feature in almost all network applications. We first take it as an example: In a stateless cluster, the online user does not have its owner server. Each time a user request arrives, it is routed randomly by the reverse proxy service to any node in the backend cluster. Although LVS, Nginx, HAProxy, TS and other mainstream reverse proxy server support node stickiness options based on Cookie or IP, but because the nodes in the cluster are stateless, so the mechanism simply increases the probability that requests from the same client will be routed to a certain backend server node and still cannot provide a guarantee of ownership. Therefore, it will not be possible to achieve further optimizations.

While benefiting from BYPSS's outstanding performance and capacity guarantee, clusters based on BYPSS can be coordinated and scheduled at the user level (i.e.: registering one port for each **active user**) to provide better overall performance. The implementation steps are:

1. As with the traditional approach, when a user request arrives at the reverse proxy service, the reverse proxy determines which back-end server node the current request should be forwarded to by the HTTP cookie, IP address, or related fields in the custom protocol. If there is no sticky tag in the request, the lowest-load node in the current back-end cluster is selected to process the request.

2. After receiving the user request, the server node checks to see if it is the owner of the requesting user by looking in the local memory table.

    a) If the current node is already the owner of the user, the node continues processing the user request.

    b) If the current node is not the owner of the user, it initiates a RegPort request to BYPSS, attempting to become the owner of the user. This request should be initiated in batch mode to further improve network utilization and processing efficiency.

        i. If the RegPort request succeeds, the current node has successfully acquired the user's ownership. The user information can then be loaded from the backend database into the local cache of the current node (which should be optimized using bulk load) and continue processing the user request.

ii. If the RegPort request fails, the specified user's ownership currently belongs to another node. In this case, the sticky field that the reverse proxy can recognize, such as a cookie, should be reset and point it to the correct owner node. Then notifies the reverse proxy service or the client to retry.

Compared with traditional architectures, taking into account the stateless services also need to use MySQL, Memcached or Redis and other stateful technologies to implement the user and session management mechanism, so the above implementation does not add much complexity, but the performance improvement is very large, as follows:

| Item | BYPSS HPC | Traditional Stateless Cluster |
|---|---|---|
| 1<br>Op. | Eliminating the deployment and maintenance costs of the user and session management cluster. | Need to implement and maintain the user management cluster separately, and provides dedicated high-availability protection for the user and session management service. Increases the number of fault points, the overall system complexity and the maintenance costs. |
| 2<br>Net. | Nearly all user matching and session verification tasks for a client request can be done directly in the memory of its owner node. Memory access is a nanosecond operation, compared to millisecond-level network query delay, performance increase of more than 100,000 times. While effectively reducing the network load in the server cluster. | It is necessary to send a query request to the user and session management service over the network each time a user identity and session validity is required and wait for it to return a result. Network load and the latency is high.<br><br>Because in a typical network application, most user requests need to first complete the user identification and session authentication to continue processing, so it is a great impact on overall performance. |
| 3<br>Cch. | Because each active user has a definite owner server at any given time, and the user is always inclined to repeat access to the same or similar data over a certain period of time (such as their own properties, the product information they have just submitted or viewed, and so on). As a result, the server's local data caches tend to have high locality and high hit rates.<br><br>Compared with distributed caching, the advantages of | No dedicated owner server, user requests can be randomly dispatched to any node in the server cluster; Local cache hit rate is low; Repeatedly caching more content in different nodes; Need to rely on the distributed cache at a higher cost.<br><br>The read pressure of the backend |

| Item | BYPSS HPC | Traditional Stateless Cluster |
| --- | --- | --- |
| | local cache is very obvious:<br>1. Eliminates the network latency required by query requests and reduces network load (See "Item 2" in this table for details).<br>2. Get the expanded data structures directly from memory, without a lot of data serialization and deserialization work.<br>3. Only the owner node caches the corresponding data, which also avoids the inconsistency between the distributed cache and the DB, and provides a strong consistency guarantee.<br><br>The server's local cache hit rate can be further improved if the appropriate rules for user owner selection can be followed, for example:<br><br>a) Group users by tenant (company, department, site);<br>b) Group users by region (geographical location, map area in the game);<br>c) Group users by interest characteristics (game team, product preference).<br><br>And so on, and then try to assign users belonging to the same group to the same server node (or to the same set of nodes). Obviously, choice an appropriate user grouping strategy can greatly enhance the server node's local cache hit rate.<br><br>This allows most of the data associated with a user or a group of users to be cached locally. This not only improves the overall performance of the cluster, but also eliminates the dependency on the distributed cache. The read pressure of the backend database is also greatly reduced. | database server is high. Additional optimizations are required, such as horizontal partitioning, vertical partitioning, and read / write separation.<br><br>There is an unavoidable data inconsistency problem between the distributed cache and the DB (unless a protocol such as Paxos/Raft is used between the distributed cache and the DB to ensure consistency, but the huge performance loss that follows will also make the distributed cache meaningless -- this is even slower than not having a distributed cache). |
| 4<br>Upd. | Due to the deterministic ownership solution, any user can be ensured to be globally serviced by a particular owner node within a given time period in the cluster. Coupled with the fact that the probability of a sudden failure of a modern PC server is also very low. | Cumulative write optimization and batch write optimization cannot be implemented because each request from the user may be forwarded to a different server node for processing. The write pressure of the backend database |

| Item | BYPSS HPC | Traditional Stateless Cluster |
|---|---|---|
| | Thus, the frequently changing user properties with lower importance or timeliness can be cached in memory. The owner node can update these changes to the database in batches until they are accumulated for a period of time.<br><br>This can greatly reduce the write pressure of the backend database.<br><br>For example, the shop system may collect and record user preference information in real time as the user browses (e.g., views each product item). The workload is high if the system needs to immediately update the database at each time a user views a new product. Also considering that due to hardware failure, some users who occasionally lose their last few hours of product browsing preference data are perfectly acceptable. Thus, the changed data can be temporarily stored in the local cache of the owner node, and the database is updated in batches every few hours.<br><br>Another example: In the MMORPG game, the user's current location, status, experience and other data values are changing at any time. The owner server can also accumulate these data changes in the local cache and update them to the database in batches at appropriate intervals (e.g.: every 5 minutes).<br><br>This not only significantly reduces the number of requests executed by the backend database, but also eliminates a significant amount of disk flushing by encapsulating multiple user data update requests into a single batch transaction, resulting in further efficiency improvements.<br><br>In addition, updating user properties through a dedicated owner node also avoids contention issues when multiple nodes are simultaneously updating the same object in a stateless cluster. It further improves database performance. | is very high.<br><br>A plurality of nodes may compete to update the same record simultaneously, further increasing the burden on the database.<br><br>Additional optimizations are required, such as horizontal partitioning and vertical partitioning, However, these optimizations will also result in side effects such as "need to implement distributed transaction support at the application layer." |

| Item | BYPSS HPC | Traditional Stateless Cluster |
|---|---|---|
| 5<br>Push | Since all sessions initiated by the same user are managed centrally in the same owner node, it is very convenient to push an instant notification message (Comet) to the user.<br><br>If the object sending the message is on the same node as the recipient, the message can be pushed directly to all active sessions belong to the recipient.<br><br>Otherwise, the message may simply be delivered to the owner node of the recipient. Message delivery can be implemented using BYPSS (send messages to the corresponding port of the recipient directly, should enable the batch message sending mechanism to optimize). Of course, it can also be done through a dedicated high-performance message middleware such as BYDMQ.<br><br>If the user's ownership is grouped as described in item 3 of this table, the probability of completing the message push in the same node can be greatly improved. This can significantly reduce the communication between servers.<br><br>Therefore, we encourage customizing the user grouping strategy based on the actual situation for the business properly. A reasonable grouping strategy can achieve the desired effect, that is, most of the message push occurs directly in the current server node.<br><br>For example, for a game application, group players by map object and place players within the same map instance to the same owner node - Most of the message push in the traditional MMORPG occurs between players within the same map instance (AOI).<br><br>Another example: For CRM, HCM, ERP and other SaaS applications, users can be grouped according to the company, place users belong to the same enterprise to the same owner node - It is clear that for such | Because different sessions of the same user are randomly assigned to different nodes, there is a need to develop, deploy, and maintain a specialized message push cluster. It also needs to be specifically designed to ensure the high performance and high availability of the cluster.<br><br>This not only increases the development and maintenance costs, but also increases the internal network load of the server cluster, because each message needs to be forwarded to the push service before it can be sent to the client. The processing latency of the user request is also increased. |

| Item | BYPSS HPC | Traditional Stateless Cluster |
|---|---|---|
| | enterprise applications, nearly 100% of the communications are from within the enterprise members.<br><br>The result is a near 100% local message push rate: the message delivery between servers can almost be eliminated. This significantly reduces the internal network load of the server cluster. | |
| 6<br>Bal. | Clusters can be scheduled using a combination of active and passive load balancing.<br><br>Passive balancing: Each node in the cluster periodically unloads users and sessions that are no longer active, and notifies the BYPSS service to bulk release the corresponding ports for those users. This algorithm implements a macro load balancing (in the long term, clusters are balanced).<br><br>Active balancing: The cluster selects the load balancing coordinator node through the BYPSS service. This node continuously monitors the load of each node in the cluster and sends instructions for load scheduling (e.g.: request node A to transfer 5,000 users owned by it to Node B). Unlike the passive balancing at the macro level, the active balancing mechanism can be done in a shorter time slice with quicker response speed.<br><br>Active balancing is usually effective when some of the nodes in the cluster have just recovered from the failure (and therefore are in no-load state), it reacts more rapidly than the passive balancing. For Example: In a cluster that spans multiple active IDCs, an IDC resumes on-line when a cable fault has just been restored. | If the node stickiness option is enabled in the reverse proxy, its load balancing is comparable to the BYPSS cluster's passive balancing algorithm.<br><br>If the node stickiness option in the reverse proxy is not enabled, its balance is less than the BYPSS active balance cluster when recovering from a failure. At the same time, In order to ensure that the local cache hit rate and other performance indicators are not too bad, the administrator usually does not disable the node sticky function.<br><br>In addition, SOA architecture tends to imbalance between multiple services, resulting in some services overload, and some light-load, nano-SOA cluster without such shortcomings. |

It is worth mentioning that such a precise collaborative algorithm does not cause any loss in availability of the cluster. Consider the case where a node in a cluster is down due to a failure: At this point, the BYPSS service will detect that the node is offline and automatically release all users belonging to that node. When one of its users initiates a new request to the cluster, the request will be routed to the lightest node in the current cluster (See step 2-b-i in the foregoing). This process is

transparent to the user and does not require additional processing logic in the client.

The above discussion shows the advantages of the BYPSS HPC cluster fine coordination capability, taking the user and session management functions that are involved in almost all network applications as an example. But in most real-world situations, the application does not just include user management functions. In addition, applications often include other objects that can be manipulated by their users. For example, in Youku.com, tudou.com, youtube.com and other video sites, in addition to the user, at least some "video objects" can be played by their users.

Here we take the "video object" as an example, to explore how the use the fine scheduling capabilities of BYPSS to significantly enhance cluster performance.

In this hypothetical video-on-demand application, similar to the user management function described above, we first select an owner node for each **active video object** through the BYPSS service. Secondly, we will divide the properties of a video object into following two categories:

1. Common Properties: Contains properties that are less updated and smaller in size. Such as video title, video introduction, video tag, video author UID, video publication time, ID of the video stream data stored in the object storage service (S3 / OSS), and the like. These properties are all consistent with the law of “read more write less”, or even more, most of these fields cannot be modified after the video is published.

   For such small-size, less-changed fields, they can be distributed in the local cache of each server node in the current cluster. Local memory caches have advantages such as high performance, low latency, and no need for serialization, plus the smaller size of the objects in cache. Combined with strategies to further enhance the cache locality, such as user ownership grouping, the overall performance can be improved effectively through a reasonable memory overhead (see below).

2. Dynamic Properties: Contains all properties that need to be changed frequently, or larger in size. Such as: video playback times, "like" and "dislike" times, scores, number of favours, number of comments, and contents of the discussion forum belong to the video and so on.

   We stipulate that such fields can only be accessed by the owner of the video object. Other nodes need to send a request to the corresponding owner to access these dynamic attributes.

   This means that we use the election mechanism provided by BYPSS to hand over properties that require frequent changes (updating the database and performing cache invalidation) or requiring more memory (high cache cost) to the appropriate owner node for management and maintenance. This result in a highly efficient distributed computing and distributed caching mechanism, greatly improving the overall performance of the application (see below).

In addition, we also stipulate that any write operation to the video object (whether for common or dynamic properties) must be done by its owner. A non-owner node can only read and cache the common properties of a video object; it cannot read dynamic properties and cannot perform any update operations.

Therefore, we can simply infer that the general logic of accessing a video object is as follows:

1. When a common property read request arrives at the server node, the local cache is checked. If the cache hit, then return the results directly. Otherwise, the common part of the video object is read from the backend database and added to the local cache of current node.

2. When an update request or dynamic property read request arrives, it checks whether the current node is the owner of the corresponding video object through the local memory table.

   a) If the current node is already the owner of the video, the current node continues to process this user request: For read operations, the result is returned directly from the local cache of the current node; depending on the situation, write operations are either accumulated in the local cache or passed directly to the backend database (the local cache is also updated simultaneously).

   b) If the current node is not the owner of the video but finds an entry matching the video in the local name resolution cache table, it forwards the current request to the corresponding owner node.

   c) If the current node is not the owner of the video and does not find the corresponding entry in the local name resolution cache table, it initiates a RegPort request to BYPSS and tries to become the owner of the video. This request should be initiated in batch mode to further improve network utilization and processing efficiency.

      i. If the RegPort request succeeds, then the current node has successfully acquired the ownership of the video. At this point, the video information can be loaded from the backend database into the local cache of the current node (which should be optimized using bulk loading) and continue processing the request.

      ii. If the RegPort request fails, the specified video object is already owned by another node. In this case, the video and its corresponding owner ID are added to the local name resolution cache table, and the request is forwarded to the corresponding owner node for processing.

      Note: Because BYPSS can push notifications to all nodes that are interested in this event each time the port is unregistered (whether due to explicit ownership release, or due to node failure offline). So the name resolution cache table does not require

a TTL timeout mechanism similar to the DNS cache. It only needs to delete the corresponding entry if the port deregistration notice is received or the LRU cache is full. This not only improves the timeliness and accuracy of entries in the lookup table, but also effectively reduces the number of RegPort requests that need to be sent, improving the overall performance of the application.

Compared with the classic stateless SOA cluster, the benefits of the above design are as follows:

| Item | BYPSS HPC | Traditional Stateless Cluster |
| --- | --- | --- |
| 1<br>Op. | The distributed cache structure is based on ownership, it eliminates the deployment and maintenance costs of distributed cache clusters such as Memcached and Redis. | Distributed cache clusters need to be implemented and maintained separately, increase overall system complexity. |
| 2<br>Cch. | A common property read operation will hit the local cache. If the owner node selection strategy that "Group users according to their preference characteristics" is used, then the cache locality will be greatly enhanced. Furthermore, the local cache hit rate will increase and the cache repetition rate in the different nodes of the cluster will decrease.<br><br>As mentioned earlier, compared to distributed cache, the local cache can eliminate network latency, reduce network load, avoid frequent serialization and deserialization of data structures, and so on.<br><br>In addition, dynamic properties are implemented using distributed cache based on ownership, which avoids the problems of frequent invalidation and data inconsistency of traditional distributed caches. At the same time, because the dynamic properties are only cached on the owner node, the overall memory utilization of the system is also significantly improved. | No dedicated owner server, user requests can be randomly dispatched to any node in the server cluster; Local cache hit rate is low; Repeatedly caching more content in different nodes; Need to rely on the distributed cache at a higher cost.<br><br>The read pressure of the backend database server is high. Additional optimizations are required, such as horizontal partitioning, vertical partitioning, and read / write separation.<br><br>Furthermore, even the CAS atomic operation based on the Revision field and other similar improvements can be added to the Memcached, Redis and other products. These independent distributed cache clusters still do not provide strong consistency guarantees (i.e.: The data in the cache may not be consistent with the records in the backend database). |
| 3<br>Upd. | Due to the deterministic ownership solution, It is ensured that all write and dynamic property read operations of video objects are globally serviced by a particular owner node within a given time period in the cluster. Coupled with the fact that the probability | Cumulative write optimization and batch write optimization cannot be implemented because each request may be forwarded to a different server node for processing. The write pressure of the |

| Item | BYPSS HPC | Traditional Stateless Cluster |
|---|---|---|
| | of a sudden failure of a modern PC server is also very low.<br><br>Thus, the frequently changing dynamic properties with lower importance or timeliness can be cached in memory. The owner node can update these changes to the database in batches until they are accumulated for a period of time.<br><br>This can greatly reduce the write pressure of the backend database.<br><br>For example: the video playback times, "like" and "dislike" times, scores, number of favours, references and other properties will be changed intensively with every user clicks. If the system needs to update the database as soon as each associated click event is triggered, the workload is high. Also considering that due to hardware failure, the loss of a few minutes of the above statistics is completely acceptable. Thus, the changed data can be temporarily stored in the local cache of the owner node, and the database is updated in batches every few minutes.<br><br>This not only significantly reduces the number of requests executed by the backend database, but also eliminates a significant amount of disk flushing by encapsulating multiple video data update requests into a single batch transaction, resulting in further efficiency improvements.<br><br>In addition, updating video properties through a dedicated owner node also avoids contention issues when multiple nodes are simultaneously updating the same object in a stateless cluster. It further improves database performance. | backend database is very high.<br><br>A plurality of nodes may compete to update the same record simultaneously, further increasing the burden on the database.<br><br>Additional optimizations are required, such as horizontal partitioning and vertical partitioning, However, these optimizations will also result in side effects such as "need to implement distributed transaction support at the application layer." |
| 4<br>Bal. | Clusters can be scheduled using a combination of active and passive load balancing.<br><br>Passive balancing: Each node in the cluster periodically unloads videos that are no longer active, | When recovering from a fault, the balance is less than the BYPSS active balanced cluster. However, there is no significant difference under normal circumstances. |

| Item | BYPSS HPC | Traditional Stateless Cluster |
| --- | --- | --- |
| | and notifies the BYPSS service to bulk release the corresponding ports for those videos. This algorithm implements a macro load balancing (in the long term, clusters are balanced).<br><br>Active balancing: The cluster selects the load balancing coordinator node through the BYPSS service. This node continuously monitors the load of each node in the cluster and sends instructions for load scheduling (e.g.: request node A to transfer 10,000 videos owned by it to Node B). Unlike the passive balancing at the macro level, the active balancing mechanism can be done in a shorter time slice with quicker response speed.<br><br>Active balancing is usually effective when some of the nodes in the cluster have just recovered from the failure (and therefore are in no-load state), it reacts more rapidly than the passive balancing. For Example: In a cluster that spans multiple active IDCs, an IDC resumes on-line when a cable fault has just been restored. | In addition, SOA architecture tends to imbalance between multiple services, resulting in some services overload, and some light-load, nano-SOA cluster without such shortcomings. |

Similar to the previously mentioned user management case, the precise collaboration algorithm described above does not result in any loss of service availability for the cluster. Consider the case where a node in a cluster is down due to a failure: At this point, the BYPSS service will detect that the node is offline and automatically release all videos belonging to that node. When a user accesses these video objects next time, the server node that received the request takes ownership of the video object from BYPSS and completes the request. At this point, the new node will (replace the offline fault node) becomes the owner of this video object (See step 2-c-i in the foregoing). This process is transparent to the user and does not require additional processing logic in the client.

The above analysis of "User Management" and "Video Services" is just an appetizer. In practical applications, the fine resource coordination capability provided by BYPSS through its high-performance, high-capacity features can be applied to the Internet, telecommunications, Internet of Things, big data processing, streaming computing and other fields.

We will continue to add more practical cases, for your reference.

## 5.4.4 Distributed Message Queue Service (BYDMQ)

The BaiY Distributed Message Queuing Service (BYDMQ, pronounced "by dark") is a distributed message queue service with strong consistency, high availability, high throughput, low latency and linear scale-out. It can support a single point of tens of millions of concurrent connections and a single point of tens of millions of message forwarding performance per second, and supports linear horizontal scaling out of the cluster.

BYDMQ itself also relies on BYPSS to perform distributed coordination such as service elections, service discovery, fault detection, distributed locks, and message dispatching. Although BYPSS also includes high-performance message routing and dispatching functions, its main design goal is to deliver distributed coordination-related control-type signals such as task scheduling. On the other hand, BYDMQ is focused on high-throughput, low-latency, large-scale business message dispatching. After the business related messages are offloaded to BYDMQ, the work pressure of BYPSS can be greatly reduced.

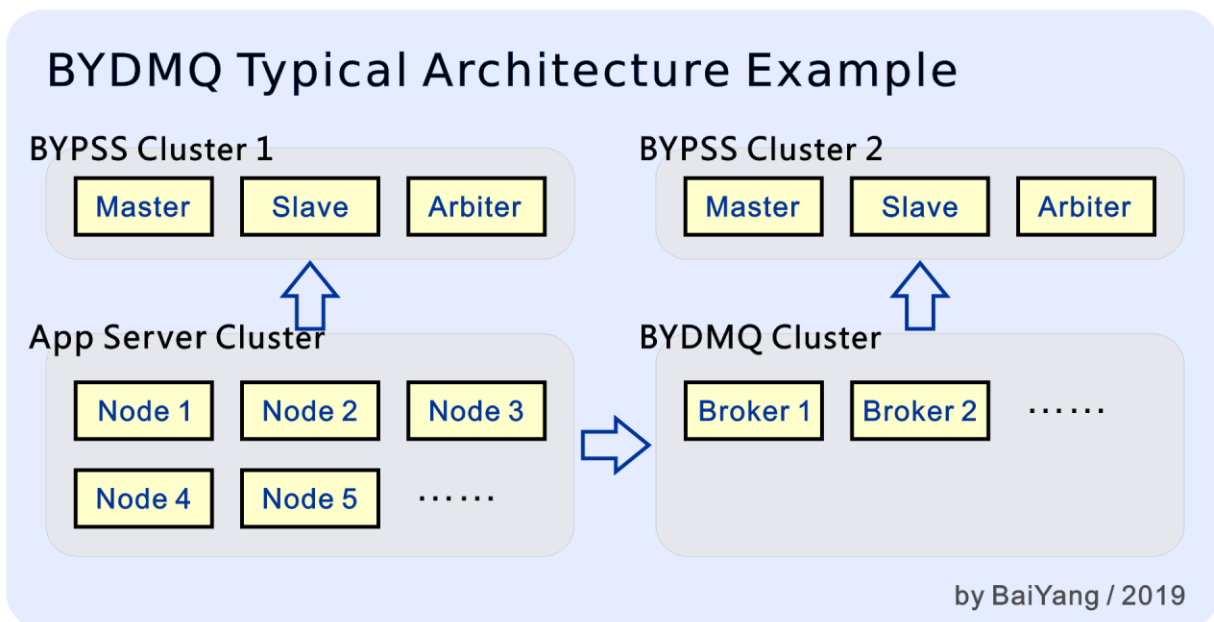



Figure 14

As shown in Figure 14, in the typical use case, BYDMQ and App Server cluster each have their own BYPSS cluster which are responsible for their respective distributed coordination tasks. The App cluster relies on BYPSS1 to complete distributed coordination, while its message communication relies on the BYDMQ cluster.

However, App Server and BYDMQ clusters can also share the same BYPSS service in a development / test environment, or a production environment with small business volume. It should also be noted that the "independent cluster" described herein refers only to logical independence. Physically, even two logically independent BYPSS clusters can share physical resources. For example, an

Arbiter node can be shared by multiple BYPSS clusters; even Master and Slave nodes in two BYPSS clusters can become backup nodes of each other. This simplifies the operation and maintenance management burden, and effectively save resources such as server hardware and energy consumption.

Before we continue to introduce the main features of BYDMQ, we first need to clarify a concept: the reliability of message queue (message middleware, MQ). As we all know, "reliable messaging" consists of three elements: the delivery process can be called reliable only if there are no missing, unordered or duplicate messages. Regrettably, there is currently no real message queue product that satisfies the above three conditions at the same time. Or in other words, It is impractical to implement a message queue product that satisfies all of the above three elements within an acceptable cost range.

To illustrate this issue, consider the following case:

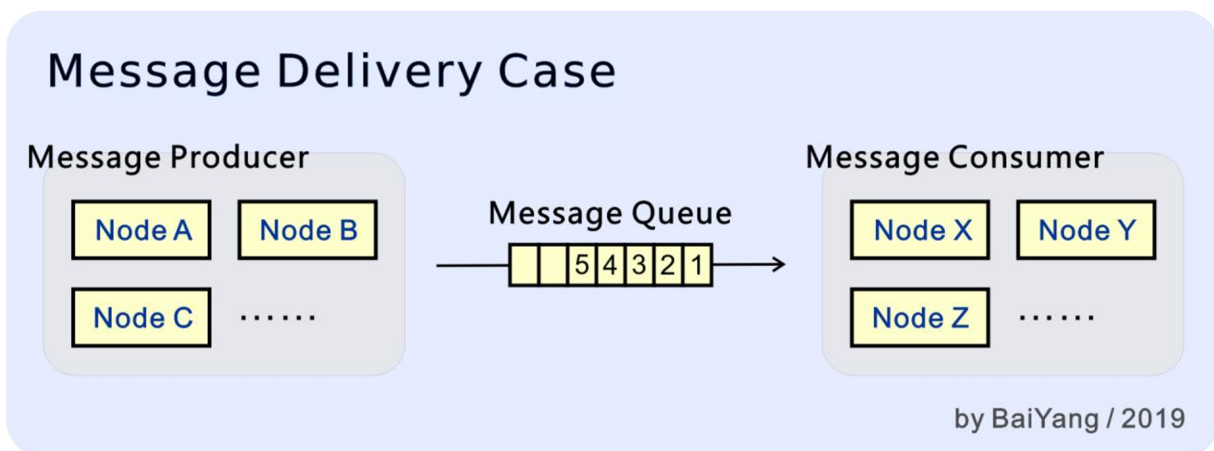



Figure 15

As shown in the figure above, in this case, the message producer consists of nodes A, B, and C, the message consumer contains nodes X, Y, and Z, and the producers deliver messages to consumers through a message queue. Now the message producer has produced 5 messages and successfully submitted them to the message queue in sequence.

Under such circumstances, let's discuss the reliability of message delivery one by one:

* **The message should not be lost**: this is the simplest one among the three elements. It can be split into two steps to discuss:
  - Storage reliability: It can be ensured by synchronously replicating each message to other nodes in the message queue service and make sure the data is written to disk. It also needs to use distributed consensus protocols such as Paxos and Raft to ensure consistency between multiple replicas. However, it is clear that this approach can greatly reduce the performance of the Message Queuing Services (thousands or even tens of thousands of times) due to increased disk IO, network replication, and consensus voting

steps compared to a pure memory solution without replicas.

- ACK mechanism: When the producer submits a message to the message queue or the message queue delivers the message to the consumer, an ACK mechanism is added to confirm that the message delivery is successful. The sender resends (reposts) the message if it does not receive the ACK mechanism within the specified time.

  Although the above two steps can ensure that the message will not be lost (at least once delivery), it can be seen that its overhead is very large, and the performance degradation is very significant.

  At the same time, it should be noted that the technologies such as fault detection and master-slave switching between multiple replicas, and timeout retransmission during message transmission and reception will introduce their respective delays. And the delay introduced in each of these steps usually exceeds a few seconds.

  In most of the real user scenario, these extra delays make the "message not lost" guarantee not so usefully: Today's users rarely wait patiently for a long time after initiating a request (such as opening a link, submitting a form, etc.) - if they still don't get a response after waiting a few seconds, they most likely close or refresh the page at all.

  At this point, regardless of whether the user has closed the page or re-initiated the request, the message (old request) that has been delayed for a few seconds (or even longer) is now worthless. Not only that, the processing of these requests is to consume network, computing and storage resources in vain because the processing results are no longer interested by anyone.

* **The message should not be duplicated**: Consider the ACK mechanism mentioned above: after processing the message, the consumer needs to reply to the message queue service with the corresponding ACK signal to confirm that the message has been consumed.

  Still following the assumptions in the previous example, MQ delivers message 1 to node X, but does not receive the corresponding ACK signal from node X within the specified time. There are many possibilities at this point, for example:

  - Message 1 was not processed: Node X did not receive the message due to a network failure.

  - Message 1 was not processed: Node X received the message, but due to a power failure, it did not have time to save and process the message.

  - Message 1 has been processed: Node X received and processed the message, but due to

a power failure, it did not have time to return the corresponding ACK signal to the MQ service.

- Message 1 has been processed: Node X received and processed the message, but due to a network failure, the corresponding ACK signal cannot be transmitted to the MQ service.

  And many more. It can be seen that after the message delivery timeout, the MQ service cannot know whether the message has been consumed. To make matters worse, due to the reasons mentioned above (the user can't wait too long), this timeout is usually set as short as possible, which makes it more impossible for the MQ service to correctly presume the actual situation.

  At this time, in order to ensure that the message is not lost, MQ usually assumes that the message is not processed and re-dispatches the message (for example, after timeout, message 1 is re-dispatched to node Y). And this is bound to no longer guarantee that the message is not duplicated, and vice versa.

* **The message should not be unordered**: In the above example, the so-called "message is not out of order" means that the messages in MQ are to be consumed one by one in the order of "1, 2, 3, 4, 5". To ensure strict ordering, MQ is required to wait for a message to be processed (received ACK) before continuing to distribute the next message in the queue, which brings at least the following problems:

  - First of all, multiple messages in MQ cannot be consumed in parallel. For example, MQ cannot dispatch messages 1, 2, and 3 to nodes X, Y, and Z simultaneously, which leaves most consumer nodes in a hungry (idle) state. Even on the node where the message processing is being performed (such as node X), a large number of computing cores such as processor cores and SMT units are wasted.

  - Secondly, in the process of processing a message, all subsequent messages can only be in a waiting state. If a message fails to be delivered (timeout), it will block all subsequent messages for a long time during its "timeout - retransmission" period, causing that they cannot be processed in time.

    It can be seen that ensuring strict orderly message delivery will greatly affect the overall message processing performance of the system, increase hardware procurement and maintenance costs, and also significantly damage the user experience.

It can be seen from the above discussion that at this stage, there is no MQ product that provides reliable delivery of messages at reasonable cost. Under this premise, the current solution mainly relies on App Server's own business logic (such as: idempotent operation, persistent state machine, etc.) algorithm to overcome these problems.

Conversely, no matter how "reliable" MQ product is used, the current App business logic also needs to deal with and overcome the above-mentioned unreliable delivery of messages. Since MQ is not reliable in its essence, and the App has overcome these unreliability, why bother to reduce the performance by thousands or even tens of thousands of times to support the "distributed storage + ACK" mechanism at the MQ layer at all?

Based on the above ideas, BYDMQ does not provide so-called (actually unachievable) "reliability" guarantees like products such as RabbitMQ and RocketMQ. In contrast, BYDMQ adopts the “best effort delivery” mode to ensure that messages are delivered as reliably as possible without compromising performance.

As mentioned earlier, the App has overcome the occasional unreliability of messaging. Therefore, such a design choice greatly improves the performance of the whole system, and does not actually increase the development workload of the business logic.

Based on the above design concept, BYDMQ includes the following features:

* Like BYPSS, it is based on various high-quality cross-platform components in the BaiY application platform, such as the network server component capable of supporting tens of millions of concurrent connections on a single node; concurrent hash table containers that support multi-core linear extensions, and more. These high-quality, high-performance components make BYDMQ perform very well in terms of portability, scalability, high capacity, and high concurrent processing.

* Like BYPSS, it has a mature message bulk packaging mechanism on both the client and server side. Support automatic batch packaging of continuous messages, greatly improving network utilization and message throughput.

* Like the BYPSS, the pipelining mechanism is supported: the client can continuously send the next request without waiting for the response result of the current one, which significantly reduces the request processing delay, improves the network throughput, and effectively increases the network utilization.

* Each client (App Server) can register a dedicated MQ and keep it alive through a keep-alive connection with heartbeat mechanism. The corresponding owner broker (BYDMQ node) also pushes the incoming message to the client in real time through this keep-alive connection (with the bulk packaging mechanisms).

* The client predicts the owner of an MQ through the consistent hash algorithm. When the broker first receives a request for a specified MQ (e.g.: registration, message sending, etc.), it will be elected as the owner of the MQ through the BYPSS service. If the election is successful, the processing continues. Otherwise if the election fails, the client is redirected to the correct

owner node.

At the same time, BYDMQ will track cluster changes (e.g. existing Broker offline, new Broker launch, etc.) in real time through the BYPSS service and push these changes to each client node instantly. This ensures that unless the BYDMQ cluster is undergoing drastic changes (a large number of Broker nodes are online or offline), the accuracy of the owner prediction through the consistent hash algorithm will be very high, so that there is basically no need to take the redirection for most of the requests.

In addition, even if the consistent hash algorithm emits a wrong result, the actual owner of the MQ will be automatically learned by the client node to its local quick lookup table, ensuring that the next time a message is sent to the MQ, it can be directly delivered to the correct broker.

BYDMQ delivery messages directly from the client to the corresponding broker which is the owner of the specified MQ. This method avoids complicated routing and multiple relays of messages in the server cluster, thereby reducing its network routing to the simplest way, greatly improving the efficiency of message delivery, and effectively reducing the network load.

Elect the owner node of the MQ through BYPSS provides the cluster with a consistency guarantee of "every MQ is globally unique". At the same time, the BYPSS service is also responsible for dispatching some control-type commands (such as node online and offline notifications) between the various broker nodes, so that BYDMQ clusters can be better coordinated and scheduled.

* All messages to be delivered are only stored in the memory of the corresponding MQ owner node (the owner broker), avoiding a lot of useless overhead such as writing disk, replcation, consensus voting, and the like.

* The sender can set its lifetime (TTL) and the maximum number of error retries for each message. The resources it consumes can be precisely controlled based on the type and value of the message. Requests that are short-lived or not important can be invalidated in time to avoid continuing to consume resources at all points, and vice versa.

* Support for dispersed delivery: When the client node of the specified MQ is not online, or its connection is disconnected for more than the specified duration, all pending messages in this message queue will be randomly sent to any MQ that can still work normally.

The dispersed delivery solution expects that the clients (App Server) are also the BYPSS-based nano-SOA architecture cluster. In this case, if an App Server node is offline due to maintenance or hardware failure, all objects owned by the node will be released by the

corresponding BYPSS manager.

At this time, the system can randomly disperse the messages sent to this node to other nodes that are still working normally, and let the new reception node regain the ownership of the related object of the request through RegPort, and take over the original owner node that has been offline to continue processing. This greatly reduces the associated request failure rate when an application server node being offline, and optimizing the customer experience. At the same time, the random dispersion also makes the objects owned by the offline node evenly divided by other nodes still working in the cluster, which ensures the load balancing of the whole cluster.

In summary, BYDMQ sacrifices the reliability of the message that cannot be guaranteed by a certain degree, combined with message packing, pipelining, and direct delivery by the owner, greatly improves the single point performance of the message queue service. At the same time, thanks to the strong consistent, high-availability, and high-performance distributed cluster computing ability introduced by BYPSS, it also has excellent linear scale-out capability. In addition, its flexible control of each message, as well as the characteristics of dispersion delivery, ultimately provide users with an ultra-high performance, high-quality distributed message queue product.

## 5.5 Distributed Full Text Search (FTS) Service

The Distributed Full Text Search Service (BYDFTS) is based on the famous Sphinx full-text search engine and the BYPSS distributed coordination service. Provides a complete, consistent, high-availability and high-performance distributed clustering solution for the full-text search and tag matching functions. The main implementation details can refer to 5.4.3 Port Switch Service and other relevant sections, not repeat them here.

In addition, BYDFTS also implemented the following common functions through UDF plug-ins and other forms:

* Distributed cross-table (cross-index) association (JOIN) query: Added a highly efficient distributed cross-index JOIN support to the Sphinx engine. Which makes up the defects that all the mainstream full-text search engines (Solr, Elasticsearch, Xapian, Sphinx...) are currently cannot support the distributed cross-table JOIN query.

  This extension supports efficient LRU local caching and can access backend data storage (e.g. SQLite, MySQL, MongoDB, Cassandra, etc.) via the standard DBC plug-in (see: 5.4 nSOA - libapidbc).

  This extension supports the following three distributed data processing schemes:

- Shared instances: All distributed BYDFTS nodes share the same back-end storage services (e.g. MySQL, MS SQL Server, PostgreSQL, MongoDB, Cassandra, etc.). This architecture is simple and clear, and the high availability and horizontal scalability of the back-end storage service can also be achieved by the deployment of NewSQL / NoSQL cluster. Its downside is the need for additional deployment of back-end storage service clusters, increasing the burden of implementation and maintenance.

- Mirror Copy: All distributed BYPDFTS nodes mirror each other with the same data, usually used with local back-end storage such as SQLite. This scheme is suitable for queries when the associative auxiliary table size is small.

- Data Shard: Shard data by the consistent hash algorithm. The data access operations are mapped to the corresponding owner node according the hash value. This scheme supports replication groups that duplicate each data slice into the corresponding node group to provide high reliability and high availability.

* The Favor Rank algorithm for the Tag Collection (MVA attribute): This algorithm can compare two sets of Tags and return their similarity according to the user specified complex weighted matching rules. For example, each document label in the result set can be weightily matched with the current user preference tag and the similarity is counted in the ranking algorithm.

* Timeness Rank algorithm for time and date fields: This algorithm can rate fields such as timestamps in the specified rules and return their timeliness coefficients. The timeness factor can also be used as a factor in the ranking algorithm, which affects the sort order of the search results.

# 5.6 Secure Tunnel Service (BYST)

The BaiY Secure Tunnel Service (BYST, pronounced "best") provides users with an end-to-end secure tunnel service that supports the following features:

1. Support PSK and PKI authentication: authentication can be performed using pre-shared key or the public key algorithms.

2. Support dozens of strong encryption algorithms: support dozens of block encryption and streaming encryption algorithms. For a list, please refer to 4.1 The Cryptographic Algorithm Module - algorithm.

3. Seamlessly integrates with our self-developed EAL5+ level smart card hardware to provide a very high security level of protection.

4. Support for on-the-fly data compression: High-performance real-time data compression based on lz4 algorithm, which can be enabled through configuration options.

5. Support message integrity check: Use the high performance hash algorithm to calculate the checksum to ensure the reliability and integrity of the message (Both the checksum and the compressed data are encrypted and then transmitted). This feature can be turned on with configuration options.

6. Support anti-replay attack: the time window range of anti-replay attacks can be enabled and set through configuration items.

7. Data obfuscation: Unlike existing solutions such as OpenVPN, SSL, IPSec, and SSH, this tunneling protocol has no features to be observed. A third-party interceptor who does not know the key can only see a random binary byte stream, and it is difficult to detect that the communication party is using the tunneling protocol by any effective means.

   Not only does it have no features, BYST can also act as a legal whitelisting protocol such as http (BYST over http), ensuring that it works in extreme environments that block only based on traffic restrictions even for unknown protocols.

   At the same time, BYST also supports HTTP chunked transfer encoding. This only adds an additional 3.5 Bytes overhead for each package (tens of KB to several MB). The data expansion rate is less than 0.01%, which avoids the problem of data bloat caused by the cumbersome HTTP header. Compared to other whitelisting protocols such as SSL / TLS, SSH, h2 and WebSocket, the additional communication overhead of this solution is significantly reduced. Therefore, it maintains a very high network utilization while achieving the whitelist communication.

8. High efficiency: Thanks to the asynchronous IO components based on assembly optimization, zero memory copy, and DMA + hardware interrupts (see: 3.2.1 High performance I/O Framework), even on a limited hardware platform, BYST can meet the demanding requirements of high performance and high concurrent services.

   In addition, BYST significantly improves network utilization (high payload ratio) and network throughput due to our advanced bulk packaged IO, patented distributed N:M dynamic connection pool acceleration algorithm, and the high performance real-time data compression algorithms.

   At the same time, compared with various existing solutions, the BYST tunneling protocol, which has been carefully simplified in terms of handshake (negotiation) and acknowledgement, also resulting in significantly lower communication delays.

9. High performance and high availability cluster: Supports high performance and high availability multi-active IDC cluster deployment based on BYPSS (see: 5.4.3 Port Switch Service (BYPSS)).

   In addition, BYST also supports multi-path automatic inspection and automatic routing switching. The overall availability of the service can be significantly improved by adding alternate links, while it also supports features such as automatic traffic aggregation and load balancing.

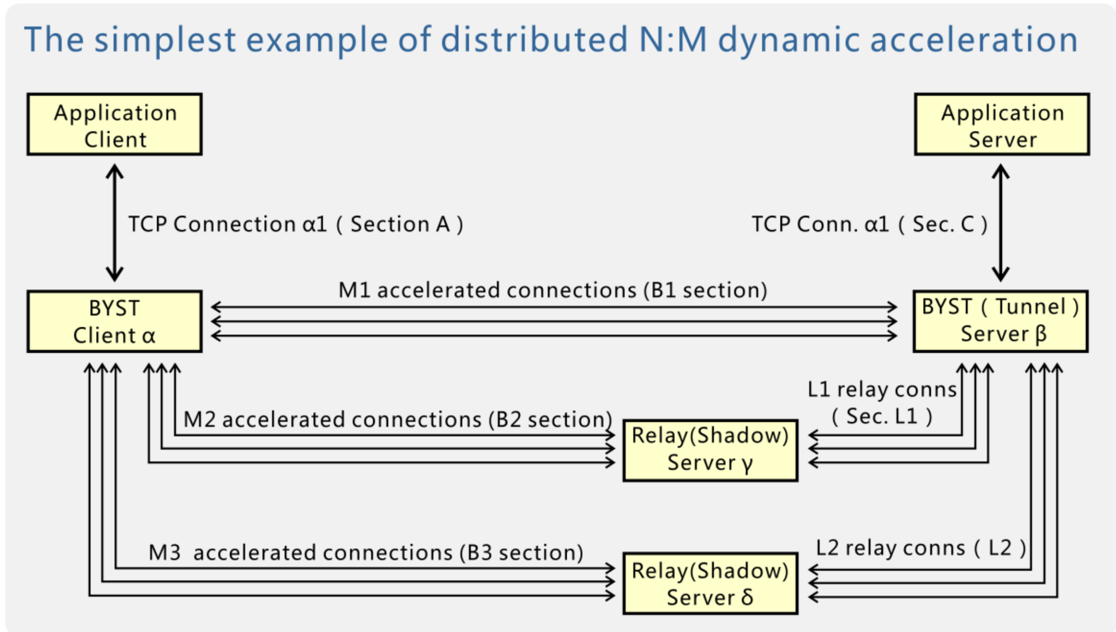
10. Line-level real-time compensation: Unique line-level real-time tracking and predictive intelligent algorithms that track and compensate for packet loss and jitter of each line in real-time.

BYST service is mainly used to establish secure and reliable data transmission channels in wide-area or metropolitan area network environments such as Internet, satellite, microwave, SDH (MSTP), and inter-area fiber-optic lines. The strong encryption and authentication algorithms ensure data security, reliability and integrity, and the bandwidth costs can be reduced through data compression. At the same time, it is also possible to protect its tunnel communication from being intercepted, recognized and blocked by an obfuscation algorithm that is difficult to analyze.

In summary, BYST mainly brings the following technical advantages:

1. Ensure communication security: Provide a secure communication tunnel with strong encryption, strong verification and anti-replay attack mechanisms.

2. Full bandwidth: Thanks to the unique IO automatic batch packaging, real-time data compression, and our patented distributed N:M:N dynamic connection mapping acceleration algorithm, BYST can fully fill the user bandwidth limit, significantly improving the site -to-Site Tunnel communication performance. Actual measurement by a large number of users shows that in the intra-city communication (MAN) environment, with only a single point of N:M acceleration enabled, BYST can increase the bandwidth throughput by more than 6 times; In a remote communication (WAN) environment, BYST's single-point N:M acceleration can achieve a throughput performance improvement of up to 70 times. On this basis, distributed N:M acceleration can linearly increase the aggregate throughput rate with the increase of distributed acceleration nodes.




3. Prevent false blocking: As mentioned above, in order to improve communication performance and reduce handshake delay, BYST itself is designed as a completely non-characteristic protocol. In addition, through data obfuscation mechanisms such as strong data encryption, IO automatic batch packaging, real-time data compression, and our patented distributed N:M:N dynamic connection mapping acceleration algorithm, all upper-layer protocols carried by BYST will also lose their recognizable characteristics. In addition, the whitelist protocol communication support of BYST over HTTP further guarantees its excellent firewall passability.

4. Authentication agent: BYST can provide safe and reliable dual-end authentication access through PSK, PKI and EAL5+ level secure smart card hardware technology. This eliminates the need to implement complex algorithms such as CHAP, IKE, and LDAP to complete authentication when interconnecting upper-layer applications.

# 6. Interface, Media and Other Tools

This includes a cross-platform audio I/O framework (libaudioio), a cross-platform I18N GUI framework (libmlgui), and etc.

## 6.1 Cross-platform Audio I/O library - libaudioio

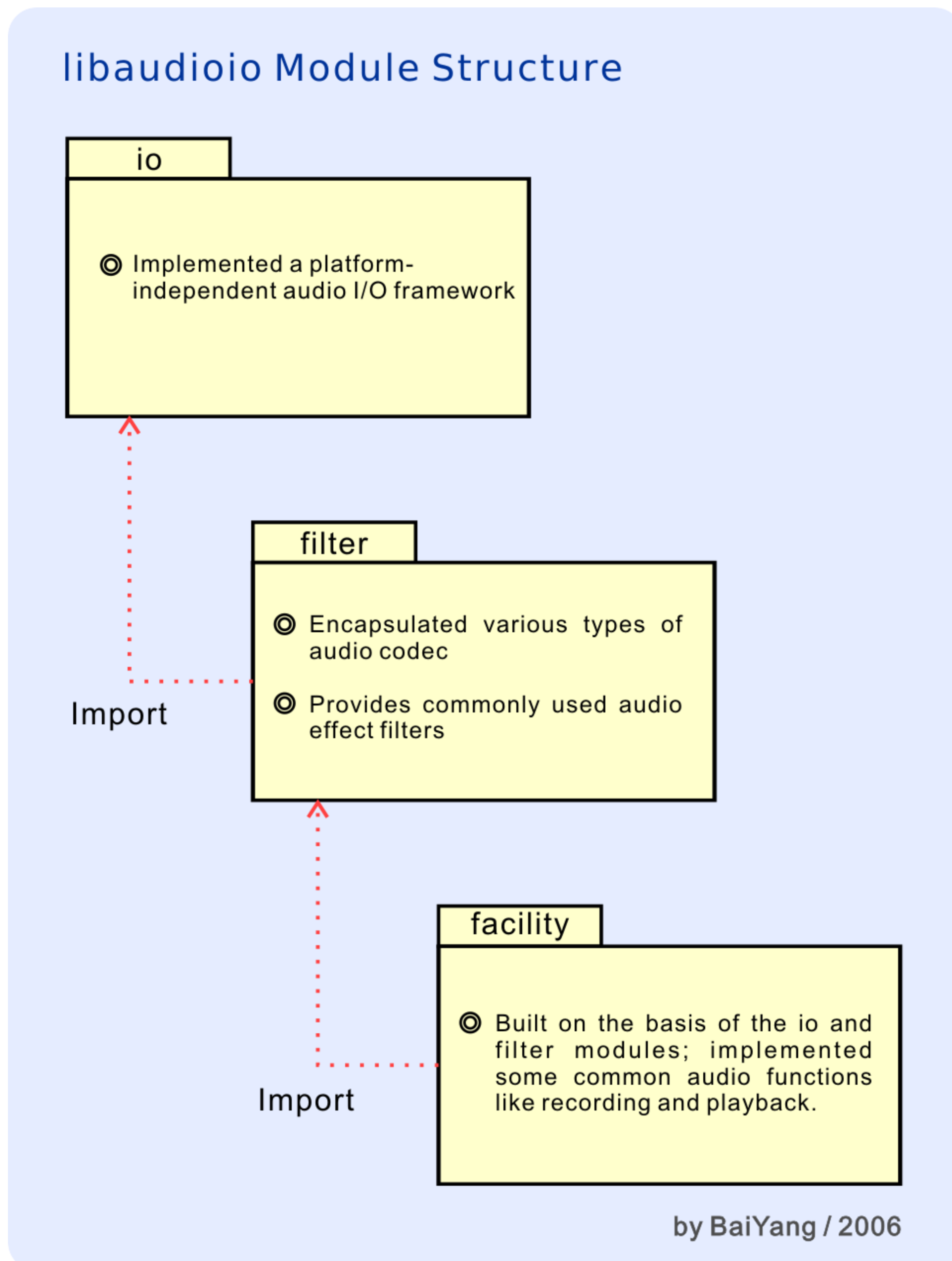



Figure 1

As illustrated in Figure 1, the libaudioio library employs a similar module structure with both libutilitis and libcrypto. The I/O module at the very bottom is responsible for encapsulating all audio I/O interface operations associated with OS, and providing a unified audio I/O interface to the upper level.

The I/O module currently supports the following audio interfaces:

- Audio Session API (Win3264)
- WDM Kernel Stream (Win32)
- DirectSound (Win3264)
- Multi Media Extension (Win3264)
- Core Audio (Macintosh)
- Sound Manager (traditional Macintosh)
- Advanced Linux Sound Architecture (Linux)
- Audio Science HPI (Linux)
- Audio Library (un*x)
- Open Sound System (un*x)
- ASIO (platform independent)
- JACK (platform independent)

Provides the upper level with a unified interface used for asynchronous communications based on the command pattern.

The filter module is established on the basis of the I/O module. It conforms to the data filter framework defined in libutilitis, and supports using various filters to freely establish a filter chain between data source and data sink. The filter module has two sub-layers: codec layer and effects layer.

Among all the objects defined in the codec layer, only data source (decode) and data sink (encode) are included. It supports encoding and decoding of various formats, such as wav (including x-law, g.7xx, gsm and other sub formats), caf, au, snd, voc, mpx (mp1, mp2 and mp3), mpc (Musepack), flac, and ogg vorbis. For mpx and mpc, only decoding is supported considering legal issues.

It is worth noting the importance of au format. As a traditional audio format for Unix, au has a large user base and is the only non-compression audio format that supports labelling sample length information at the end of the file (just set the corresponding field in the file header to -1). This is a very important characteristic for generating a temporary file and for stream I/O.

We have defined various audio effect filters in the effects layer. Because currently most products don’t have high audio processing requirements, we have implemented only the basic filters like attenuator (volume control) and sample rate convertor.

The module located at the top of the libaudioio library is named as facility, the same as libutilitis and libcrypto. It encapsulates the high level general functions associated with audio processing. We have defined only a set of audio recording and audio playing tools, considering that currently most software products do not have high audio processing requirements.

## 6.2 Cross-platform I18N GUI Framework - libmlgui

Unlike traditional system functions and various algorithms, the GUI’s implementation methods for platforms differ from each other in a variety of ways (mechanism and model). The implementation details are very complex. For common software development organizations, it requires huge efforts to establish a complete and robust cross-platform GUI framework from scratch.

The libmlgui library has implemented a universal GUI framework with I18N support and various functions, by utilizing the famous open source GUI framework wxWidgets (has over 24 years history) and by co-working with libraries like libutilitis.

The libmlgui framework is implemented in four-layer architecture:

## libmlgui Module Structure

**FS extension**

◎ Extends wxFileSystem capabilities.

Import

**I18N components**

◎ Implemented a graphic controls library with I18N support, by utilizing related functions provided by libutilitis as well as the standard controls within wxWidgets.

Import

**Quick help framework**

◎ Offers a suite of universal GUI framework with a four-level help system.

Import

**Universal graphic controls**

◎ Implemented universal graphic controls such as the configurations editor.

by BaiYang / 2006

Figure 2

### 6.2.1 File System Extension

In this module, VFS and other mechanisms defined in libutilitis are extended to be the driver layer within wxWidgets and wxFileSystem:

- Memory-based file system: Use the virtual registry (CConfig) component provided by libutilitis to implement a memory-based virtual file system that supports directory structure and zero-copy access.
- VFS-based file system: Employ the VFS framework defined in libutilitis to enable wxFileSystem to support virtualizing one or more files into a single read-only file system. Because the VFS (in libcrypto) that supports encryption and compression functions is also implemented based on the VFS framework defined by libutilitis, this extension has simultaneously added the support for such kind of VFS to wxFileSystem.

  It is worth noting that VFS systems are categorized into two types: file based (virtualizes a certain type of file into a file system) and encapsulation based (encapsulates real file systems like FTP into objects that can satisfy VFS user interface). So this extension has simultaneously added the capability for accessing VFS (based on FTP or HTTP) to wxFileSystem.

### 6.2.2 I18N Components Library

It is easy to implement a graphic controls library with multi-language support by utilizing the language pack (uses hash index table) and the multi-language object class, as well as the various GUI controls (e.g., buttons, menus, windows) within wxWidgets.

This section takes buttons (the most commonly used GUI control) for example, to explain how the multi-language GUI controls are implemented.

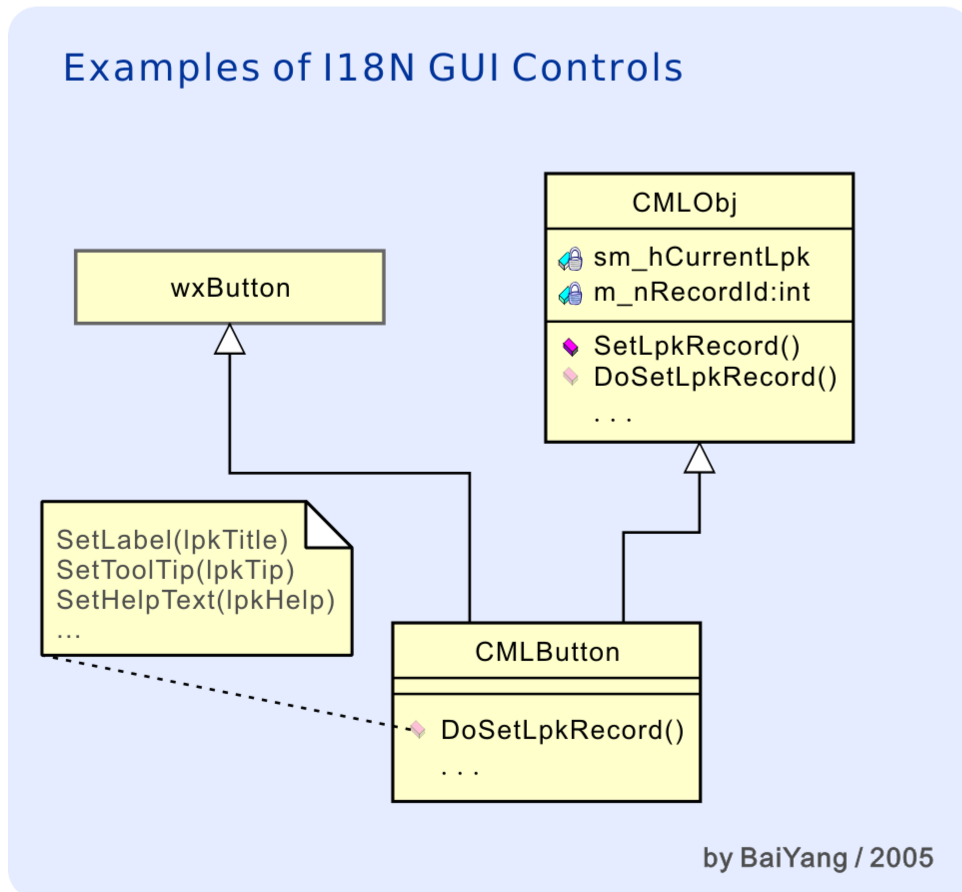



Figure 3

Figure 3 illustrates how the multi-language controls in liblmgui look like. In libutilitis, the common base class CMLObj (for all multi-language UI controls) is defined along with the language pack. This CMLObj class retains a handle of the language pack in the form of static member (the handle is a smart pointer that supports reference counting and customized destroy strategy). Language resource ID used by the current object is kept by a non-static integer member.

A multi-language graphic button control is derived from the standard button class (wxButton) and the CMLObj class. We can set appropriate language, font and other information while each graphic object is being initialized, by correctly overloading the virtual method DoSetLpkRecord.

This library also utilizes charset encoding conversion functionality provided by libutilitis and the font mapping mechanism provided by wxWidgets, to present these language resources to users as correctly as possible.

One thing to note is that we have intentionally ignored some necessary functions (e.g., the Refresh mechanism required for changing language pack in real time), for the sake of easier description. For complex controls like list and combo box, the language pack can also resolve them through corresponding mechanisms (e.g., each record can correspond to multiple values simultaneously).

Moreover, this module supports loading language pack from various wxFileSystem mentioned above.

### 6.2.3 Quick Help Framework

Software designers have noticed that users always have limitless requirements for applications. In addition to a rich set of functionalities, users also expect higher usability from the software. To address this challenge, there is a need to implement a suite of universal quick help framework, in addition to designing a user interface that is easy to understand, highly consistent and logic.

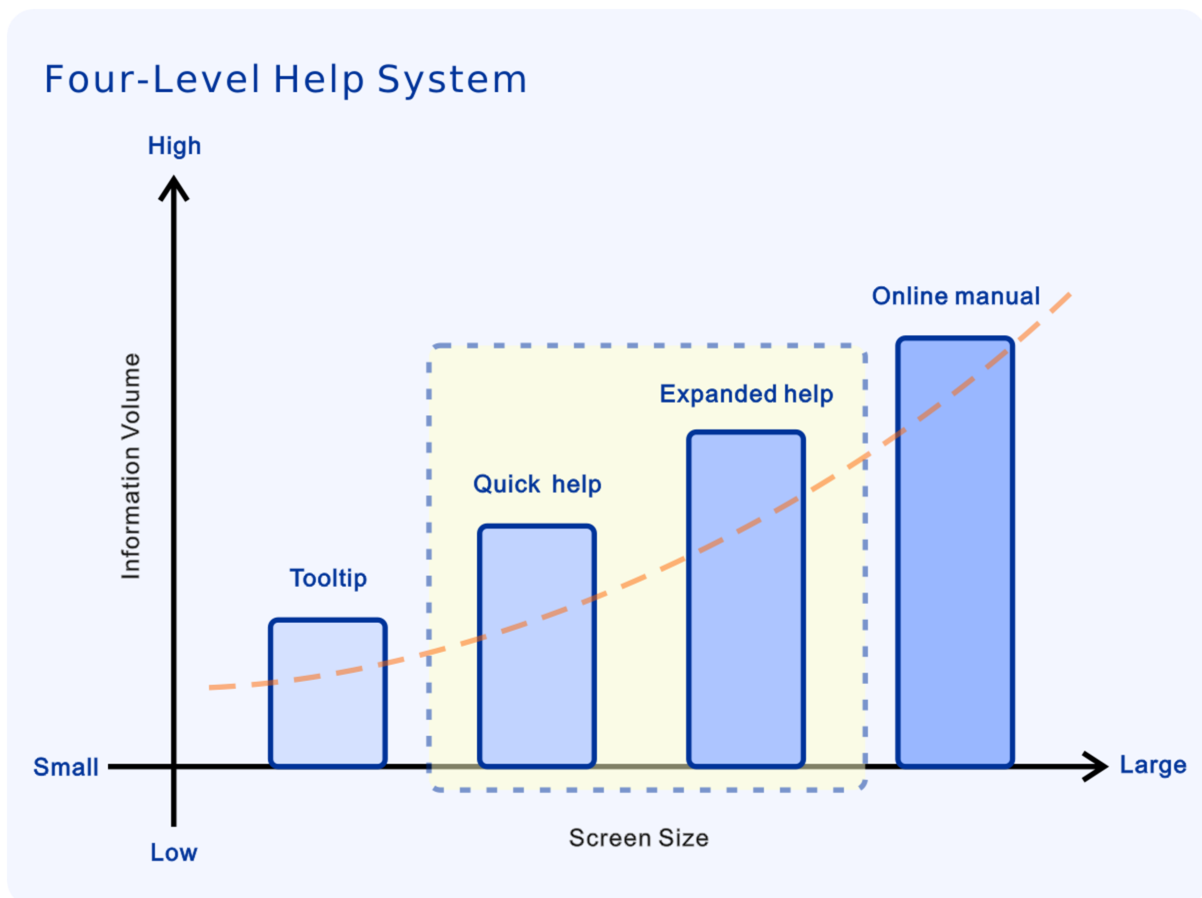



Figure 4

As shown in Figure 4, in an application with quick help system enabled, help information is expanded to four types. In this four-level help system, the content of help will become more complete and detailed as the help level increases. At the same time, the help system will require more screen space and user attention as the help level increases.

In a traditional two-level help system, “tooltip” has advantages in requiring smaller screen space and less user attention, though the information it can provide is also limited and cannot be steadily

shown on the screen for users to read deeply. On the contrary, online manual has elaborate content and explicit structure, but most users are uninterested in reading online help of a software product due to its disadvantages (requiring more screen space and user attention).

The four-level help system provides a good balance between the above two aspects, by adding “Quick Help” and “Expanded Help” mechanisms.

## Quick Help

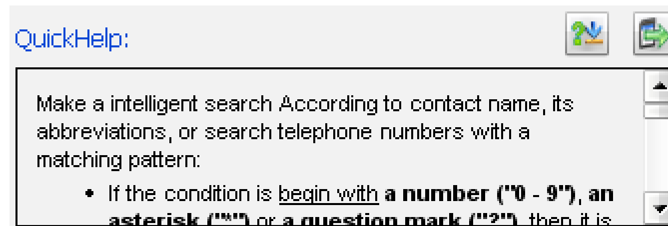
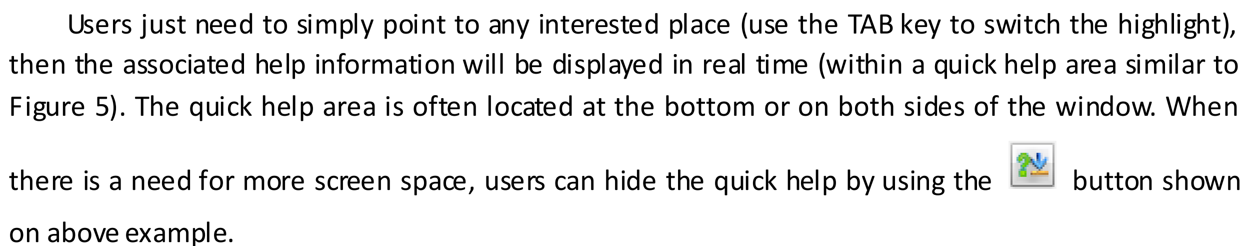



Figure 5

Users just need to simply point to any interested place (use the TAB key to switch the highlight), then the associated help information will be displayed in real time (within a quick help area similar to Figure 5). The quick help area is often located at the bottom or on both sides of the window. When there is a need for more screen space, users can hide the quick help by using the  button shown on above example.

To improve expression capability, the quick help area supports standard HTML texts, and can display commonly used graphic formats like ico, bmp, png, gif, jpg, pcx, and xpm. Furthermore, the behaviour that happens after clicking a hyperlink is customizable by software designers (e.g., either to open a specific page within the online help or to load an application).

To keep portability and to save resources like CPU and memory, the HTML window within the quick help utilizes the controls within wxWidgets and functionalities offered by libutilitis, instead of using a third-party WebView component like Internet Explorer.

Extended Help

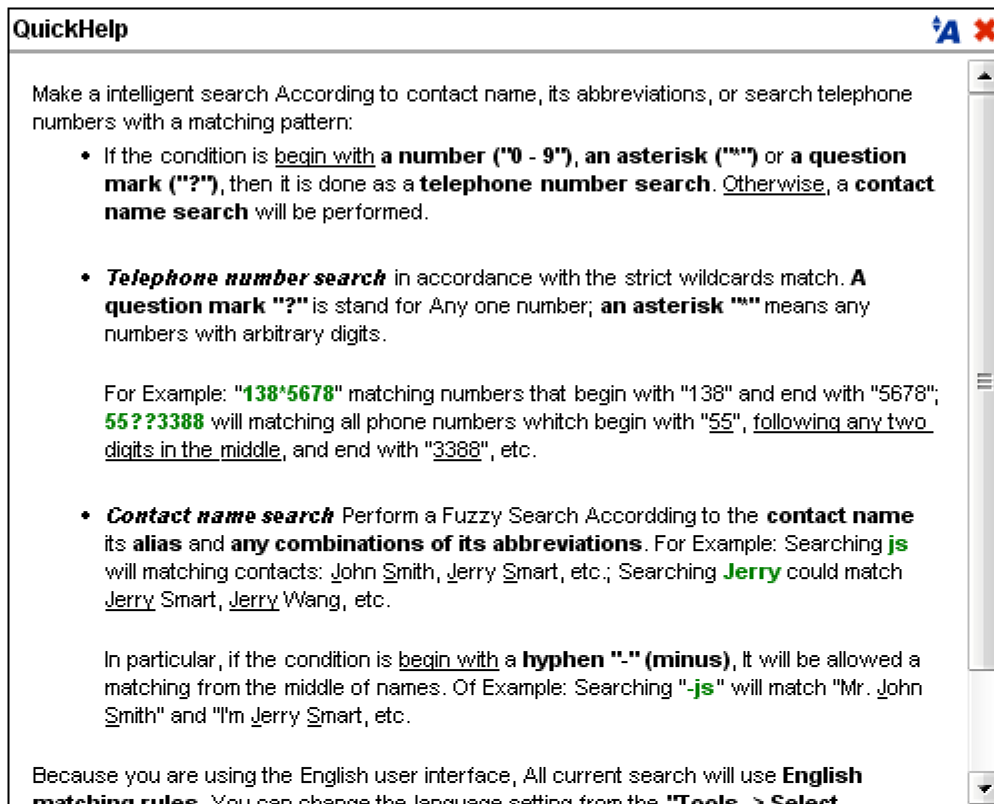
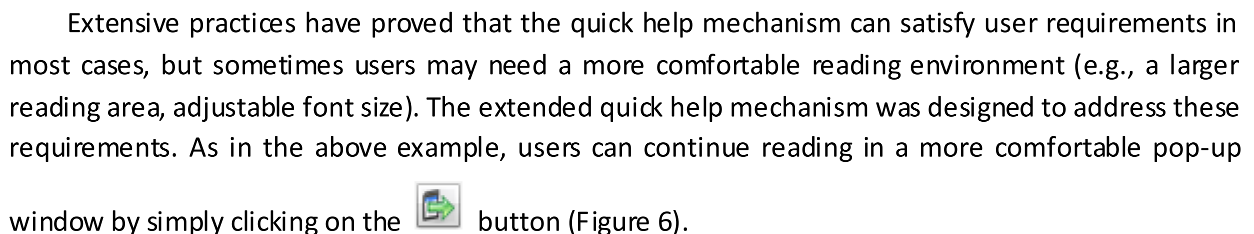



Figure 6

Extensive practices have proved that the quick help mechanism can satisfy user requirements in most cases, but sometimes users may need a more comfortable reading environment (e.g., a larger reading area, adjustable font size). The extended quick help mechanism was designed to address these requirements. As in the above example, users can continue reading in a more comfortable pop-up window by simply clicking on the  button (Figure 6).

The quick help framework can automatically acquire correct help information from controls and language packs and display them, and simultaneously supports loading specified HTML page or bitmap directly from wxFileSystem.

## 6.2.4 Universal Graphic Controls

We have defined some commonly used graphic controls here, including configurations editor, busy-waiting dialog box, and etc. Here we introduce the configurations editor CConfig in more details.

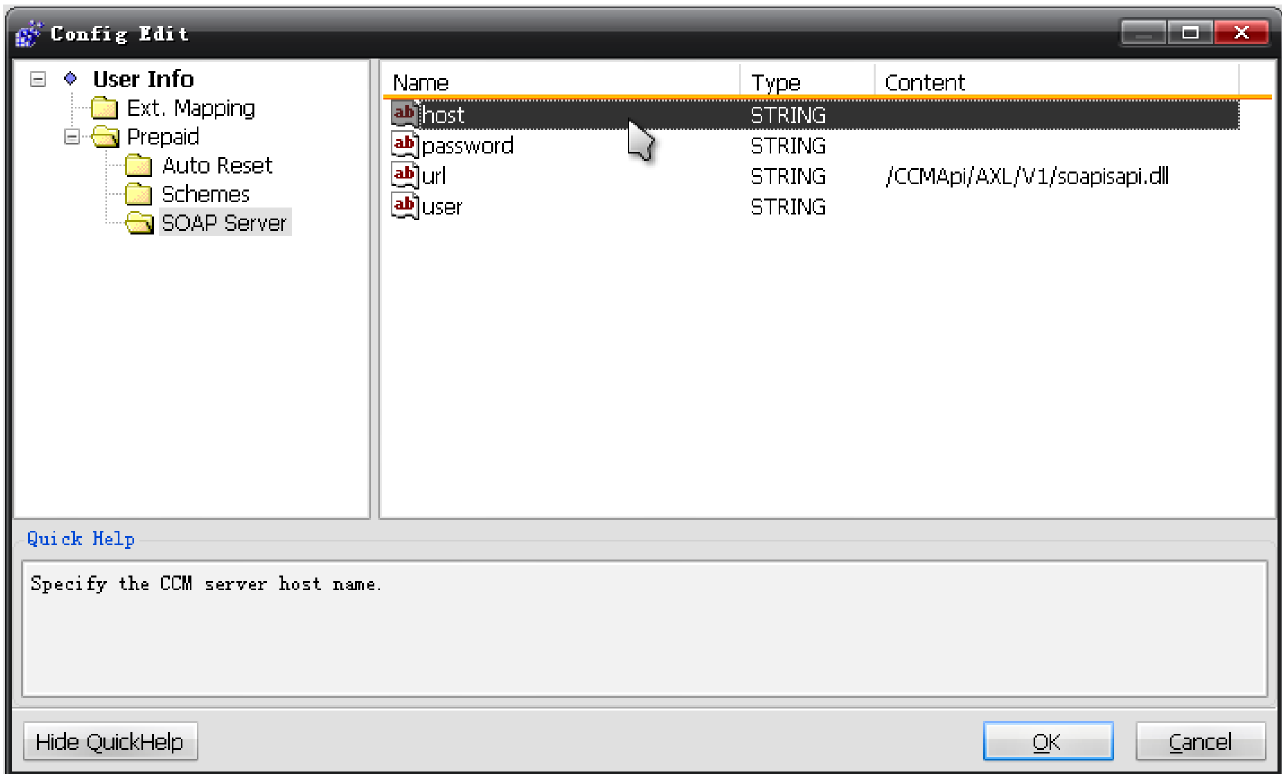



Figure 7

As shown in Figure 7, this control has its own quick help area, which can provide users with instant help information such as usage instructions about the current option or sub-key.

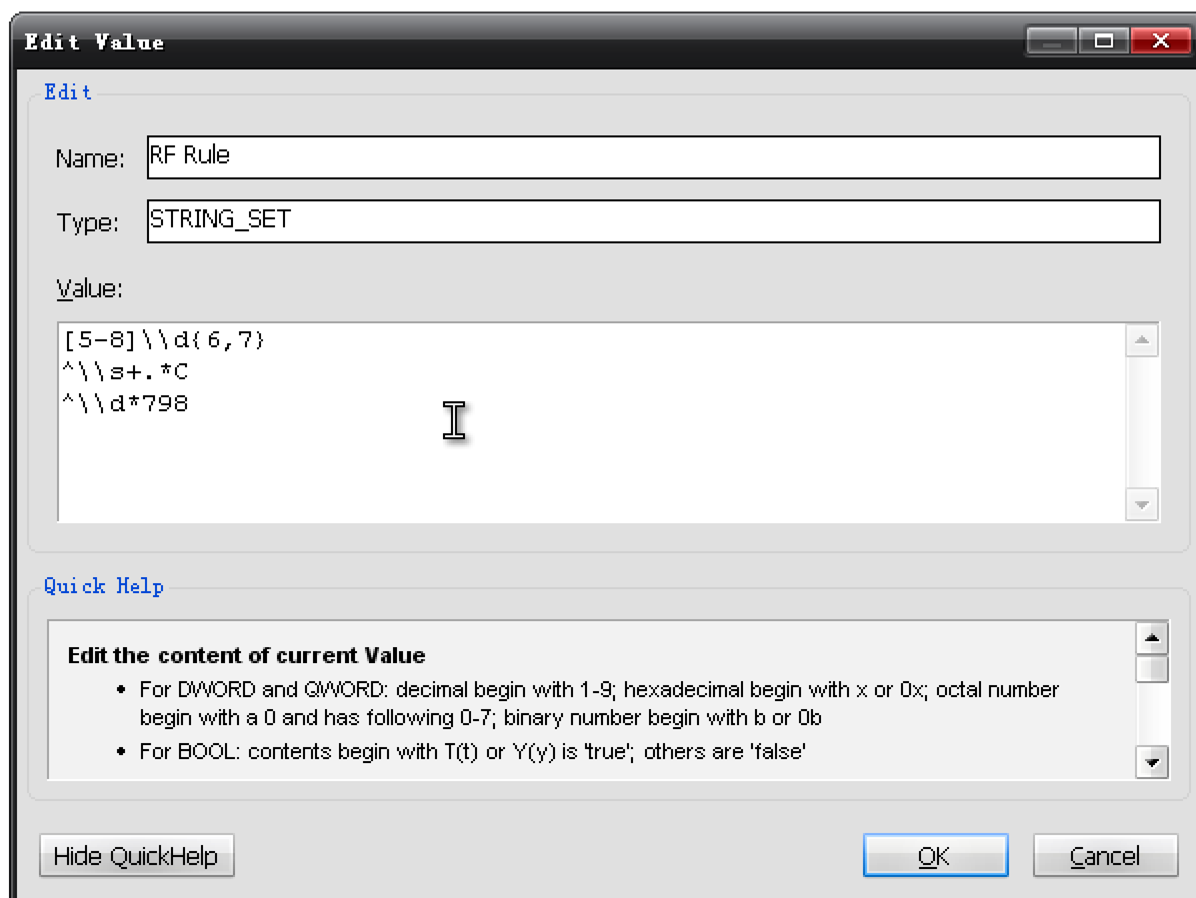



Figure 8

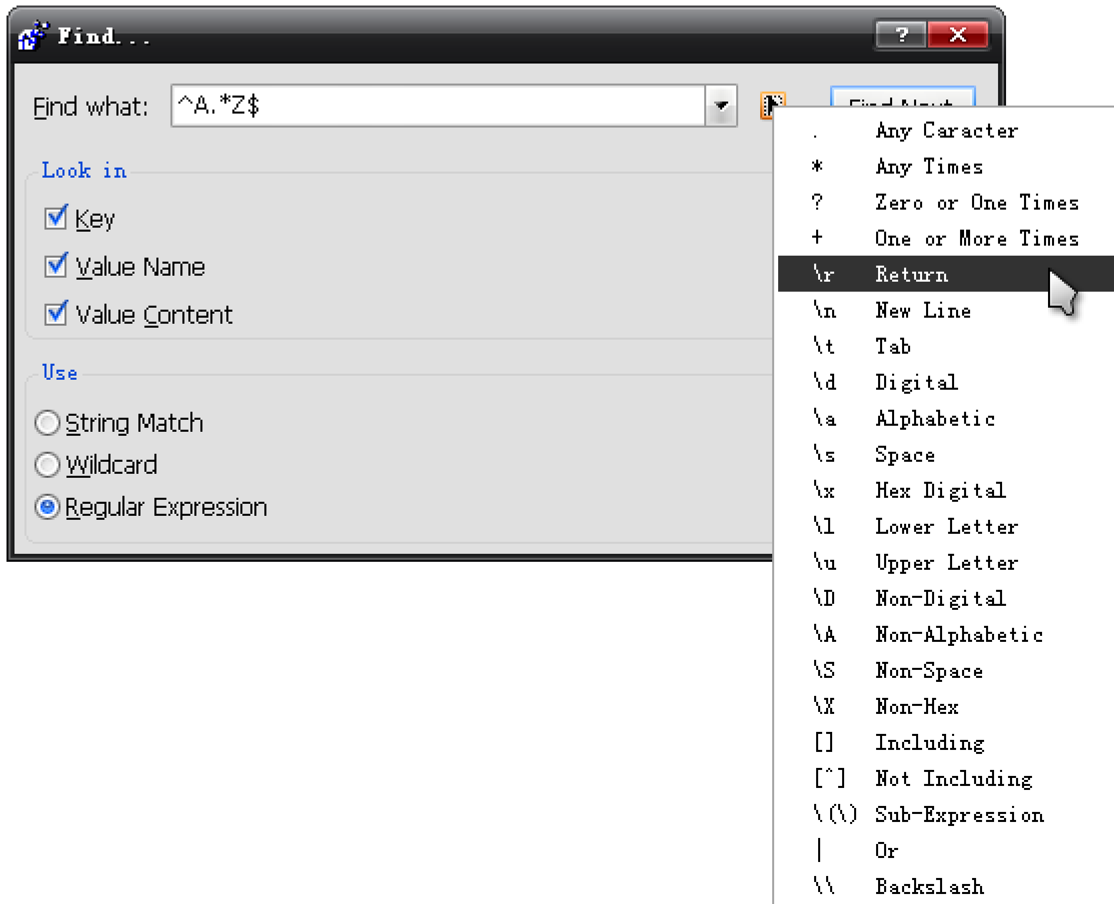



Figure 9

Figure 8 and Figure 9 show the value editing and search functions provided by this control.

The following questions depend on what kind of virtual registry is passed to the editor by the software designer or the user: what sub-key structure will appear in the configurations editor control; what specific values are contained in the subkeys; and what quick help information correspond to these subkeys.

In the descriptions about libutilitis, we mentioned a component that can simulate registry service in non-Windows platforms. This cross-platform tool can provide better performance than Windows registry, and can be stored in almost any place (e.g., VFS, memory, network media, or embedded into executable modules). So the configurations editor control described here is intrinsically a virtual registry (CConfig) editor. It can be used for browsing and editing any tree view structure and configurations within specified virtual registry object. Also, it can display the quick help information corresponding to the current configuration instantly. The flexibility of the design enables the configurations editor control to adapt to various complex applications.

In addition to the Advanced Configurations dialog box, the universal control module should also contain other universal controls like busy-waiting and language-selection dialog boxes, which have very

simple meaning and will not be described here.

## 6.3 CConfig Language Binding Component

The CConfig within libutilitis is a tree data structure used for storing complex information. It supports a wide variety of data types. Cross-platform, high efficiency, internalization, and flexible interaction are some of its characteristics.

CConfig has been widely used in configuration management and in situations when there is a need for passing information among API, WebAPI and module interfaces. For the purpose of facilitating second development by users or by any third-party, we have implemented language binding which is specifically designed for mainstream languages and environments, including C/C++, Java, C#, F#, VB.NET, PHP, and JavaScript.

The CConfig Language Binding component offers the following functions in object-oriented style:

- Save and Open: open and save CConfig data in JSON, CSV, XML, INI, and binary (ISXF) formats. (JavaScript version supports CSV, INI and JSON formats only)
- Value access: read, write, replace, create and delete operations with values; supports type identification, existence checks, and other management options.
- Sub-key access: create, delete, detach, import, export, replace, and insert operations with sub-keys; supports existence checks and other management options.
- Search and traversal: traversal of subkeys and their values; search based on wildcard and regular expressions.
- Status information: whether the current object is null or has been changed; to acquire the number of sub-keys and values; equality comparisons and etc.
- Multi-thread safety: all operations on the CConfig component are thread safe; supports exclusive access by explicit locking (Lock/TryLock/Unlock) with time-out timer (except for JavaScript version which does not require these); CConfig (except for JavaScript version) employs a spinlock mechanism with limited spin times to improve performance.

For more information on CConfig Language Binding, see the CConfig Reference Guide.

## 6.4 JavaScript Tools Library - libbaiy

The libbaiy library was designed with the purpose of facilitating the communications and

interactions among browser-end codes and BaiY platform based server-end codes in applications with B/S architecture. It comprises two parts: function library and interface library.

### 6.4.1 Functions Library

The function library focuses on functional operations associated with calculation and data encoding/decoding, including:

- Cryptographic algorithms, such as SHA1 hash algorithm, HMAC algorithm, and algorithms for encoding and decoding between HEX and BLOB.
- Tools for parsing and generating CSV data that are fully compliant with RFC 4180; customizable separator, line break, and quote character; support quote character escape within a quoted field.
- The CConfig component used for accessing CConfig Schema in a convenient and efficient way (see 6.3 CConfig Language Binding Component for more details). CConfig can access sub-keys and values at an efficiency of O(1) or O(log(N)), which depends on how the JavaScript engine has implemented associated containers.
- The language pack component can load the language packs that are defined and implemented within libutilitis. In addition to access every record defined in the language pack, it also can access language name, compatible code page and ISO standard name, charset and charset encoding, and others meta information like original encoding. This component can access all the above fields at an efficiency of O(1) or O(log(N)), which depends on how the JavaScript engine has implemented associated containers.
- I18N string comparison and sorting algorithms supporting customized sorting rules (e.g., Chinese -> Hanyu Pinyin, Chinese -> Taiwan Pinyin, Japanese -> Rome Pinyin); table (two-dimensional array) sorting algorithms supporting composite multi-column ascending or descending order; I18N string processing tool such as rule compiler which supports strict matching, wildcard matching and regular expression matching.
- Keyword tree container. The keyword tree is a common “key -> value” data structure designed to implement efficient prefix matching. The JavaScript version interface implemented within libbaiy is compatible with the C++ version within libutilitis. However, its function is only a minimal subset of the C++ version.
- Message dispatching component. The message dispatcher (API Nexus) can dispatch different types of messages to their corresponding processor. If the processor of specified type is not registered yet, the dispatcher will store messages in a temporary message queue, and will dispatch the messages in FIFO order (to solve inter-module dependency) once the message processor is registered.

* Task queue component designed for managing and executing a series of specified tasks in order. The order for executing these tasks can be pre-defined, and is adjustable while tasks are being executed. After all tasks are executed, or when unexpected exceptions occur, the finally callback will be triggered. This component is mainly used for solving callback management issues (callback hell) caused by asynchronous mode in JavaScript environment.
* Mobile platform tools designed for mobile devices such as iPad, iPhone and Android devices. For example, a tool for determining mobile platform runtime, and the drag and drop adapter for simulating Touch events as mouse events.
* Other miscellaneous functions, such as the XMLHttpRequest (XHR) object factory that is independent of web browser; dynamic load (synchronous or asynchronous) and cross-site load of JavaScript; dynamic load and unload of CSS style sheet; set or change background graphic for specified DOM object in stretch style; test current browser platform; test long-polling compatibility of the current browser; cross-browser mouse event Hook; cross-iframe message passing; byte swap operation with 16-bit and 32-bit integer; http header parser, and etc.

## 6.4.2 User Interface Library

It offers ExtJS-based UI controls and associated framework. All controls support I18N and themes.

This UI library has implemented the following features:

* Extended quick help framework. A simplified browser-end 6.2.3 Quick Help Framework.
* A simplified browser-end configurations editor. See the following sections for more details.
* Miscellaneous functions such as: a patch for adding L10N sorting capability to ExtJS table; the load mask control with shadow effect and supporting delayed popup and hide.

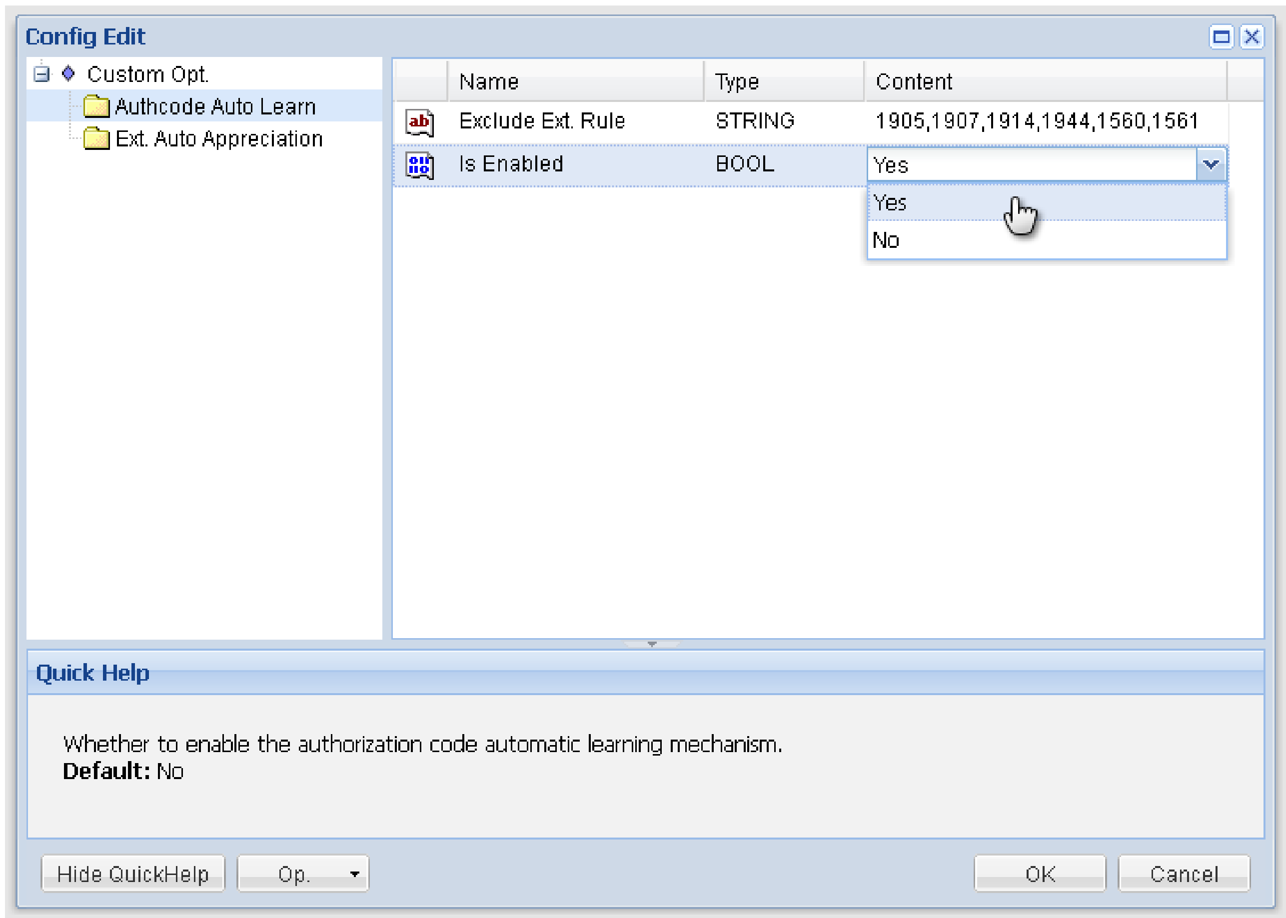



Figure 10

Figure 10 shows the CConfig configurations editor of JavaScript version. It has similar interface and functions with the universal configurations editor described in 6.2.4 Universal Graphic Controls.

# 7. Error Processing Mechanism

The application platform is at the bottom of the whole application, in which location things turn to be error-prone. On the other hand, processing a specific error is usually decided by upper-layer business logic.

The exceptions mechanism in C++ is ideal for use in this case for the following reasons: the use of exceptions eliminates the need to judge if each operation is successful by returned value and the public variable, as well as the pains and errors caused by such judgement; it also eliminates the embarrassment of returning from the error point to the position which allows the error to be processed step by step; the exceptions mechanism is helpful for implementing a more structured error processing method.

From performance perspective, the exceptions mechanism in C++ is very suitable for error processing for the following reasons:

1. The exceptions mechanism is enabled only when an error occurs. So it will not affect performance in normal conditions. On the contrary, enabling this mechanism may increase performance of the program, because it eliminates the need to judge returned value and/or to check public error variable for all operations one by one.

2. Even if an exception occurs, only exception catching and stack unwinding operations involve the O(N) algorithm which is closely associated with the current function call stack. However, the algorithm used for traditional mechanism (returning to the error processing position step by step) is comparable to O(N) in terms of complexity.

Based on the above reasons, we have defined a set of exception class structure for the application platform.

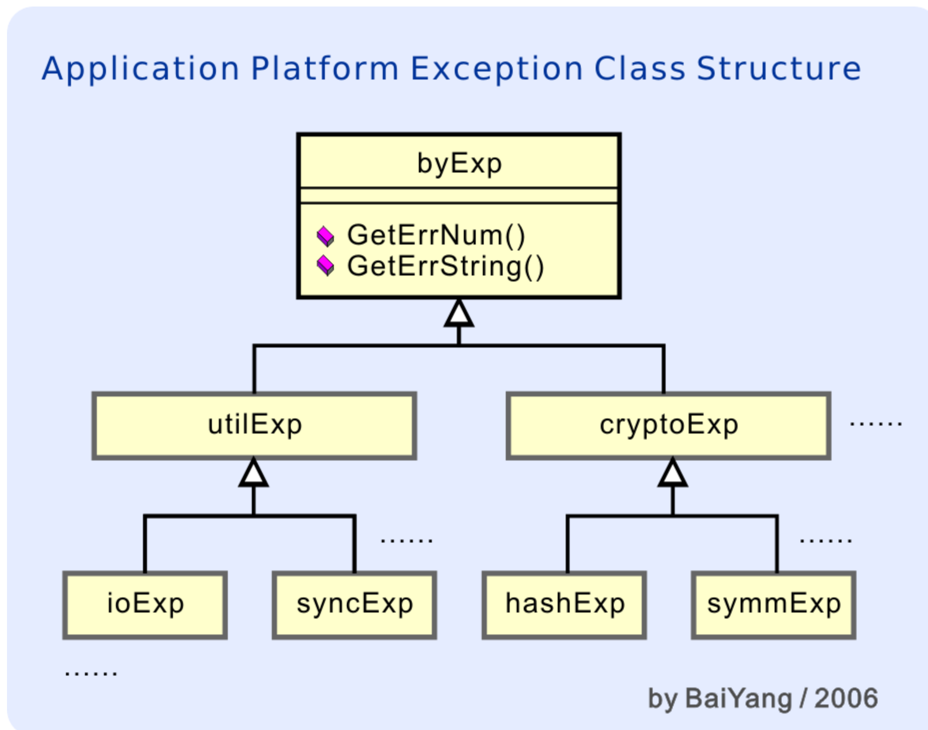



Figure 11

Figure 11 shows only part of the exception class structure, but it is enough to illustrate the design ideas of the exceptions processing mechanism. It is a simple four-level structure. Though the above figure shows only three among the four levels, we can imagine that types like "fileExp" and "socketExp" will be derived from "ioExp", and types like "mutexExp" and "semExp" will be derived from “syncExp”.

Each exception object carries three types of information: its location within the whole exception class structure (i.e., its type information); an error code; and detailed description about this error. Practically, a Reason Code will be added for some complex exception objects. Then the upper-layer exceptions processor can implement error processing strategy for different granularities accordingly (e.g., either to process all cryptography associated errors or to process hash verification errors only).

For more discussions on the usage of exceptions, see section Exceptions in my document C++ Coding Guidelines. For the details of how a compiler implement the C++ exception mechanism, and its performance analysis, please refer to "Inside C ++ exception" section (Chinese only).

Now we have finished a quick overview of the BaiY Application Platform. For more information, refer to the user manual and developer’s guide for each component.