nano-SOA - a powerful alternative and complementary of SOA

Bai Yang
baiyang@gmail.com
http://baiy.cn

Abstract

SOA (Service Oriented Architecture) and micro SOA (Micro Service) have the advantages of high cohesion and low coupling, but at the same time they also bring complicated implementation, maintenance, high network load, and weak consistency. At the same time, they also increase product development and operation costs. This article attempts to use an improved approach by a kind of plug-in isolation mechanism that avoids the above issues as much as possible while preserving the benefits of SOA.

In addition, this paper also proposes a new strong consistent distributed coordination algorithm for improving the low performance and high overhead (at least three network broadcasts and multiple disk I/O requests per request) problem of existing Paxos/Raft algorithms. This algorithm, at the expense of data reliability, completely eliminates the above overhead and provides the same level of strong agreement and high availability guarantees as the Paxos/Raft algorithm.

1. The nano-SOA Architecture

1.1 AIO vs. SOA

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.

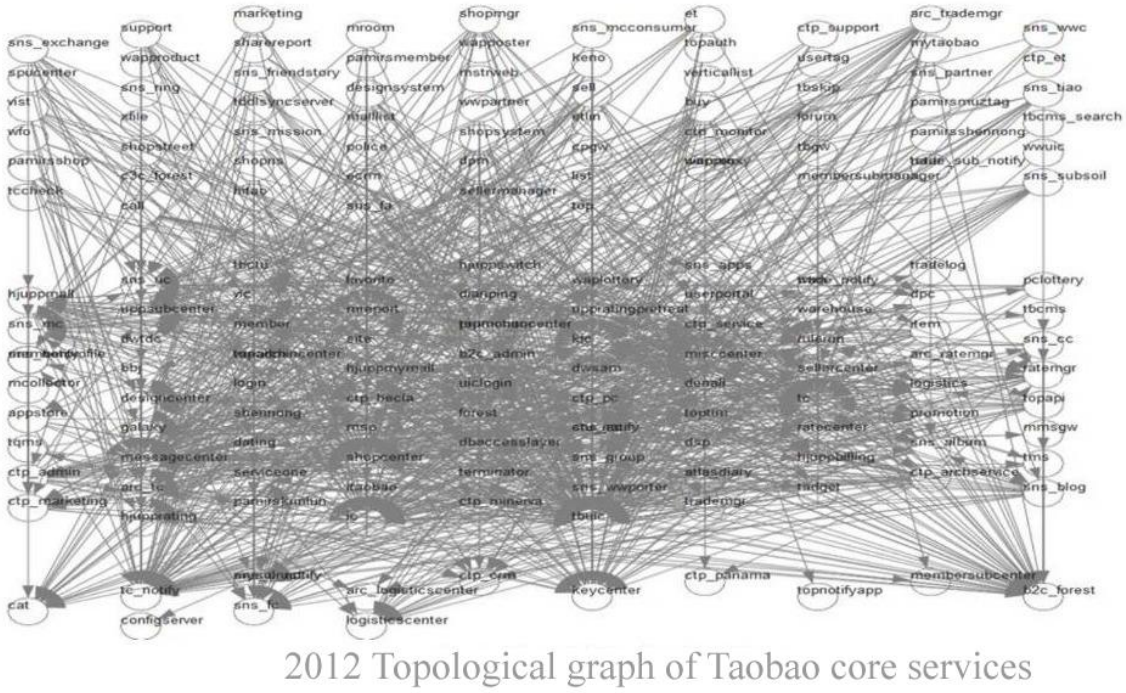

Figure 1 The Messy SOA Dependencies (Image from the Internet)

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.

1.2 nano-SOA Architecture

Through extensive exploration and practice with real projects, I have defined, implemented and improved the patented 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.

2. BaiY Port Switch Service (BYPSS)

BaiY Port Switch Service (BYPSS) is designed for providing a high available, strongly consistent and high performance distributed coordination and message dispatching service which supports ten billion level ports, one hundred thousand level nodes, and millions to ten millions of messages processed per second. The key concepts of the patented algorithm 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 two seconds; 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.

2.1 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.

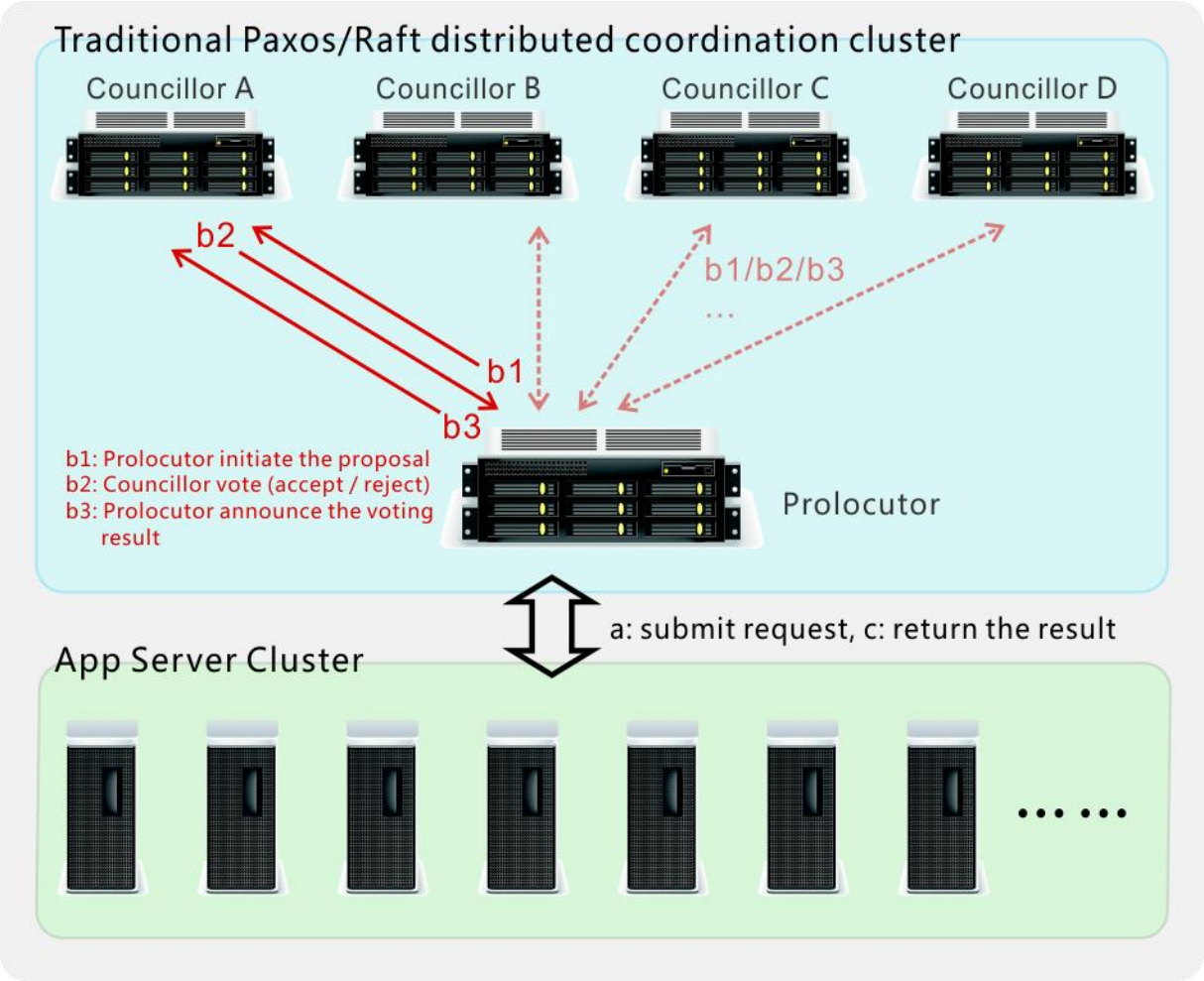

Figure 2 Traditional Paxos/Raft distributed coordination cluster

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 three 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: 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 and July 22, 2016; July 22, 2013 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 µSOA 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.

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.

2.2 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; supports multiple active IDC.
Consistency	Strong consistency; the master node is elected by the quorum.	Strong consistency; multi-replica.
Concurrency	Tens of millions of concurrent connections; hundreds of thousands of concurrent nodes.	Up to 5,000 nodes.
Capacity	Each 10GB memory can hold about 100 million message ports; each 1TB memory can hold about ten billion message ports; two-level concurrent Hash table structure allows capacity to be linearly extended to PB level.	Usually supports up to tens 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 three 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.	The characteristics of the algorithm itself make it impossible to support batch operations; less than 100 requests per second. (Because each atomic operation requires three 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 tolerance	The system goes offline when there is no quorum partition, but broadcast storm will not occur.	The system goes offline when there is no quorum partition. It is possible to produce a broadcast storm aggravated the network failure.
Message	Yes and with high performance: both the server	None.

Item	BYPSS	ZooKeeper, Consul, etcd…
dispatching	and client support automatic message batching.	
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. Failed: successfully discover the current owner of the service.	Success: lock acquired successfully. 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.

2.3 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: about 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 tens of billions to hundreds of billions of message ports capacity. This provides a good foundation for the fine coordination of distributed dusters. Compared with the traditional stateless cluster, BYPSS-based fine collaborative dusters 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 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 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 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.

Item	BYPSS HPC	Traditional Stateless Cluster
		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 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. Compared with distributed caching, the advantages of local cache is very obvious: 1. Eliminates the network latency required by query requests and reduces network load (See "Item 2" in this table for details). 2. Get the expanded data structures directly from memory, without a lot of data serialization and deserialization work. The server's local cache hit rate can be further improved if the appropriate rules for user owner selection can be followed, for example: a) Group users by tenant (company, department, site); b) Group users by region (geographical location, map area in the game); c) Group users by interest characteristics (game team, product preference). 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. This allows most of the data associated with a user or a group of users to be cached locally. This not only	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. The read pressure of the backend database server is high. Additional optimizations are required, such as horizontal partitioning, vertical partitioning, and read / write separation.

Item	BYPSS HPC	Traditional Stateless Cluster
	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.	
4 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. 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. This can greatly reduce the write pressure of the backend database. 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. 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). 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	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 is very high. A plurality of nodes may compete to update the same record simultaneously, further increasing the burden on the database. 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
	a single batch transaction, resulting in further efficiency improvements. 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.	
5 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. 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. 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 message middleware (e.g.: Kafka, RocketMQ, RabbitMQ, ZeroMQ, etc.). 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. 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. 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	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. 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
	between players within the same map instance (AOI). 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 enterprise applications, nearly 100% of the communications are from within the enterprise members. 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 Bal.	Clusters can be scheduled using a combination of active and passive load balancing. 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). 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. 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. 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. 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 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 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. 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. 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. The read pressure of the backend database server is high. Additional optimizations are required, such as horizontal partitioning, vertical partitioning, and read / write separation. 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 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 of a sudden failure of a modern PC server is also very	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 backend database is very high.

Item	BYPSS HPC	Traditional Stateless Cluster
	low. 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. This can greatly reduce the write pressure of the backend database. 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. 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. 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.	A plurality of nodes may compete to update the same record simultaneously, further increasing the burden on the database. 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 Bal.	Clusters can be scheduled using a combination of active and passive load balancing. Passive balancing: Each node in the cluster periodically unloads videos that are no longer active, 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,	When recovering from a fault, the balance is less than the BYPSS active balanced cluster. However, there is no significant difference under normal circumstances. In addition, SOA architecture tends to imbalance between multiple services,

Item	BYPSS HPC	Traditional Stateless Cluster
	clusters are balanced). 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. 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.	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.

References

M.Swientek, U.Bleimann and P.S.Dowland, Service-Oriented Architecture: Performance Issues and Approaches.

SOA Performance Considerations, Microsoft MSDN.

Roberdan (2007) SOA in the Real World, Microsoft.

Takanori Ueda, Takuya Nakaike, and Moriyoshi Ohara (2016) Workload Characterization for Microservices, IBM Research – Tokyo.

Amir, Y. and Kirsch, J. (2008) Paxos for system builders.

Diego Ongaro and John Ousterhout, In search of an Understandable Consensus Algorithm (Extended Version), Stanford University.

Leslie Lamport (2005) Fast Paxos

Leslie Lamport (2001) Paxos Made Simple.