A distributed Redis framework for use in the UCWW

Conference Paper · October 2014

DOI: 10.1109/CyberC.2014.50

4 authors, including:

Zhanlin Ji
University of Limerick
86 PUBLICATIONS 611 CITATIONS
SEE PROFILE

Ivan Ganchev
Plovdiv University "Paisii Hilendarski"
297 PUBLICATIONS 1,686 CITATIONS
SEE PROFILE

Mairtin S. O'Droma
University of Limerick
272 PUBLICATIONS 1,757 CITATIONS
SEE PROFILE

A Distributed Redis Framework for Use in the UCWW

Zhanlin Ji[1,2], Ivan Ganchev[1], Máirtín O'Droma[1], Tiefan Ding[2]
[1]*Telecommunications Research Centre, University of Limerick, Ireland*
{Zhanlin.Ji; Ivan.Ganchev; Mairtin.ODroma}@ul.ie;
[2]*College of Information, Hebei United University, China*
dtfsmile1014@gmail.com

***Abstract*—The performance of data access plays an important role for the applications of the emerging ubiquitous consumer wireless world (UCWW). As a 'Not only SQL' (NoSQL) database, Redis has been deployed in the UCWW cloud as an open-source key value store acting as the key component in the system. Given the limited performance of a single Redis node, to improve the system's performance and handle large amounts of requests from the web applications in the UCWW system, a distributed Redis framework has been designed and is proposed in this paper. Considering the complex management of the Redis cluster, in the distributed Redis framework the ZooKeeper is used to manage the Redis nodes at the database layer and service governance is utilized at the service layer. With this layered design, the resultant framework is flexible and scalable, and could be easily integrated into the UCWW cloud.**

Keywords-UCWW; Distributed Framework; ZooKeeper; Redis; Publish/Subscribe Design Pattern; Spring

I. Introduction

The Ubiquitous Consumer Wireless World (UCWW) [1], is a new, directed, evolutionary step in, and a significant change to, the global wireless techno-business environment encompassing the present and future rapid growth of wireless communications technologies and networks. In the UCWW, most of applications will be cloud-based - either light-weight HTML5 applications or native Android applications on the client side; and Java Platform Enterprise Edition (Java EE) [2] applications at the web layer and Hadoop [3] applications at the cloud layer on the server side. To design a high-throughput and high-availability system, the NginX [4] could be used as a load balancer to handle the users' requests by using an event-driven architecture instead of the thread technique. In addition, the Redis [5], acting as a 'Not only SQL' (NoSQL) database, could be utilized in the memory of servers to improve the system's performance. Figure 1 depicts the high-level view of the UCWW applications. The focus of this paper is on the web application layer and Redis layer.

The Redis is an open-source, memory-based, key-value database, which performs extremely well compared to traditional databases. It has been used by Amazon, Vmware, Windows, Google+, YouTube, LinkedIn, etc. As Redis is a single-thread, single-progress memory database, when the UCWW grows up, it may face the single point problem

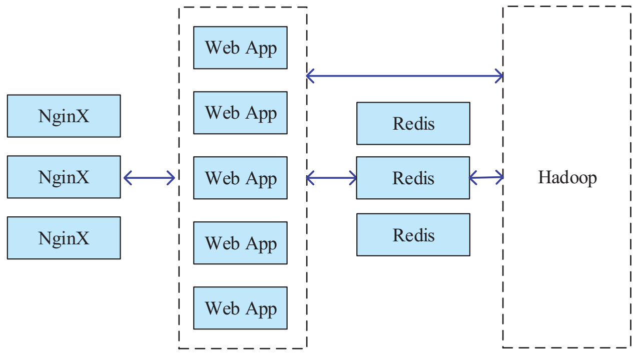

Figure 1. The high-level view of the applications in the UCWW environment.

[6]. To solve this problem, Redis will be developed as a distributed cluster for nodes' dynamic expansion and cluster's failover/tolerance, but as per June 2014, the distributed Redis 3.0 [7] is still in beta version and thus cannot be used for production environments.The Redis cluster provides an algorithm to shard data across multiple Redis nodes automatically. It constitutes a database-level distributed solution.

Beside increasing the query per second (QPS) rate and improving the distributed operations in the UCWW environment, a ZooKeeper-based distributed framework is proposed in this paper. ZooKeeper [8] is an open-source Apache project [9], which supports distributed configuration and locking techniques, along with publish/subscribe design patterns. Different from the Redis cluster specification, a ZooKeeper-based Redis framework was designed at the database layer and a ZooKeeper-based Redis service governance architecture was elaborated at the service layer.

II. System Architecture

The distributed Redis framework for use in the UCWW (Figure 2) was developed based on the layered design shown in Figure 1.

At the database layer, the framework relies on N (N mod 2 = 1) ZooKeeper servers, each running m Redis master nodes and m Redis slave nodes for replication.

The $N * m$ Redis servers are divided into N categories, whereby each category corresponds to one particular

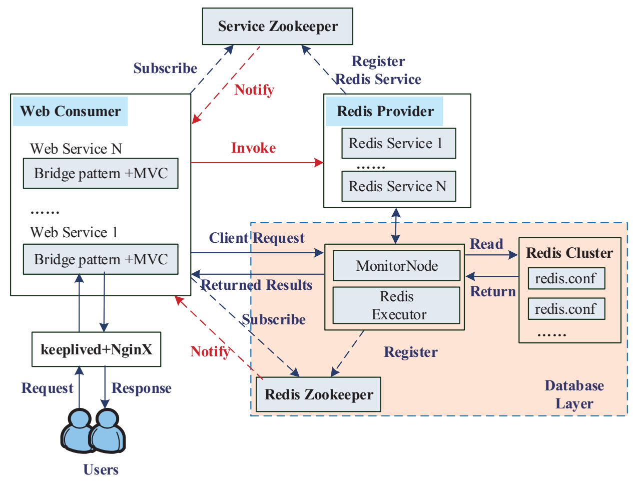

Figure 2. The distributed Redis framework for the UCWW.

UCWW service

A *RedisExecutor* algorithm was designed for the Redis cluster's writing and reading operations. For monitoring operations, a ZooKeeper is used, i.e., a *MonitorNode* acts as the parent node and a *Redis-monitor* function receives the heartbeat (described by an IP address and a port number) from the Redis servers and adds it to the parent node. When a Redis node fails, a watcher will notify the ZooKeeper to update the cluster list.

In the service layer, to develop a loosely-coupled system, a producer/consumer model is used for service governance, which provides sync-over-async and request-response messaging functions with a remote procedure call (RPC) [10] mechanism. The producer registers a service to the ZooKeeper, when a consumer subscribes to a service. The ZooKeeper notifies the consumer and the corresponding service is invoked.

III. System Design

A. Database layer

At the database layer, in order to support the service layer's producer/consumer model, a Redis package was designed with *Jedis* (Redis Java client) operations, i.e., Keys, Strings, Hashes, Lists, Sets, Sorted, Sets, etc. In the package, a functional *RedisExecutor* interface with one abstract execute method was designed. With the *ConcurrentHashMap* function, an abstract Java class *RandomRead* class implements the *RedisExecutor* to enable random read operations, and an abstract Java class *MultipleWrite* extends from *RandomRead* to provide category-level data sharing write operations. Three Java abstract classes (*Readlist*, *ReadSet*, and *ReadSum*) extend from the *MultipleWrite* classes to provide concurrent read operations. The return types are *List*, *Set*, and *Long*, respectively. Figure 3 depicts the Java classes' diagram of the Redis package.

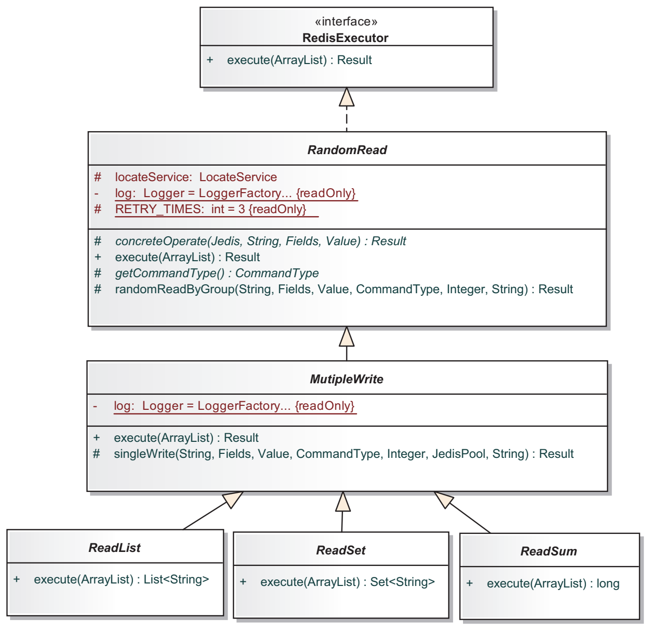
«interface»
RedisExecutor
+ execute(ArrayList) : Result
RandomRead
# locateService: LocateService
- log: Logger = LoggerFactory... {readOnly}
# RETRY_TIMES: int = 3 {readOnly}
# concreteOperate(Jedis, String, Fields, Value) : Result
+ execute(ArrayList) : Result
# getCommandType() : CommandType
# randomReadByGroup(String, Fields, Value, CommandType, Integer, String) : Result
MutipleWrite
- log: Logger = LoggerFactory... {readOnly}
+ execute(ArrayList) : Result
# singleWrite(String, Fields, Value, CommandType, Integer, JedisPool, String) : Result
ReadList
+ execute(ArrayList) : List<String>
ReadSet
+ execute(ArrayList) : Set<String>
ReadSum
+ execute(ArrayList) : long

Figure 3. The Java classes' diagram of the Redis package.

In the monitor section, an Apache Curator framework [11] was used to monitor the Redis server's connection status. The curator is an open-source ZooKeeper client framework, which provides simple application programming interface (API) for recipe's abstract encapsulation, i.e., including leader election, shared lock, path cache and watcher, etc. When a *Monitor* is instantiated in the XML-based configuration metadata, a *Curator* namespace is instantiated by the *CuratorFrameworkFactory.builder* function. For every category, the node contents are cached by the *PathChildrenCache* function and a cache Listener is started to watch the node. Figure 4 depicts the monitor processes' diagram.

B. Server side

The service layer provides a Redis service and a monitor service for the web layer. To design a loosely coupled system, a distributed service framework is used for service governance, i.e., exploring services' interfaces and providing distributed RPC mechanism. To develop the application in a productive manner, the Spring framework was selected as the development environment. The first step of the design was to define the service interfaces in the web layer and implement interfaces in the service layer. In the Spring's configuration metadata, the provider defines the remote method invocation (RMI) protocol port to explore services, and the ZooKeeper protocol port and IP address to initialize the register center. The consumer generates a remote service agent and uses as a local bean. With this service governance design, at the

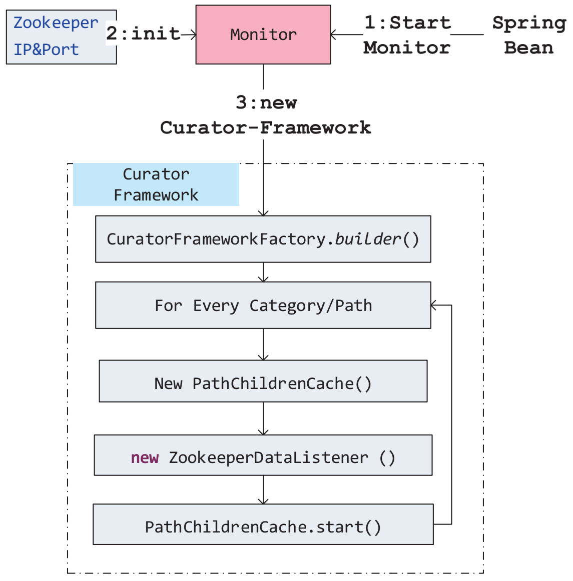

Figure 4. The monitor processes' diagram.

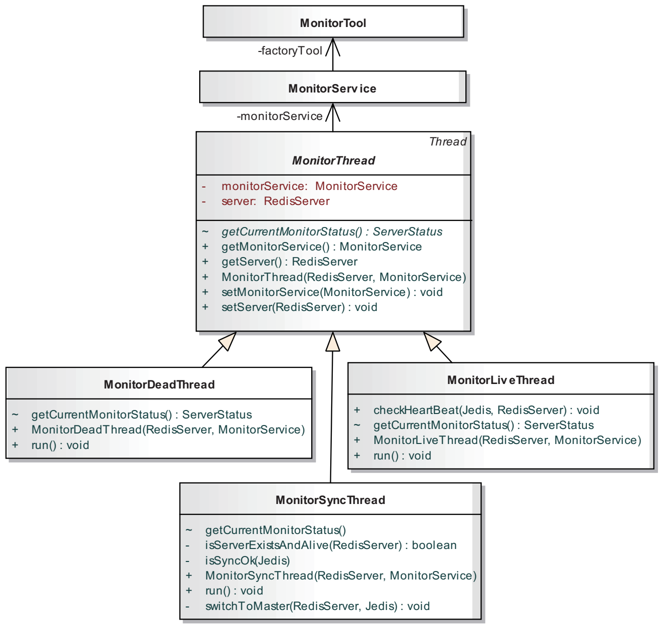
MonitorTool
-factoryTool
MonitorService
-monitorService
Thread
MonitorThread
- monitorService: MonitorService
- server: RedisServer
~ getCurrentMonitorStatus() : ServerStatus
+ getMonitorService() : MonitorService
+ getServer() : RedisServer
+ MonitorThread(RedisServer, MonitorService)
+ setMonitorService(MonitorService) : void
+ setServer(RedisServer) : void
MonitorDeadThread
~ getCurrentMonitorStatus() : ServerStatus
+ MonitorDeadThread(RedisServer, MonitorService)
+ run() : void
MonitorLiveThread
+ checkHeartBeat(Jedis, RedisServer) : void
~ getCurrentMonitorStatus() : ServerStatus
+ MonitorLiveThread(RedisServer, MonitorService)
+ run() : void
MonitorSyncThread
~ getCurrentMonitorStatus()
- isServerExistsAndAlive(RedisServer) : boolean
- isSyncOk(Jedis)
+ MonitorSyncThread(RedisServer, MonitorService)
+ run() : void
- switchToMaster(RedisServer, Jedis) : void

Figure 5. The Java classes' diagram of the monitor package.

web layer, the *ReferenceConfigure* Java class utilizes the RMI protocol to generate the RMI invoker instance; then the corresponding interface is transferred by the *getProxy*() method. At the service layer, after receiving the request, a corresponding exporter invokes the *AbstractProxyInvoker* instance and runs the corresponding service.

IV. Results

To demonstrate the successful operation of the proposed distributed Redis framework, a web application was designed and implemented. Similarly to the service layer, the Spring framework was used for rapid development. With the annotations and the fast model-view-controller (MVC) development tool, the JavaServer pages (JSPs) and the corresponding monitor controller were implemented. Inside the controller, a monitor service was started. Figure 5 shows the monitor service's Java classes' diagram. A *MonitorLiveThread*, a *MonitorSyncThread*, and a *MonitorDeadThread* all extend from the *MonitorThread* to monitor the Redis cluster. The heartbeat checking algorithm was implemented in the *MonitorLiveThread*.

Four Intel XEON PCs (E3-1220L CPU, 32GB RAM), with installed Hadoop 1.2, are used configured as master, slave1, slave2, and slave3, respectively. The Redis cluster is deployed on the four servers. The web applications are deployed on the master. To evaluate the system performance, five Redis groups were created by means of the ZooKeeper's zkCli.sh command line, named {1,2,3,4,5}, and representing five different mobile services in UCWW. On the four dedicated server PCs, 20 Redis servers are running with IP addresses 192.168.1.5–8 and ports 5000–5005.

At the web layer, a number of connections (between the database layer and the service layer) are utilized to send out mobileinfo?java&{service} reading requests from the web applications to the system. From these, there are {191, 196, 244, 196, 378} connections for services {1,2,3,4,5}, respectively. The average number of queries per second (QPS) is about 2750. Figure 6 illustrates the number of connections and QPS as functions of time. The results demonstrate that the designed Redis framework is able to run efficiently in a distributed fashion within a UCWW environment.

V. Conclusion

The design and implementation of a distributed Redis framework for use in the ubiquitous consumer wireless world (UCWW) is described in this paper. At the database layer, a ZooKeeper-based Redis cluster for distributed operations was designed and used. To simplify the access to the Redis cluster, a *RedisExecutor* was designed with a *ConcurrentHashMap* algorithm for database's writing and reading operations, and a Redis monitor was designed for checking the Redis server's connection status. At the service layer, to provide a Redis service and a monitor service for the upper layer, a service governance scheme was designed for exploring services' interfaces and providing distributed RPC mechanism. For the purposes of performance visualization, a Spring-based web application was designed. The results show that the designed Redis framework is able to run in a distributed fashion within a UCWW environment.

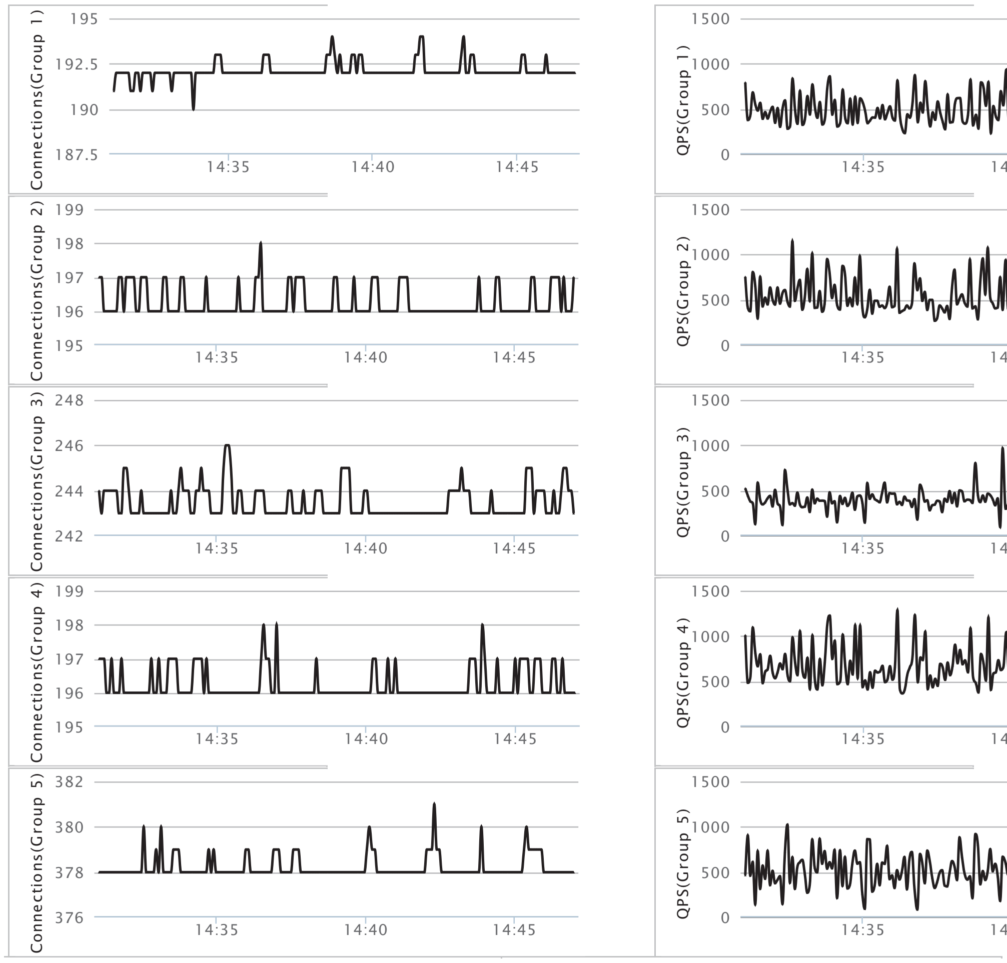

Figure 6. The performance visualization of the distributed Redis framework: number of connections (left); queries per second (right).

ACKNOWLEDGMENT

This work has been supported by the Government of Ireland Post-Doctoral Fellowship (GOIPD/2013/243), the Telecommunications Research Centre (TRC) of the University of Limerick, and the Natural Science Foundation of Hebei Province of China (F2012401050).

REFERENCES

[1] M. O'Droma, and I. Ganchev, "The creation of a ubiquitous consumer wireless world through strategic ITU-T standardization," Communications Magazine, IEEE, vol. 48, no. 10, pp. 158-165, 2010.

[2] E. Jendrock, I. Evans, D. Gollapudi, K. Haase, R. Cervera-Navarro, C. Srivathsa, and W. Markito, Java EE 7 Tutorial: Pearson Education, 2014.

[3] A. Rabkin, and R. H. Katz, "How Hadoop Clusters Break," IEEE Software, vol. 30, no. 4, pp. 88-94, 2013.

[4] W. Reese, "Nginx: the high-performance web server and reverse proxy," Linux Journal, vol. 2008, no. 173, pp. x.1-x.2, 2008.

[5] J. Zawodny, "Redis: Lightweight key/value store that goes the extra mile," Linux Magazine, August, vol. 31, 2009.

[6] J. Han, E. Haihong, G. Le, and J. Du, "Survey on NoSQL database," in the 6th international conference on Pervasive computing and applications (ICPCA), pp. 363-366, Port Eliza, South Africa, 2011.

[7] Redis, "Redis cluster Specification, 6th beta of Redis 3.0.0," 2014, http://www.redis.io.

[8] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "ZooKeeper: wait-free coordination for internet-scale systems," in Proceedings of the 2010 USENIX annual technical conference, Boston, USA, pp. x.1-x.14, June 22-25, 2010.

[9] Apache, "Apache ZooKeeper," 2014, http://zookeeper.apache.org/.

[10] T. Haynes, and N. Williams, "Remote Procedure Call (RPC) Security Version 3," IETF, 2011.

[11] Apache. "Apache curator," 2014, http://curator.apache.org/.