UNITED STATES DISTRICT COURT
FOR THE WESTERN DISTRICT OF TEXAS
WACO DIVISION

BCS SOFTWARE, LLC, Plaintiff v. **FACEBOOK, INC.,** Defendant	**Case No. 6:20-cv-00695** **JURY TRIAL DEMANDED**

COMPLAINT FOR PATENT INFRINGEMENT

Plaintiff BCS Software, LLC ("Plaintiff" or "BCS") hereby asserts the following claims for patent infringement against Facebook, Inc. ("Defendant" or "Facebook"), and alleges, on information and belief, as follows:

THE PARTIES

1. BCS Software, LLC is a limited liability company organized and existing under the laws of the Texas with its principal place of business in Austin, Texas.

2. Facebook is a corporation organized and existing under the laws of the State of Delaware having a principal place of business at 1 Hacker Way, Bldg. 10, Menlo Park, California 94025-1456.

3. Facebook may be served with process through its registered agent, Corporation Service Company, DBA CSC – Lawyers Inco, 211 East 7th Street, Suite 620, Austin, Texas 78701.

4. On information and belief, since about April 2009, Facebook has been registered to do business in the state of Texas under Texas SOS file number 0801108427.

JURISDICTION AND VENUE

5. This action arises under the patent laws of the United States, 35 U.S.C. § 1, *et seq*. This Court has subject matter jurisdiction under 28 U.S.C. §§ 1331 and 1338(a).

6. Defendant has committed acts of infringement in this judicial district.

7. On information and belief, Defendant has a regular and established place of business in this judicial district at 9420 Research Blvd, Austin, Texas 78759.

8. On information and belief, the Court has personal jurisdiction over Defendant because Defendant has committed, and continues to commit, acts of infringement in the state of Texas, has conducted business in the state of Texas, and/or has engaged in continuous and systematic activities in the state of Texas.

9. On information and belief, Defendant's instrumentalities that are alleged herein to infringe were and continue to be used, imported, offered for sale, and/or sold in the Western District of Texas.

10. Venue is proper in the Western District of Texas pursuant to 28 U.S.C. § 1400(b).

U.S. PATENT NO. 7,890,809

11. BCS is the owner, by assignment, of U.S. Patent No. 7,890,809 ("the '809 Patent"), entitled HIGH LEVEL OPERATIONAL SUPPORT SYSTEM, which issued on February 15, 2011. A copy of the '809 Patent is attached as **Exhibit A**.

12. The '809 Patent is valid, enforceable, and was duly issued in full compliance with Title 35 of the United States Code.

13. The '809 Patent was invented by Messrs. Blaine Nye and David Sze Hong.

14. The priority date for the '809 Patent is at least May 1, 2003.

15. The expiration date of the '809 Patent is August 21, 2023.

16. The ’809 Patent has been referenced by 18 United States Patents, United States Patent Applications and foreign patents.

17. The ’809 Patent was examined by United States Patent Examiner Joshua Lohn. During the examination of the ’809 Patent, the United States Patent Examiner searched for prior art in the following US Classifications: 714/38, 714/47, 719/320.

18. After conducting a search for prior art during the examination of the ’809 Patent, the United States Patent Examiner identified and cited U.S. Patent No. 6,748,555 to Teegan et al as one of the most relevant prior art references found during the search.

19. After conducting a search for prior art during the examination of the ’809 Patent, the United States Patent Examiner identified and cited U.S. Patent No. 6,862,698 to Shyu as one of the most relevant prior art references found during the search.

20. After conducting a search for prior art during the examination of the ’809 Patent, the United States Patent Examiner identified and cited U.S. Patent No. 7,003,560 to Mullen et al as one of the most relevant prior art references found during the search.

21. After conducting a search for prior art during the examination of the ’809 Patent, the United States Patent Examiner identified and cited U.S. Patent No. 7,100,195 to Underwood as one of the most relevant prior art references found during the search.

22. After conducting a search for prior art during the examination of the ’809 Patent, the United States Patent Examiner identified and cited U.S. Patent Application No. 2003/0037288 by Harper et al as one of the most relevant prior art references found during the search.

23. After conducting a search for prior art during the examination of the ’809 Patent, the United States Patent Examiner identified and cited U.S. Patent Application No.

2003/0204791 by Helgren et al as one of the most relevant prior art references found during the search.

24. After conducting a search for prior art during the examination of the '809 Patent, the United States Patent Examiner identified and cited U.S. Patent Application No. 2004/0073566 by Trivedi as one of the most relevant prior art references found during the search.

25. After conducting a search for prior art during the examination of the '809 Patent, the United States Patent Examiner identified and cited U.S. Patent Application No. 2004/0088401 by Tripathi et al as one of the most relevant prior art references found during the search.

26. After conducting a search for prior art during the examination of the '809 Patent, the United States Patent Examiner identified and cited U.S. Patent Application No. 2005/0044535 by Coppert as one of the most relevant prior art references found during the search.

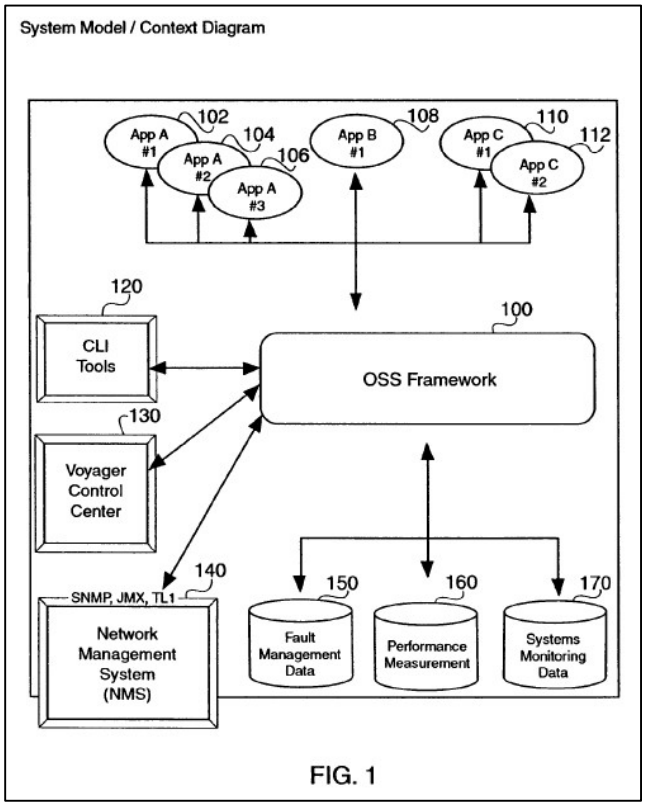
27. After conducting a search for prior art during the examination of the '809 Patent, the United States Patent Examiner identified and cited U.S. Patent Application No. 6,748,555 by Shyu as one of the most relevant prior art references found during the search.

28. The '809 Patent relates to:

> A high level Operational Support System (OSS) framework provides the infrastructure and analytical system to enable all applications and systems to be managed dynamically at runtime regardless of platform or programming technology. Applications are automatically discovered and managed. Java applications have the additional advantage of auto-inspection (through reflection) to determine their manageability. Resources belonging to application instances are associated and managed with that

> application instance. This provides operators the ability to not only manage an application, but its distributed components as well. They are presented as belonging to a single application instance node that can be monitored, analyzed, and managed. The OSS framework provides the platform-independent infrastructure that heterogeneous applications require to be monitored, controlled, analyzed and managed at runtime. New and legacy applications written in C++ or Java are viewed and manipulated identically with zero coupling between the applications themselves and the tools that scrutinize them.

'809 Patent (Abstract).



Id. (Figure 1).

29. The field of the invention is to improvements in "wireless communication carriers. More particularly, it relates to operational support system (OSS), application/systems management, and network management." *Id.*, col. 1:17-20.

30. As disclosed in the '809 Patent, "[m]any network management technologies exist that allow operators to manage applications and devices at runtime. For instance, SNMP, TL1 and JMX each attempt to provide operators with the ability to manipulate and affect change at runtime." *Id.*, col. 1:22-26.

31. As disclosed in the '809 Patent, "[t]he fundamental of each is similar. It is to manipulate the objects of an application through messaging." *Id.*, col. 1:26-27.

32. As disclosed in the '809 Patent, "SNMP is the standard basic management service for networks that operate in TCP/IP environments. It is intended primarily to operate well-defined devices easily and does so quite successfully. However, it is limited to the querying and updating of variables." *Id.*, col. 1:28-32.

33. As disclosed in the '809 Patent, "Transaction Language 1 (TL1) is a set of ASCII-based instructions, or 'messages,' that an operations support system (OSS) uses to manage a network element (NE) and its resources. *Id.*, col. 1:32-35.

34. As disclosed in the '809 Patent, "JMX is a Java centric technology that permits the total management of objects: not only the manipulation of fields, but also the execution of object operations. It is designed to take advantage of the Java language to allow for the discovery and manipulation of new or legacy applications or devices." *Id.*, col. 1:35-40.

35. As disclosed in the '809 Patent, "Operational Support for enterprise applications is currently realized using a variety of technologies and distinct, separate services. For instance, network management protocols (SNMP, JMX, TL1, etc.) provide runtime configuration and some provide operation invocation, but these technologies are not necessarily geared toward applications." *Id.*, col. 1:40-45.

36. As disclosed in the '809 Patent, "[s]ome are language specific (e.g., JMX) and require language agnostic bridging mechanisms that must be implemented, configured and maintained. SNMP is generic (e.g., TL1 and SNMP) and very simple in nature, but it requires application developers to implement solutions to common OSS tasks on top of SNMP. *Id.*, col. 1:46-51.

37. As disclosed in the '809 Patent, "TL1 is also ASCII based and generic. However, while it is very flexible and powerful, it is another language that must be mastered, and it's nature is command line based. As a result, it is not intuitively based in presentation layer tools. While all the technologies have their respective benefits, they do not provide direct means of providing higher level OSS functionality. Conventionally, applications are monitored, analyzed and managed at runtime." *Id.*, col. 1:52-59.

38. As disclosed in the '809 Patent, one or more claims "provid[e] a high level operational support system framework comprises monitoring a health of a plurality of applications. The health of the plurality of applications is assessed, and the health of the plurality of applications is analyzed, whereby each of the plurality of applications are managed dynamically at runtime regardless of a platform of each of the plurality of applications." *Id.*, col. 1:64–2:3.

39. Consequently, the '809 Patent improves the computer functionality itself and represents a technological improvement to the operation of computers.

NOTICE OF BCS' PATENTS

40. Plaintiff is the owner, by assignment, of U.S. Patent No. 6,240,421 (the "'421 Patent"), entitled "System, software and apparatus for organizing, storing and retrieving

information from a computer database," which issued on May 29, 2001. A copy of the '421 Patent is available at https://patents.google.com/patent/US6240421B1/en?oq=6240421.

41. Plaintiff is the owner, by assignment, of U.S. Patent No. 6,421,821 (the "'821 Patent"), entitled "Flow chart-based programming method and system for object-oriented languages," which issued on July 16, 2002. A copy of the '821 Patent is available at https://patents.google.com/patent/US6421821B1/en?oq=6421821.

42. Plaintiff is the owner, by assignment, of U.S. Patent No. 6,438,535 (the "'535 Patent"), entitled "Relational database method for accessing information useful for the manufacture of, to interconnect nodes in, to repair and to maintain product and system units," which issued on August 20, 2002. A copy of the '535 Patent is available at https://patents.google.com/patent/US6438535B1/en?oq=6438535.

43. Plaintiff is the owner, by assignment, of U.S. Patent No. 6,658,377 (the "'377 Patent"), entitled "Method and system for text analysis based on the tagging, processing, and/or reformatting of the input text," which issued on December 2, 2003. A copy of the '377 Patent is available at https://patents.google.com/patent/US6658377B1/en?oq=6658377.

44. Plaintiff is the owner, by assignment, of U.S. Patent No. 6,662,179 (the "'179 Patent"), entitled "Relational database method for accessing information useful for the manufacture of, to interconnect nodes in, to repair and to maintain product and system units," which issued on December 9, 2003. A copy of the '179 Patent is available at https://patents.google.com/patent/US6662179B2/en?oq=6662179.

45. Plaintiff is the owner, by assignment, of U.S. Patent No. 6,895,502 (the "'502 Patent"), entitled "Method and system for securely displaying and confirming request to

perform operation on host computer," which issued on May 17, 2005. A copy of the '502 Patent is available at https://patents.google.com/patent/US6895502B1/en?oq=6895502.

46. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,200,760 (the "'760 Patent"), entitled "System for persistently encrypting critical software data to control the operation of an executable software program," which issued on April 3, 2007. A copy of the '760 Patent is available at https://patents.google.com/patent/US7200760B2/en?oq=7200760.

47. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,302,612 (the "'612 Patent"), entitled "High level operational support system," which issued on November 27, 2007. A copy of the '612 Patent is available at https://patents.google.com/patent/US7302612B2/en?oq=7302612.

48. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,533,301 (the "'301 Patent"), entitled "High level operational support system," which issued on May 12, 2009. A copy of the '301 Patent is available at https://patents.google.com/patent/US7533301B2/en?oq=7533301.

49. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,730,129 (the "'129 Patent"), entitled "Collaborative communication platforms," which issued on June 1, 2010. A copy of the '129 Patent is available at https://patents.google.com/patent/US7730129B2/en?oq=7730129.

50. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,774,296 (the "'296 Patent"), entitled "Relational database method for accessing information useful for the manufacture of, to interconnect nodes in, to repair and to maintain product and system

units," which issued on August 10, 2010. A copy of the '296 Patent is available at https://patents.google.com/patent/US7774296B2/en?oq=7774296.

51. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,840.893 (the "'893 Patent"), entitled "Display and manipulation of web page-based search results," which issued on November 23, 2010. A copy of the '893 Patent is available at https://patents.google.com/patent/US7840893B2/en?oq=7840893.

52. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,895,282 (the "'282 Patent"), entitled "Internal electronic mail system and method for the same," which issued on February 22, 2011. A copy of the '282 Patent is available at https://patents.google.com/patent/US7895282B1/en?oq=7895282.

53. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,996,464 (the "'464 Patent"), entitled "Method and system for providing a user directory," which issued on August 9, 2011. A copy of the '464 Patent is available at https://patents.google.com/patent/US7996464B1/en?oq=7996464.

54. Plaintiff is the owner, by assignment, of U.S. Patent No. 7,996,469 (the "'469 Patent"), entitled "Method and system for sharing files over networks," which issued on August 9, 2011. A copy of the '469 Patent is available at https://patents.google.com/patent/US7996469B1/en?oq=7996469.

55. Plaintiff is the owner, by assignment, of U.S. Patent No. 8,171,081 (the "'081 Patent"), entitled "Internal electronic mail within a collaborative communication system," which issued on May 1, 2012. A copy of the '081 Patent is available at https://patents.google.com/patent/US8171081B1/en?oq=8171081.

56. Plaintiff is the owner, by assignment, of U.S. Patent No. 8,176,123 (the "'123 Patent"), entitled "Collaborative communication platforms," which issued on May 8, 2012. A copy of the '123 Patent is available at https://patents.google.com/patent/US8176123B1/en?oq=8176123.

57. Plaintiff is the owner, by assignment, of U.S. Patent No. 8,285,788 (the "'788 Patent"), entitled "Techniques for sharing files within a collaborative communication system," which issued on October 9, 2012. A copy of the '788 Patent is available at https://patents.google.com/patent/US8285788B1/en?oq=8285788.

58. Plaintiff is the owner, by assignment, of U.S. Patent No. 8,554,838 (the "'838 Patent"), entitled "Collaborative communication platforms," which issued on October 8, 2013. A copy of the '838 Patent is available at https://patents.google.com/patent/US8554838B1/en?oq=8554838.

59. Plaintiff is the owner, by assignment, of U.S. Patent No. 8,819,120 (the "'120 Patent"), entitled "Method and system for group communications," which issued on August 26, 2014. A copy of the '120 Patent is available at https://patents.google.com/patent/US8819120B1/en?oq=8819120.

60. Plaintiff is the owner, by assignment, of U.S. Patent No. 8,984,063 (the "'063 Patent"), entitled "Techniques for providing a user directory for communication within a communication system," which issued on March 17, 2015. A copy of the '063 Patent is available at https://patents.google.com/patent/US8984063B1/en?oq=8984063.

61. Plaintiff is the owner, by assignment, of U.S. Patent No. 9,396,456 (the "'456 Patent"), entitled "Method and system for forming groups in collaborative communication

system," which issued on July 19, 2016. A copy of the '456 Patent is available at https://patents.google.com/patent/US9396456B1/en?oq=9396456.

DEFENDANT'S SYSTEM

62. Upon information and belief, Defendant makes, uses, and operates the Facebook.com platform, which includes the Apache Zookeeper service, which is exemplified by the following references:

- The Underlying Technology of Messages ("**Underlying Technology**"), *available at* https://www.facebook.com/notes/facebook-engineering/the-underlying-technology-of-messages/454991608919 (last accessed July 29, 2020);
- Observers: Making ZooKeeper Scale Even Further ("**ZooKeeper Scale**"), *available at* https://www.facebook.com/notes/cloudera/observers-making-zookeeper-scale-even-further/204351007002/ (last accessed July 29, 2020);
- What is Zookeeper ("**What is Zookeeper**"), *available at* http://www.corejavaguru.com/bigdata/zookeeper/what-is-zookeeper (last accessed July 29, 2020);
- Welcome to Apache ZooKeeper ("**Apache ZooKeeper**"), *available at* https://zookeeper.apache.org/ (last accessed July 29, 2020);
- ZooKeeper: A Distributed Coordination Service for Distributed Applications ("**Distributed Coordination**"), *available at* https://zookeeper.apache.org/doc/r3.2.2/zookeeperOver.html (last accessed July 29, 2020);
- What is Apache Zookeeper? ("**Zookeeper_intellipaat**"), *available at* https://intellipaat.com/blog/what-is-apache-zookeeper/ (last accessed July 29, 2020);
- Start making? Zookeeper's API ("**Zookeeper's API**"), *available at* https://topic.alibabacloud.com/a/start-making-zookeeper39s-api_8_8_30841990.html (last accessed July 29, 2020);
- ZooKeeper by Benjamin Reed, Flavio Junqueira ("**ZooKeeper by Benjamin Reed**"), *available at* https://www.oreilly.com/library/view/zookeeper/9781449361297/ch01.html (last accessed July 29, 2020);

- Apache Zookeeper Tutorial ("**Zookeeper Tutorial**"), *available at* https://www.dezyre.com/hadoop-tutorial/zookeeper-tutorial (last accessed July 29, 2020);

- Curator RPC ("**Curator RPC**"), *available at* http://zookeeper-user.578899.n2.nabble.com/ANN-Curator-RPC-td7580113.html (last accessed July 29, 2020);

- Introduction to Thrift ("**Introduction to Thrift**"), *available at* https://thrift-tutorial.readthedocs.io/en/latest/intro.html (last accessed July 29, 2020);

- Service Discovery Server ("**Service Discovery Server**"), *available at* https://curator.apache.org/curator-x-discovery-server/index.html (last accessed July 29, 2020);

- Apache Thrift ("**Apache Thrift**"), *available at* https://en.wikipedia.org/wiki/Apache_Thrift (last accessed July 29, 2020);

- ZooKeeper Flavio Junqueira Benjamin Reed ("**ZooKeeper Flavio Junqueira**"), *available at* http://www.54manong.com/ebook/%E5%A4%A7%E6%95%B0%E6%8D%AE/20181208232851/ZooKeeper-Flavio%20Junqueira%20&%20Benjamin%20Reed/ZooKeeper-Flavio%20Junqueira%20&%20Benjamin%20Reed.html (last accessed July 29, 2020);

- ZooKeeper Monitoring ("**ZooKeeper Monitoring**"), *available at* https://www.site24x7.com/plugins/zookeeper-monitoring.html (last accessed July 29, 2020);

- ZooKeeper ("**DataDog**"), *available at* https://docs.datadoghq.com/integrations/zk/ (last accessed July 29, 2020);

- Chapter 4. Dealing with State Change ("**State Change**"), *available at* https://www.oreilly.com/library/view/zookeeper/9781449361297/ch04.html (last accessed July 29, 2020);

- ZooKeeper Programmer's Guide ("**Programmer's Guide**"), *available at* https://zookeeper.apache.org/doc/r3.3.5/zookeeperProgrammers.html (last accessed July 29, 2020);

- Architecture of ZAB – ZooKeeper Atomic Broadcast protocol ("**Architecture of ZAB**"), *available at* https://distributedalgorithm.wordpress.com/tag/zookeeper/ (last accessed July 29, 2020); and

- Introduction to Apache ZooKeeper ("**Introduction**"), *available at* https://hadooptechblog.wordpress.com/2015/12/29/introduction-to-apache-zookeeper/ (last accessed July 29, 2020).

63. The information contained in the references identified in paragraph 62 is incorporated by reference as if set forth fully herein.

64. The information contained in the references identified in paragraph 63 accurately describes the operation and functionality of the Apache Zookeeper service.

COUNT I
(Infringement of U.S. Patent No. 7,890,809)

65. BCS incorporates paragraphs 1-64 herein by reference.

66. Defendant has been on notice of the '809 Patent at least as early as the date it received service of this complaint.

67. Upon information and belief, Defendant has infringed and continues to infringe one or more claims, including Claim 1, of the '809 Patent by making, using, and operating the Apache Zookeeper service.

68. Defendant, with knowledge of the '809 Patent, infringes the '809 Patent by inducing others to infringe the '809 Patent. In particular, Defendant intends to induce its customers to infringe the '809 Patent by encouraging its customers to use the Apache Zookeeper service.

69. Defendant also induces others, including its customers, to infringe the '809 Patent by providing technical support for the use of the Apache Zookeeper service.

70. Upon information and belief, at all times Defendant owns and controls the operation of the Apache Zookeeper service.

71. Claim 1 of the '809 Patent recites:

1. A method of providing a high level support framework, comprising:

monitoring from a physical server a health of a plurality of client applications and a health of said plurality of client applications' distributed components, using a common monitoring protocol, said monitoring being independent of a programming technology of said plurality of client applications and respective distributed components;

assessing said health of said plurality of client applications and said respective distributed components; and

associating said health of said plurality of client applications and said respective distributed components as belonging to a single application node.

72. With the ThinQ product, Defendant provides a high-level operational support system framework.

73. The Apache Zookeeper satisfies the claim element "a high level support framework."

74. The Apache Zookeeper service is an open source centralized service for coordination of distributed applications and also known as king of coordination.

The Underlying Technology of Messages

15 November 2010 at 12:46

We're launching a new version of Messages today that combines chat, SMS, email, and Messages into a real-time conversation. The product team spent the last year building out a robust, scalable infrastructure. As we launch the product, we wanted to share some details about the technology.

The current Messages infrastructure handles over 350 million users sending over 15 billion person-to-person messages per month. Our chat service supports over 300 million users who send over 120 billion messages per month. By monitoring usage, two general data patterns emerged:

Since Messages accepts data from many sources such as email and SMS, we decided to write an application server from scratch instead of using our generic Web infrastructure to handle all decision making for a user's messages. It interfaces with a large number of other services: we store attachments in Haystack, wrote a user discovery service on top of Apache ZooKeeper, and talk to other infrastructure services for email account verification, friend relationships, privacy decisions, and delivery decisions (for example, should a message be sent over chat or SMS). We spent a lot of time making sure each of these services are reliable, robust, and performant enough to handle a real-time messaging system.

Underlying Technology.

As readers of our previous post on the subject will recall, ZooKeeper is a distributed coordination service suitable for implementing coordination primitives like locks and concurrent queues. One of ZooKeeper's great strengths is its ability to operate at scale. Clusters of only five or seven machines can often serve the coordination needs of several large applications.

ZooKeeper Scale.

ZooKeeper is a distributed, open-source coordination service for distributed applications. It is also called as 'King of Coordination'. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

What is Zookeeper.

> Apache ZooKeeper is an effort to develop and maintain an open-source server which enables highly reliable distributed coordination.

> ZooKeeper is a centralized service for maintaining configuration information, naming, providing distributed synchronization, and providing group services. All of these kinds of services are used in some form or another by distributed applications. Each time they are implemented there is a lot of work that goes into fixing the bugs and race conditions that are inevitable. Because of the difficulty of implementing these kinds of services, applications initially usually skimp on them, which make them brittle in the presence of change and difficult to manage. Even when done correctly, different implementations of these services lead to management complexity when the applications are deployed.

Apache ZooKeeper.

75. The Zookeeper server satisfies the claim element "a physical server."

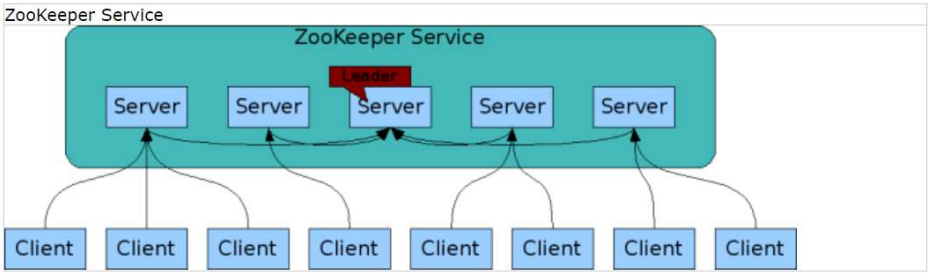
76. The clients connected to the zookeeper server satisfies the claim element "a plurality of client applications."

77. The children nodes such as z-nodes/data-nodes of applications can be created. The z-node is called as directory for storing data satisfies the claim element "plurality of client applications' distributed components."

78. The monitoring of health of session between client and zookeeper satisfies the claim element "health of a plurality of client applications."

79. A number of clients are connected to the zookeeper service. The zookeeper service allows the monitoring of the health of session between client and zookeeper (herein referred as plurality of client applications). Further, a notification is generated (herein inferred as monitoring of health) whenever a z-node/data-node (herein referred as plurality of client applications' distributed components) is created/deleted.

ZooKeeper is replicated. Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an ensemble.



The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

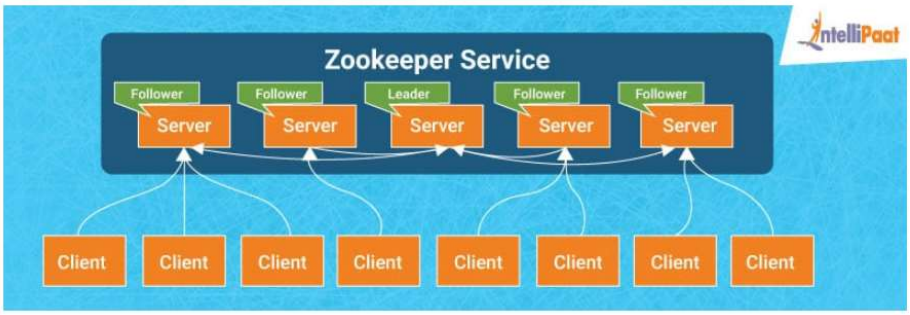
Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

Distributed Coordination.

ZooKeeper Architecture

Apache ZooKeeper works on the Client–Server architecture in which clients are machine nodes and servers are nodes.

The following figure shows the relationship between the servers and their clients. In this, we can see that each client sources the client library, and further they communicate with any of the ZooKeeper nodes.



Components of the ZooKeeper architecture has been explained in the following table.

Part	Description
Client	Client node in our distributed applications cluster is used to access information from the server. It sends a message to the server to let the server know that the client is alive, and if there is no response from the connected server the client automatically resends the message to another server.
Server	The server gives an acknowledgement to the client to inform that the server is alive, and it provides all services to clients.
Leader	If any of the server nodes is failed, this server node performs automatic recovery.
Follower	It is a server node which follows the instructions given by the leader.

Zookeeper_intellipaat.

Watcher
an object that is used to receive session events, which requires our own creation. Because Wacher is defined as an interface, so we need to implement one ourselves, and then initialize the class's instance and passes the zookeeper in the constructor. The client uses the Watcher interface to monitor and the health of the session between zookeeper. Establish or lose connectivity with the zookeeper server event occurs. They can also be used to monitor changes in zookeeper data. In the end, such as The zookeeper session expires, and the event is passed through the Watcher interface to notify the client the application.

Zookeeper's API.

> Another important issue with communication failures is the impact they have on synchronization primitives like locks. Because nodes can crash and systems are prone to network partitions, locks can be problematic: if a node crashes or gets partitioned away, the lock can prevent others from making progress. ZooKeeper consequently needs to implement mechanisms to deal with such scenarios. First, it enables clients to say that some data in the ZooKeeper state is *ephemeral*. Second, the ZooKeeper ensemble requires that clients periodically notify that they are alive. If a client fails to notify the ensemble in a timely manner, then all ephemeral state belonging to this client is deleted. Using these two mechanisms, we are able to prevent clients individually from bringing the application to a halt in the presence of crashes and communication failures.

ZooKeeper by Benjamin Reed.

> Along with this strong consistency guarantee, ZooKeeper also promises high availability, which can loosely be interpreted to mean that it can withstand a significant number of machine failures before the service stops being available to clients. ZooKeeper achieves this availability in a traditional way - by replicating the data that is being written and read amongst a small number of machines so that if one fails, there are others ready to take over without the client being any wiser.

ZooKeeper Scale.

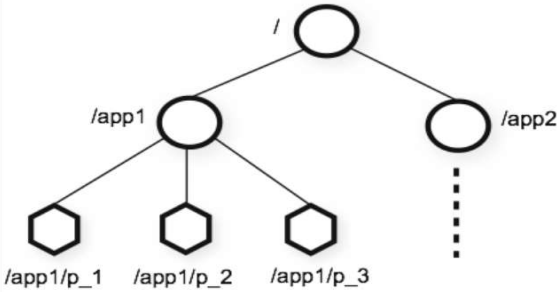
> In a distributed environment, coordinating and managing a service has become a difficult process. Apache ZooKeeper was used to solve this problem because of its simple architecture, as well as API, that allows developers to implement common coordination tasks like electing a master server, managing group membership, and managing metadata.
>
> Apache ZooKeeper is used for maintaining centralized configuration information, naming, providing distributed synchronization, and providing group services in a simple interface so that we don't have to write it from scratch. Apache Kafka also uses ZooKeeper to manage configuration. ZooKeeper allows developers to focus on the core application logic, and it implements various protocols on the cluster so that the applications need not implement them on their own.

Zookeeper_intellipaat.

Data model and the hierarchical namespace

The name space provided by ZooKeeper is much like that of a standard file system. A name is a sequence of path elements separated by a slash (/). Every node in ZooKeeper's name space is identified by a path.

ZooKeeper's Hierarchical Namespace



Unlike is standard file systems, each node in a ZooKeeper namespace can have data associated with it as well as children. It is like having a file-system that allows a file to also be a directory. (ZooKeeper was designed to store coordination data: status information, configuration, location information, etc., so the data stored at each node is usually small, in the byte to kilobyte range.) We use the term *znode* to make it clear that we are talking about ZooKeeper data nodes.

Znodes maintain a stat structure that includes version numbers for data changes, ACL changes, and timestamps, to allow cache validations and coordinated updates. Each time a znode's data changes, the version number increases. For instance, whenever a client retrieves data it also receives the version of the data.

The data stored at each znode in a namespace is read and written atomically. Reads get all the data bytes associated with a znode and a write replaces all the data. Each node has an Access Control List (ACL) that restricts who can do what.

Distributed Coordination.

Before getting deeper into watches, let's establish some terminology. We talk about an *event* to denote the execution of an update to a given znode. A *watch* is a one-time trigger associated with a znode and a type of event (e.g., data is set in the znode, or the znode is deleted). When the watch is triggered by an event, it generates a *notification*. A notification is a message to the application client that registered the watch to inform this client of the event.

When an application process registers a watch to receive a notification, the watch is triggered at most once and upon the first event that matches the condition of the watch. For example, say that the client needs to know when a given znode `/z` is deleted (e.g., a backup master). The client executes an `exists` operation on `/z` with the watch flag set and waits for the notification. The notification comes in the form of a callback to the application client.

There are two types of watches: data watches and child watches. Creating, deleting, or setting the data of a znode successfully triggers a data watch. `exists` and `getData` both set data watches. Only `getChildren` sets child watches, which are triggered when a child znode is either created or deleted. For each event type, we have the following calls for setting a watch:

We use the same watch mechanism for notifying the application of events related to the state of a ZooKeeper session and events related to znode changes. Although session state changes and znode state changes constitute independent sets of events, we rely upon the same mechanism to deliver such events for simplicity.

State Change.

How to Use Apache ZooKeeper to Build Distributed Apps?

All the details mentioned above are done by the Zookeeper and the user does not have to do anything. The master is elected, the observers are set and the stage is made ready for the user to use the Zookeeper.

As compared earlier user can use Zookeeper like a file system where directories can be created and data can be stored inside it. The directories made above can also have children and grandchildren like any other file system. This file system is stored centrally thus giving access from any spot. Example of Apache Zookeeper can be a data model. Each directory in our example is called znode in Zookeeper. They are containers for data and other nodes. It stores statistical data like version details and user data up to 1Mb. This tiny space available to store information makes it clear that Zookeeper is not used for data storage like database but instead it is used for storing small amount of data like configuration data that needs to be shared.

There are 2 types of znodes:

- Persistent: This is the default type of znode in any Zookeeper. Persistent nodes are always present and they contain the important configuration details. When a new node is added to the Zookeeper it goes to persistent znode and gets the configuration information.
- Ephemeral: They are session nodes which gets created when an application fire ups and get deleted when the application has finished. This is mainly useful to keep check on client applications in case of failures. As the application fails the znode dies.

Zookeeper Tutorial.

80. The Curator RPC module using Apache Thrift protocol satisfies the claim element "a common monitoring protocol."

81. Coordination of software components/independent programs running independently on ever-changing multiple machines satisfies the claim element "monitoring being independent of a programming technology of said plurality of client applications and respective distributed components."

82. The clients connected to the zookeeper server satisfies the claim element "a plurality of client applications."

83. The zookeeper service enables the coordination of distributed applications with the help of curator RPC Module which provides a bridge to non-java environment. It uses an apache thrift protocol which supports large set of languages and environment thus zookeeper service can unify across languages and environments (herein referred as using a common monitoring protocol). The zookeeper service provides Coordination of software components/independent programs running independently on ever-changing multiple machines (herein inferred as monitoring being independent of a programming technology).

Announcing Curator RPC

The just release version 2.6.0 of Apache Curator includes a new project, Curator RPC.

The Curator RPC module implements a proxy that bridges non-java environments with the Curator framework and recipes. It uses Apache Thrift which supports a large set of languages and environments.

The benefits of Curator RPC are:

* Gives access to Curator to non JVM languages/environments
** Curator has become the de-facto JVM client library for ZooKeeper
** Curator makes using Apache ZooKeeper much easier
** Curator contains well-tested recipes for many common ZooKeeper usages
* Organizations can unify their ZooKeeper usage across languages/environments (i.e. use Curator's Service Discovery recipe)
* The quality of ZooKeeper clients for some non-JVM languages is lacking
* There are Thrift implementations for a large number of languages/environments

Curator RPC.

> Thrift is a lightweight, language-independent software stack with an associated code generation mechanism for RPC. Thrift provides clean abstractions for data transport, data serialization, and application level processing. Thrift was originally developed by Facebook and now it is open sourced as an Apache project. Apache Thrift is a set of code-generation tools that allows developers to build RPC clients and servers by just defining the data types and service interfaces in a simple definition file. Given this file as an input, code is generated to build RPC clients and servers that communicate seamlessly across programming languages.

> In this tutorial I will describe how Thrift works and provide a guide for build and installation steps, how to write thrift files and how to generate from those files the source code that can be used from different client libraries to communicate with the server. Thrift supports a variety of languages including C++, Java, Python, PHP, Ruby but for simplicity I will focus this tutorial on examples that include Java and Python.

Introduction to Thrift.

> The Service Discovery Server bridges non-Java or legacy applications with the Curator Service Discovery. It exposes RESTful web services to register, remove, query, etc. services.

Service Discovery Server.

Thrift is an interface definition language and binary communication protocol[1] used for defining and creating services for numerous languages.[2] It forms a remote procedure call (RPC) framework and was developed at Facebook for "scalable cross-language services development". It combines a software stack with a code generation engine to build cross-platform services which can connect applications written in a variety of languages and frameworks, including ActionScript, C, C++,[3] C#, Cappuccino,[4] Cocoa, Delphi, Erlang, Go, Haskell, Java, JavaScript, Objective-C, OCaml, Perl, PHP, Python, Ruby, Elixir[5], Rust, Smalltalk and Swift.[6] Although developed at Facebook, it is now an open source project in the Apache Software Foundation. The implementation was described in an April 2007 technical paper released by Facebook, now hosted on Apache.[7][8]

Apache Thrift.

There are multiple definitions of a *distributed system*, but for the purposes of this book, we define it as a system comprised of multiple software components running independently and concurrently across multiple physical machines. There are a number of reasons to design a system in a distributed manner. A distributed system is capable of exploiting the capacity of multiple processors by running components, perhaps replicated, in parallel. A system might be distributed geographically for strategic reasons, such as the presence of servers in multiple locations participating in a single application.

Having a separate coordination component has a couple of important advantages. First, it allows the component to be designed and implemented independently. Such an independent component can be shared across many applications. Second, it enables a system architect to reason more easily about the coordination aspect, which is not trivial (as this book tries to expose). Finally, it enables a system to run and manage the coordination component separately. Running such a component separately simplifies the task of solving issues in production.

ZooKeeper by Benjamin Reed.

Why Apache Zookeeper

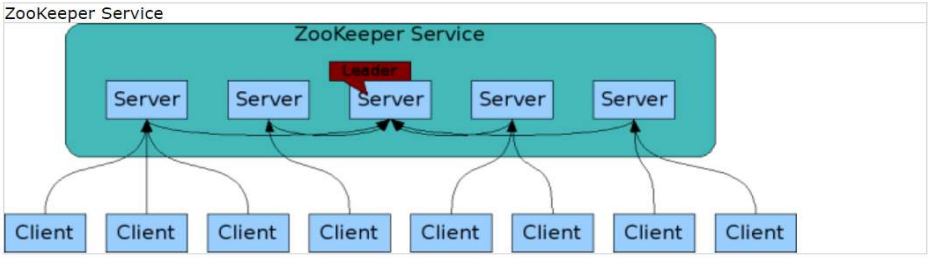
In the good old past, each application software was a single program running on a single computer with a single CPU. Today, things have changed. In the Big Data world, application softwares are made up of many independent programs running on an ever-changing set of computers. These applications are known as Distributed Application. A distributed application can run on multiple systems in a network simultaneously by coordinating among themselves to complete a particular task in a fast and efficient manner.

Building distributed systems is hard. Nowadays, a lot of the software applications people use daily, however, depend on such systems, and it doesn't look like we will stop relying on distributed computer systems any time soon. Coordinating the actions of the independent programs in a distributed systems is far more difficult than writing a single program to run on a single computer. It is easy for developers to get mired in coordination logic and lack the time to write their application logic properly or perhaps the converse, to spend little time with the coordination logic and simply to write a quick-and-dirty master coordinator that is fragile and becomes an unreliable single point of failure.

ZooKeeper was designed to be a robust service that enables application developers to focus mainly on their application logic rather than coordination. It exposes a simple API, inspired by the filesystem API, that allows developers to implement common coordination tasks, such as electing a master server, managing group membership, and managing metadata. ZooKeeper is an application library with two principal implementations of the APIs—Java and C—and a service component implemented in Java that runs on an ensemble of dedicated servers.

What is Zookeeper.

ZooKeeper is replicated. Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an ensemble.

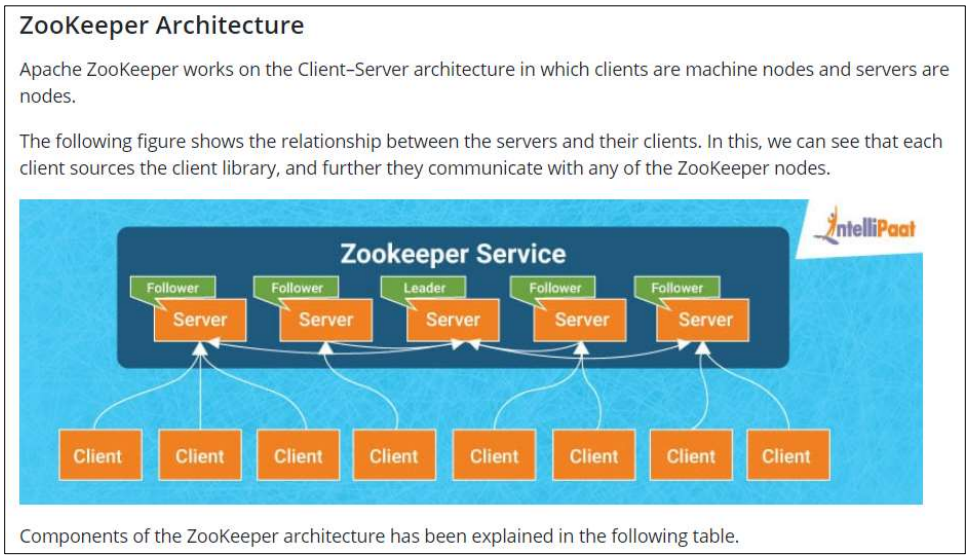


The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

Distributed Coordination.



Zookeeper_intellipaat.

84. The clients connected to the zookeeper server satisfies the claim element "a plurality of client applications."

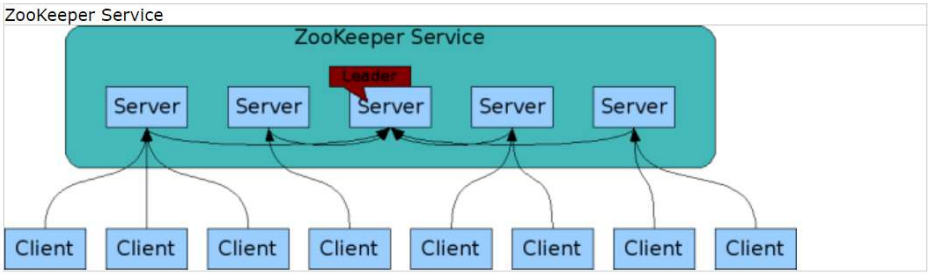
85. The children nodes such as z-nodes/data-nodes of applications can be created. The z-node is called as directory for storing data satisfies the claim element "plurality of client applications' distributed components."

86. The monitoring of health of session between client and zookeeper satisfies the claim element "health of a plurality of client applications."

87. The monitoring of health session of client with zookeeper is done using watcher interface. The Zookeeper monitoring metrices provides total/live clients connected to zookeeper server satisfies the claim element "assessing health."

88. The watcher interface in the zookeeper services watches/monitors the events related to health session of client with zookeeper and events related to z-nodes/data-nodes. The zookeeper monitoring metrices provides total/live clients connected to the zookeeper service in form of graphs.

ZooKeeper is replicated. Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an ensemble.



The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

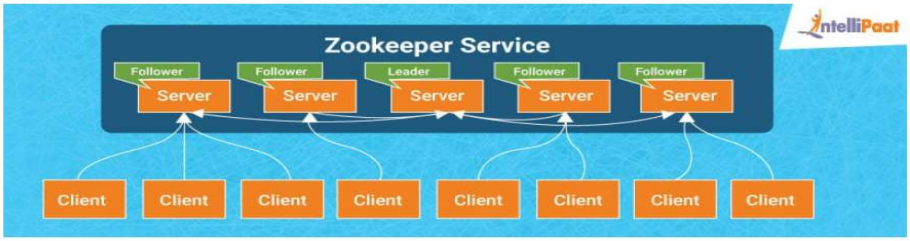
Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

Distributed Coordination.

ZooKeeper Architecture

Apache ZooKeeper works on the Client–Server architecture in which clients are machine nodes and servers are nodes.

The following figure shows the relationship between the servers and their clients. In this, we can see that each client sources the client library, and further they communicate with any of the ZooKeeper nodes.



Components of the ZooKeeper architecture has been explained in the following table.

Part	Description
Client	Client node in our distributed applications cluster is used to access information from the server. It sends a message to the server to let the server know that the client is alive, and if there is no response from the connected server the client automatically resends the message to another server.
Server	The server gives an acknowledgement to the client to inform that the server is alive, and it provides all services to clients.
Leader	If any of the server nodes is failed, this server node performs automatic recovery.
Follower	It is a server node which follows the instructions given by the leader.

Zookeeper_intellipaat.

in the previous chapters, we introduced the basic operation of zookeeper using the Zkcli tool. starting with this chapter, we'll see how the API is used in the app. Let's start by introducing How to use the Zookeeper API for development, showing how to create a session and implement a monitoring point (Watcher). We're still coding from the master-pattern example .

Watcher
an object that is used to receive session events, which requires our own creation. Because Wacher is defined as an interface, so we need to implement one ourselves, and then initialize the class's instance and passes the zookeeper in the constructor. The client uses the Watcher interface to monitor and the health of the session between zookeeper. Establish or lose connectivity with the zookeeper server event occurs. They can also be used to monitor changes in zookeeper data. In the end, such as The zookeeper session expires, and the event is passed through the Watcher interface to notify the client the application.

Zookeeper's API.

Another important issue with communication failures is the impact they have on synchronization primitives like locks. Because nodes can crash and systems are prone to network partitions, locks can be problematic: if a node crashes or gets partitioned away, the lock can prevent others from making progress. ZooKeeper consequently needs to implement mechanisms to deal with such scenarios. First, it enables clients to say that some data in the ZooKeeper state is *ephemeral*. Second, the ZooKeeper ensemble requires that clients periodically notify that they are alive. If a client fails to notify the ensemble in a timely manner, then all ephemeral state belonging to this client is deleted. Using these two mechanisms, we are able to prevent clients individually from bringing the application to a halt in the presence of crashes and communication failures.

ZooKeeper by Benjamin Reed.

`watcher`

An object we need to create that will receive session events. Because `Watcher` is an interface, we will need to implement a class and then instantiate it to pass an instance to the ZooKeeper constructor. Clients use the `Watcher` interface to monitor the health of the session with ZooKeeper. Events will be generated when a connection is established or lost to a ZooKeeper server. They can also be used to monitor changes to ZooKeeper data. Finally, if a session with ZooKeeper expires, an event is delivered through the `Watcher` interface to notify the client application.

Implementing a Watcher

To receive notifications from ZooKeeper, we need to implement watchers. Let's look a bit more closely at the `Watcher` interface. It has the following declaration:

ZooKeeper Flavio Junqueira.

The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

Distributed Coordination.

Apache ZooKeeper is a distributed hierarchical key-value store, which is used to provide a distributed configuration service, synchronization service, and naming registry for large distributed systems. Install and use our ZooKeeper monitoring tool and get detailed insights into system activity and health.

This document details how to configure the ZooKeeper plugin and the monitoring metrics for providing in-depth visibility into the performance, availability, and usage stats of ZooKeeper servers.

Maximum connections

Use the metric "maxclientcnxns" and get the total number of concurrent connections that a single client, identified by IP address, may make to a single member of the ZooKeeper system.

Connections

The metric "connections" lists total number of connection/session details for all clients connected to the ZooKeeper server.

ZooKeeper Monitoring.

Overview

The Zookeeper check tracks client connections and latencies, monitors the number of unprocessed requests, and more.

zookeeper.num_alive_connections (gauge)	The total count of client connections. *Shown as connection*

DataDog.

Before getting deeper into watches, let's establish some terminology. We talk about an *event* to denote the execution of an update to a given znode. A *watch* is a one-time trigger associated with a znode and a type of event (e.g., data is set in the znode, or the znode is deleted). When the watch is triggered by an event, it generates a *notification*. A notification is a message to the application client that registered the watch to inform this client of the event.

When an application process registers a watch to receive a notification, the watch is triggered at most once and upon the first event that matches the condition of the watch. For example, say that the client needs to know when a given znode `/z` is deleted (e.g., a backup master). The client executes an `exists` operation on `/z` with the watch flag set and waits for the notification. The notification comes in the form of a callback to the application client.

State Change.

89. The clients connected to the zookeeper server satisfies the claim element "a plurality of client applications."

90. The children nodes such as z-nodes/data-nodes of applications can be created. The z-node is called as directory for storing data satisfies the claim element "plurality of client applications' distributed components."

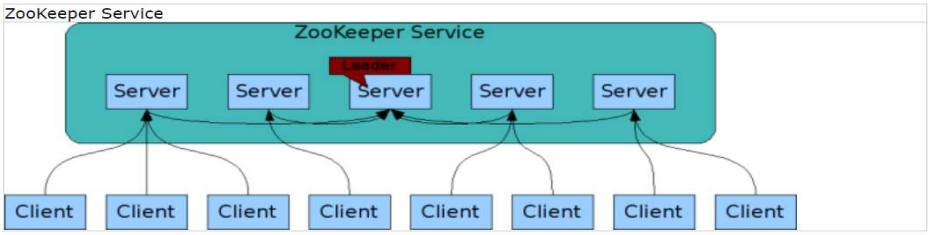
91. The monitoring of health of session between client and zookeeper satisfies the claim element "health of a plurality of client applications."

92. Event Notification by watcher of client's session with zookeeper and of deletion and creation of z-nodes/data-nodes (herein referred as respective distributed components) satisfies the claim element "associating said health."

93. The Leader Node satisfies the claim element "a single application node."

94. The zookeeper service provides the event notification regarding the health of session between client and zookeeper and creation/deletion of z-nodes/data-nodes through watcher (herein inferred as associating said health). When the session between client and zookeeper expires or disconnected there is a change in the state of client. All the state changes are accepted from clients are accepted by leader and replicated to the follower node.

ZooKeeper is replicated. Like the distributed processes it coordinates, ZooKeeper itself is intended to be replicated over a sets of hosts called an ensemble.



The servers that make up the ZooKeeper service must all know about each other. They maintain an in-memory image of state, along with a transaction logs and snapshots in a persistent store. As long as a majority of the servers are available, the ZooKeeper service will be available.

ZooKeeper is a distributed, open-source coordination service for distributed applications. It exposes a simple set of primitives that distributed applications can build upon to implement higher level services for synchronization, configuration maintenance, and groups and naming. It is designed to be easy to program to, and uses a data model styled after the familiar directory tree structure of file systems. It runs in Java and has bindings for both Java and C.

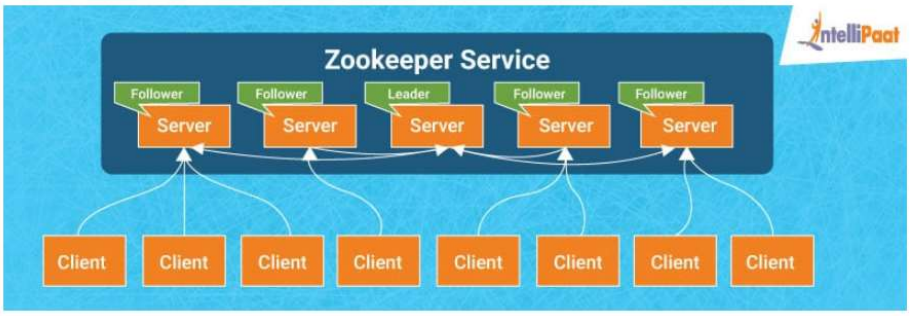
Clients connect to a single ZooKeeper server. The client maintains a TCP connection through which it sends requests, gets responses, gets watch events, and sends heart beats. If the TCP connection to the server breaks, the client will connect to a different server.

Distributed Coordination.

ZooKeeper Architecture

Apache ZooKeeper works on the Client–Server architecture in which clients are machine nodes and servers are nodes.

The following figure shows the relationship between the servers and their clients. In this, we can see that each client sources the client library, and further they communicate with any of the ZooKeeper nodes.



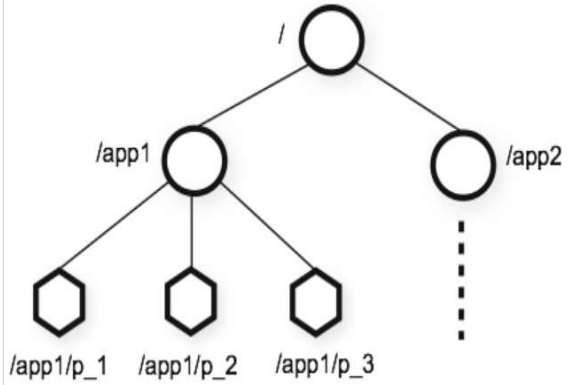
Components of the ZooKeeper architecture has been explained in the following table.

Zookeeper_intellipaat.

Data model and the hierarchical namespace

The name space provided by ZooKeeper is much like that of a standard file system. A name is a sequence of path elements separated by a slash (/). Every node in ZooKeeper's name space is identified by a path.

ZooKeeper's Hierarchical Namespace



Unlike is standard file systems, each node in a ZooKeeper namespace can have data associated with it as well as children. It is like having a file-system that allows a file to also be a directory. (ZooKeeper was designed to store coordination data: status information, configuration, location information, etc., so the data stored at each node is usually small, in the byte to kilobyte range.) We use the term *znode* to make it clear that we are talking about ZooKeeper data nodes.

Znodes maintain a stat structure that includes version numbers for data changes, ACL changes, and timestamps, to allow cache validations and coordinated updates. Each time a znode's data changes, the version number increases. For instance, whenever a client retrieves data it also receives the version of the data.

The data stored at each znode in a namespace is read and written atomically. Reads get all the data bytes associated with a znode and a write replaces all the data. Each node has an Access Control List (ACL) that restricts who can do what.

Distributed Coordination.

Watcher

an object that is used to receive session events, which requires our own creation. Because Wacher is defined as an interface, so we need to implement one ourselves, and then initialize the class's instance and passes the zookeeper in the constructor. The client uses the Watcher interface to monitor and the health of the session between zookeeper. Establish or lose connectivity with the zookeeper server event occurs. They can also be used to monitor changes in zookeeper data. In the end, such as The zookeeper session expires, and the event is passed through the Watcher interface to notify the client the application.

Zookeeper's API.

> Another important issue with communication failures is the impact they have on synchronization primitives like locks. Because nodes can crash and systems are prone to network partitions, locks can be problematic: if a node crashes or gets partitioned away, the lock can prevent others from making progress. ZooKeeper consequently needs to implement mechanisms to deal with such scenarios. First, it enables clients to say that some data in the ZooKeeper state is *ephemeral*. Second, the ZooKeeper ensemble requires that clients periodically notify that they are alive. If a client fails to notify the ensemble in a timely manner, then all ephemeral state belonging to this client is deleted. Using these two mechanisms, we are able to prevent clients individually from bringing the application to a halt in the presence of crashes and communication failures.

ZooKeeper by Benjamin Reed.

> Along with this strong consistency guarantee, ZooKeeper also promises high availability, which can loosely be interpreted to mean that it can withstand a significant number of machine failures before the service stops being available to clients. ZooKeeper achieves this availability in a traditional way - by replicating the data that is being written and read amongst a small number of machines so that if one fails, there are others ready to take over without the client being any wiser.

ZooKeeper Scale.

> Before getting deeper into watches, let's establish some terminology. We talk about an *event* to denote the execution of an update to a given znode. A *watch* is a one-time trigger associated with a znode and a type of event (e.g., data is set in the znode, or the znode is deleted). When the watch is triggered by an event, it generates a *notification*. A notification is a message to the application client that registered the watch to inform this client of the event.

> When an application process registers a watch to receive a notification, the watch is triggered at most once and upon the first event that matches the condition of the watch. For example, say that the client needs to know when a given znode `/z` is deleted (e.g., a backup master). The client executes an `exists` operation on `/z` with the watch flag set and waits for the notification. The notification comes in the form of a callback to the application client.

> There are two types of watches: data watches and child watches. Creating, deleting, or setting the data of a znode successfully triggers a data watch. `exists` and `getData` both set data watches. Only `getChildren` sets child watches, which are triggered when a child znode is either created or deleted. For each event type, we have the following calls for setting a watch:

> We use the same watch mechanism for notifying the application of events related to the state of a ZooKeeper session and events related to znode changes. Although session state changes and znode state changes constitute independent sets of events, we rely upon the same mechanism to deliver such events for simplicity.

State Change.

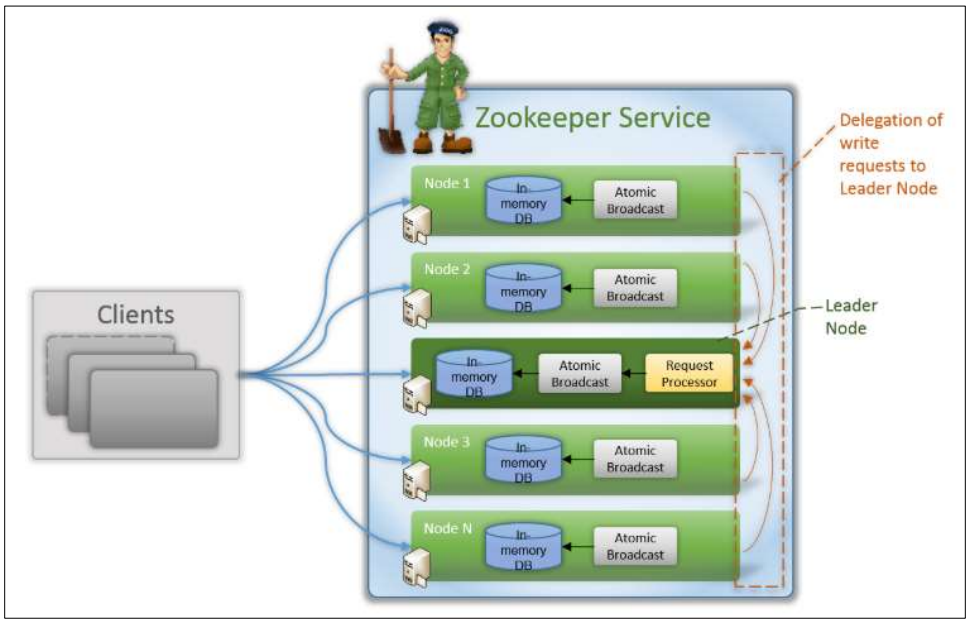
> Another parameter to the ZooKeeper session establishment call is the default watcher. Watchers are notified when any state change occurs in the client. For example if the client loses connectivity to the server the client will be notified, or if the client's session expires, etc... This watcher should consider the initial state to be disconnected (i.e. before any state changes events are sent to the watcher by the client lib). In the case of a new connection, the first event sent to the watcher is typically the session connection event.

Programmer's Guide.

- leader and followers- in ZooKeeper cluster, one of the nodes has a leader role and the rest have followers roles. The leader is responsible for accepting all incoming state changes from the clients and replicate them to itself and to the followers. read requests are load balanced between all followers and leader.

- ZooKeeper uses a variation of two-phase-commit protocol for replicating transactions to followers. When a leader receive a change update from a client it generate a transaction with sequel number *c* and the leader's epoch *e* (see definitions) and send the transaction to all followers. a follower adds the transaction to its history queue and send ACK to the leader. When a leader receives

Architecture of ZAB.



Introduction.

95. BCS has been damaged by Defendant's infringement of the '809 Patent.

PRAYER FOR RELIEF

WHEREFORE, BCS respectfully requests the Court enter judgment against Defendant:

1. declaring that the Defendant has infringed the '809 Patent;

2. awarding BCS its damages suffered as a result of Defendant's infringement of the '809 Patent;

3. awarding BCS its costs, attorneys' fees, expenses, and interest; and

4. granting BCS such further relief as the Court finds appropriate.

JURY DEMAND

BCS demands trial by jury, Under Fed. R. Civ. P. 38.

Dated: July 29, 2020

Respectfully Submitted

/s/ Raymond W. Mort, III

Raymond W. Mort, III
Texas State Bar No. 00791308
raymort@austinlaw.com

THE MORT LAW FIRM, PLLC
100 Congress Ave, Suite 2000
Austin, Texas 78701
Tel/Fax: (512) 865-7950

ATTORNEYS FOR PLAINTIFF