Understanding Real-World Timeout Problems in Cloud Server Systems

Ting Dai, Jingzhu He, Xiaohui Gu, Shan Lu[†]
North Carolina State University, {*tdai, jhe16, xgu*}*@ncsu.edu*
[†]University of Chicago, *shanlu@cs.uchicago.edu*

***Abstract*—Timeouts are commonly used to handle unexpected failures in distributed systems. In this paper, we conduct a comprehensive study to characterize real-world timeout problems in 11 commonly used cloud server systems (e.g., Hadoop, HDSF, Spark, Cassandra, etc.). Our study reveals timeout problems are widespread among cloud server systems. We categorize those timeout problems in three aspects: 1) what are the root causes of those timeout problems? 2) what impact can timeout problems impose to cloud systems? 3) how are timeout problems currently diagnosed or misdiagnosed?**

Our results show that root causes of timeout problems include misused timeout, missing timeout, improper timeout handling, unnecessary timeout, and clock drifting. We further find timeout bugs impose serious impact (e.g., system hang or crash, job failure, performance degradation, data loss) to both applications and systems. Our study also shows that 60% of the bugs do not produce any error messages and 12% bugs produce misleading error messages, which makes it difficult to diagnose those timeout bugs.

I. Introduction

Cloud server systems (e.g., Hadoop, Cassandra, HDFS, Spark) have become increasingly complex, which often consist of many inter-dependent components. It is challenging to achieve reliability in cloud server applications because 1) different components need to communicate frequently with each other via unreliable networks and 2) individual component may fail at any time. Timeout is one of the commonly used mechanisms to handle unexpected failures in distributed computing environments [1], [2], [3], [4], [5]. Timeout mechanism can be used in both intra-node and inter-node communication failover. For example, when a component C_1 sends a request to another component C_2, C_1 sets a timer timeout value and waits for the response from C_2 until the timer expires. In case C_2 fails or a message loss occurs, C_1 can break out of the waiting state triggered by the timeout event and take proper actions (e.g., retrying or skipping) accordingly.

However, many real-world cloud server applications lack proper configuration and handling of those timeout events. As the scale of server applications grow, the likelihood of timeout bugs also increases. In 2015, Amazon DynamoDB service was down for five hours [6]. The service outage is caused by a timeout bug in the metadata server. When the metadata server was already overloaded, the new requests from storage servers to the metadata server failed due to timeout. Storage servers kept retrying, causing further failures and retries, creating a cascading failure.

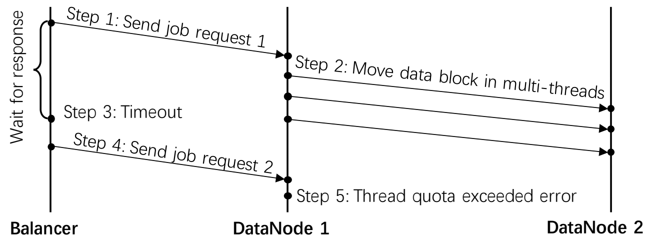
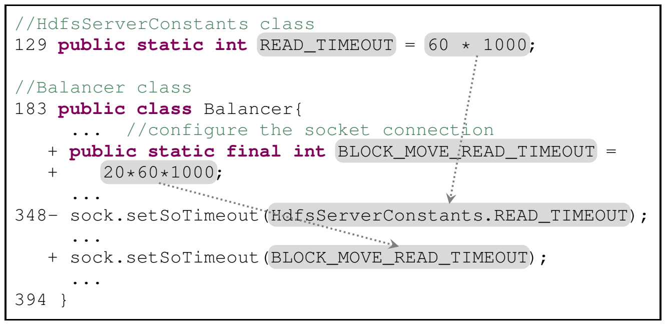

Wait for response
Step 1: Send job request 1
Step 2: Move data block in multi-threads
Step 3: Timeout
Step 4: Send job request 2
Step 5: Thread quota exceeded error
Balancer
DataNode 1
DataNode 2

Figure 1. The HDFS-6166 timeout bug. The DataNode 1 reports "thread quota exceeded" exception after the bug is triggered. The root cause of this bug is the misconfigured timeout value for socket connection between the Balancer and the DataNode 1.



```
//HdfsServerConstants class
129 public static int READ_TIMEOUT = 60 * 1000;

//Balancer class
183 public class Balancer{
      ...  //configure the socket connection
    + public static final int BLOCK_MOVE_READ_TIMEOUT =
    +    20*60*1000;
      ...
348- sock.setSoTimeout(HdfsServerConstants.READ_TIMEOUT);
      ...
    + sock.setSoTimeout(BLOCK_MOVE_READ_TIMEOUT);
      ...
394 }
```

Figure 2. The patch of the HDFS-6166 bug. The ┈┈➤ represents how the `Balancer` makes the configurable timeout changed from 1 min to 20 mins.

A. *A Motivating Example*

To better understand how real-world timeout bugs happen, and how they can affect cloud servers and applications, we use the HDFS-6166 [1] bug as one example shown by Figure 1 and 2. This bug is caused by misconfiguring a timeout value for the socket connection between the Balancer node and the DataNode. In Hadoop Distributed File Systems (HDFS), DataNodes store the data in a distributed manner and the Balancer schedules data block movement among the DataNodes. In this bug example, the Balancer sends a data moving request to DataNode 1. Then DataNode 1 performs data moving to DataNode 2. After completing the data movement, DataNode 1 sends a reply message to the Balancer. As shown in Figure 2, the developer configured a timeout checking (line 348 of `Balancer` class) for the connection between the Balancer and DataNode 1. In the buggy version, the timeout value was set to `HdfsServerConstants.READ_TIMEOUT` which is

[1]We use "system name-bug #" to denote different bugs.

Table I
THE STATISTICS OF TIMEOUT BUGS IN EACH SYSTEM WE STUDIED.

System	Description	Num of bugs
Cassandra	Distributed database management system	17
Flume	Distributed streaming service	13
Hadoop Common	Hadoop utilities and libraries	15
Hadoop Mapreduce	Hadoop big data processing framework	15
Hadoop Yarn	Hadoop resource management platform	4
HDFS	Hadoop distributed file system	26
HBase	Non-relational, distributed database	28
Phoenix	Distributed database engine	6
Qpid	Messaging service	20
Spark	Big data computation framework	4
Zookeeper	Synchronization service	8
Total		156

one minute. If the data migration between the Balancer and DataNode 1 needs more than one minute, the connection keeps timeout and the Balancer node resends the same data migration request again and again.

When this bug occurs, cloud applications using HDFS experience continuous execution failures. Since the system reports many misleading errors such as "threads quota exceeded" errors, it is hard for developers to find the root cause of the job failures is actually related to timeout value misconfiguration. After figuring out the root cause, the developer further needs to set a proper timeout value for the data movement operations to fix the bug, which is another challenging problem. If the timeout value is set to be too small, the system still experiences many data migration failures as in this buggy version; If the timeout value is set to be too large, it causes unnecessary system idle time, which can further cause application performance degradation. In the patch, the timeout value is increased to 20 minutes in order to tolerant bad network condition, which may not be ideal for other network conditions.

B. Our Contribution

In this paper, we perform a comprehensive characteristic study about 156 real-world timeout bugs in 11 popular cloud server applications listed in Table I. We characterize those bugs in three key aspects: 1) what are the root causes of those timeout bugs? 2) what impact can timeout bugs impose to cloud servers and applications? 3) how are timeout bugs currently diagnosed or misdiagnosed?

Our study has made the following findings:

- **Timeout bug root causes:** Our study shows that real-world timeout problems are caused by a variety of reasons including 1) *misused timeout value* where a timeout variable is misconfigured, ignored or incorrectly reused; 2) *missing timeout checking* where inter-component communication lacks timeout protection; 3) *improper timeout handling* where a timeout event is handled by inappropriate retries or aborts; 4) *unnecessary timeout* where a timeout is used for a function call which does not need timeout protection; and 5) *clock drifting* where timeout problems are caused by asynchronous clocks between distributed hosts. The misused timeout value is the top root cause which is attributed to 47% bugs followed by the missing timeout root cause attributing to 31% bugs we studied.
- **Timeout bug impact:** Timeout bugs impose serious impact to both cloud servers and cloud applications. Among 156 timeout bugs we studied, 40% timeout bugs cause the whole or part of servers to become unavailable due to software hang or crash, 33% timeout bugs cause application job execution failures, 26% timeout bugs cause significant performance degradations for cloud applications, and 2% timeout bugs cause serious data loss.
- **Timeout bug diagnosis:** Our study shows that real-world timeout bugs are difficult to diagnose: 60% timeout bugs do not produce any error message and 12% timeout bugs produce misleading error messages.

The rest of the paper is organized as follows. Section II describes our bug study methodology. Section III provides root cause analysis over all the identified timeout bugs. Section IV describes the impact of timeout bugs to both systems and applications. Section V evaluates the diagnosability of all studied timeout bugs. Section VI discusses related work. Finally, the paper concludes in Section VII.

II. METHODOLOGY

In this section, we present our bug collection and analysis methodology. First, we introduce our target cloud server systems. Second, we describe how we collect real-world timeout bugs from those studied systems. Third, we present the key characteristics we consider for classifying different timeout bugs.

A. Target Cloud Server Systems

Our study covers a range of popular open source cloud server systems listed in Table I: Cassandra and HBase are distributed key-value stores; Flume is a distributed streaming system; Hadoop MapReduce and Spark are big data processing platforms; Hadoop Yarn is a distributed resource management service; HDFS is a distributed file system; Phoenix is a distributed database engine; Qpid is a distributed messaging service; and Zookeeper is a distributed synchronization service. We try to cover as many cloud server systems as possible to show that timeout bugs are widespread among them. Moreover, those systems are implemented in different programming languages including Java, C/C++, Python, and Scala.

B. Timeout Bug Collection

Our benchmark includes 156 distinct real-world timeout bugs, shown in Table I. They are collected from commonly used bug repositories (e.g., Apache JIRA [7]), using the following criteria: 1) issue type: `Bug`, `Improvement`, `New Feature`; 2) status: `RESOLVED`, `CLOSED`, `PATCH AVAILABLE`; and 3) keyword: `timeout`. We then manually examined each bug, eliminating about 5000 "will not fix", "duplicated" and "not a problem" cases.

C. Characteristic Categories

We classified all timeout bugs in regard to three characteristics: root causes, impact to systems and/or applications, and diagnosability. Some characteristics are subdivided into several types and the percentage of the corresponding samples in each type is also reported.

Our study found that timeout bugs have a large variety of root causes. A clear understanding of how timeout bugs occur helps developers to avoid them. Specifically, we classify all kinds of timeout bug root causes into the following categories:

1) *misused timeout value* represents those issues where timeout values are configured improperly. We further divide this category into six subcategories: 1.a) misconfigured timeout value where the timeout value is set to a wrong one; 1.b) ignored timeout value where the configured timeout value in the configuration file is not passed into the program; 1.c) incorrectly reused timeout value where a timeout variable is reused by more than one timeout checking function; 1.d) inconsistent timeout value where two different values are assigned to one timeout variable; 1.e) stale timeout value where timeout value is not updated during the execution; and 1.f) improper timeout scope where a timeout variable is set at the wrong location of the program.
2) *missing timeout checking* represents those issues caused by lacking necessary timeout detection and handling. We further divide this category into two subcategories: 2.a) missing timeout for network communication, and 2.b) missing timeout for synchronization.
3) *unnecessary timeout protection* represents those issues that are caused by unneeded timeout checking.
4) *improper timeout handling* represents those issues which are caused by wrong handling operations for timeouts. We further divide this category into five subcategories: 4.a) insufficient retries, 4.b) excessive retries, 4.c) incorrect retry, 4.d) incomplete abort, and 4.e) incorrect abort.
5) *clock drifting* represents those issues where a timeout occurs too soon or too late due to asynchronous clock in distributed systems.

Our study found that timeout bugs can impose serious impact to both server systems and applications. We classify the impact of timeout bugs into the following categories:

1) *system unavailability impact* where timeout bugs make the whole or part of the server system unavailable to clients;
2) *job failure impact* where timeout bugs cause application request failures;
3) *performance degradation impact* where timeout bugs cause prolonged delay in application job execution;
4) *data loss impact* where data loss occurs due to timeout bugs.

Because developers still heavily rely on log messages to debug, diagnosability of timeout bugs depends on whether correct and relevant log messages are produced when those timeout bugs are triggered. So our evaluation on the diagnosability of timeout bugs is based on the types of log messages produced by the buggy code:

1) *correct error message* represents the case where the server system reports a correct and relevant error message when a timeout problem occurs;
2) *no error message* represents the case where the system does not produce any error message when a timeout problem occurs;
3) *wrong error message* represents the case where the server system produces a misleading or irrelevant error message when a timeout problem occurs.

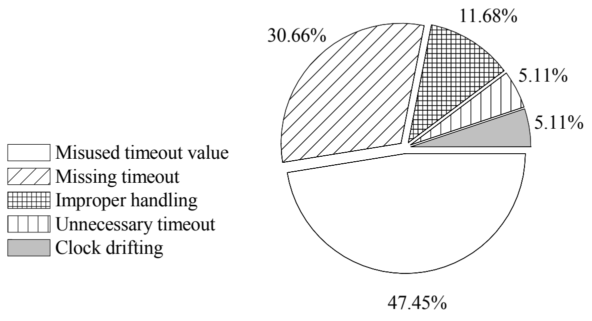
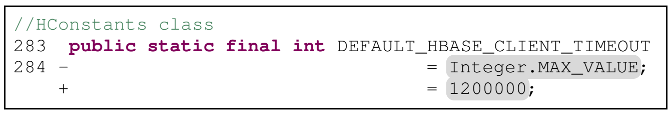


Figure 3. The statistics of root causes of timeout problems.



```
//HConstants class
283  public static final int DEFAULT_HBASE_CLIENT_TIMEOUT
284 -                                   = Integer.MAX_VALUE;
    +                                   = 1200000;
```

Figure 4. The patch for the HBase-13647 bug. An example of misconfigured timeout value.

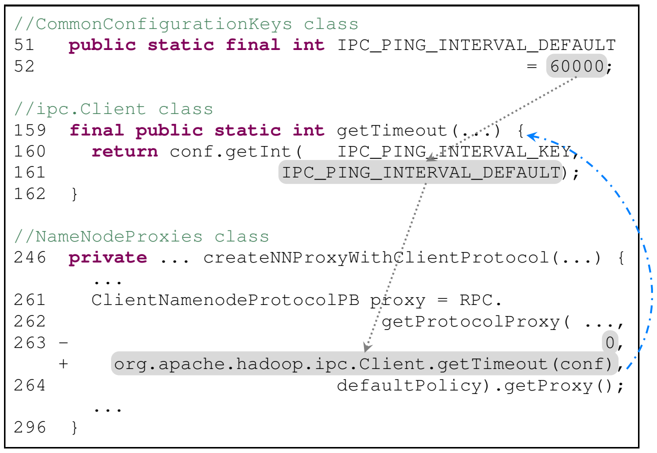
III. Root Cause Analysis

In this section, we present a detailed analysis of the root causes of the 156 real-world timeout problems we studied. We characterize them into five categories. As shown in Figure 3, we found 47% timeout bugs are caused by misused timeout value, 31% timeout problems are caused by missing timeout checking, 12% are caused by the improper timeout handling, 5% are caused by unnecessary timeout, and 5% are caused by clock drifting. We now discuss each category in detail along with examples.

A. Misused timeout value

In our study, we found 65 out of 156 timeout bugs are caused by misused timeout values. To better understand how those timeout values are misused, we further characterize them into six subcategories: misconfigured timeout value, ignored timeout value, incorrectly reused timeout value, using inconsistent timeout value, using stale timeout value, and improper timeout scope.

Misconfigured timeout values: We found 38 bugs are caused by misconfigured timeout values, that is, timeout variables are assigned with wrong values. Timeout value misconfigurations are often caused by software developers'



```
//CommonConfigurationKeys class
51   public static final int IPC_PING_INTERVAL_DEFAULT
52                                              = 60000;

//ipc.Client class
159  final public static int getTimeout(...) {
160    return conf.getInt(   IPC_PING_INTERVAL_KEY,
161                         IPC_PING_INTERVAL_DEFAULT);
162  }

//NameNodeProxies class
246  private ... createNNProxyWithClientProtocol(...) {
       ...
261    ClientNamenodeProtocolPB proxy = RPC.
262                               getProtocolProxy( ...,
263 -                                                 0,
    +    org.apache.hadoop.ipc.Client.getTimeout(conf),
264                                defaultPolicy).getProxy();
       ...
296  }
```

Figure 5. The patch for the HDFS-4646 bug. An example of ignored timeout value. The ⋯⋯→ represents how the configurable timeout variable flows into `createNNProxyWithClientProtocol` function, and the -·-→ represents the function call flows.

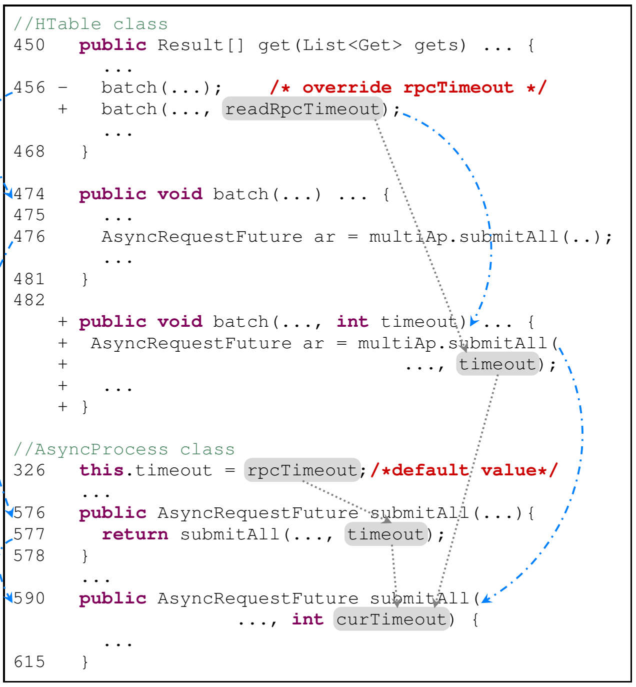
lack of understanding about the timeout mechanism. Some mature systems, such as Hadoop and Spark, provide manuals to users with commonly used timeout values [8], [9], [10], [11], [12]. However, misconfigured timeout values are still the top root cause of those timeout bugs in cloud server systems.

Let us use the HBase-13647 bug as an example to illustrate the misconfigured timeout value root cause. As shown in Figure 4, `Integer.MAX_VALUE` is the default value of the timeout variable `hbase.client.operation.timeout` used on the connection between a HBase client and a HBase server. When the HBase server (e.g., HMaster, RegionServer) fails, HBase client hangs for about 24 days. In the patch, the timeout value is changed to 20 minutes, because HDFS takes about 10 minutes to recover from a failure. We found several cases which set the timeout value to `Integer.MAX_VALUE` or `Long.MAX_VALUE`, including the HBase-6684 and the HBase-13647 bugs.

The HBase-3273 bug is another example of misconfiguring a timeout value for `zookeeper.session.timeout`. The timeout value is originally set to one minute. When the Zookeeper server experiences a big garbage collection pause, which lasts for more than one minute, all the sessions expire during the process. It leads to a system failure, which further causes application failures. To allow the big garbage collection pause, the timeout value is increased to three minutes.

Ignored timeout value: We found ten timeout bugs where a correctly configured timeout value is always ignored. Normally, the timeout variables are configured with default or user-specified values read from the configuration files (e.g., `Constants` class, `conf.xml` file). Systems often provide APIs to utilize the pre-configured timeout variables. Without understanding how and when to invoke those APIs correctly, software developers use a hard-coded timeout value by mistake.

For example, in the HDFS-4646 bug, shown by Figure 5, when HDFS creates a NameNode proxy by the RPC call, i.e., `createNNProxyWithClientProtocol`, the configured timeout value (`IPC_PING_INTERVAL_DEFAULT`) for the RPC call is ignored and a wrong timeout value (0) is passed in (line 263 in `RPC.getProtocolProxy` function). As a result, timeout does not work as expected because the value is set to be 0.



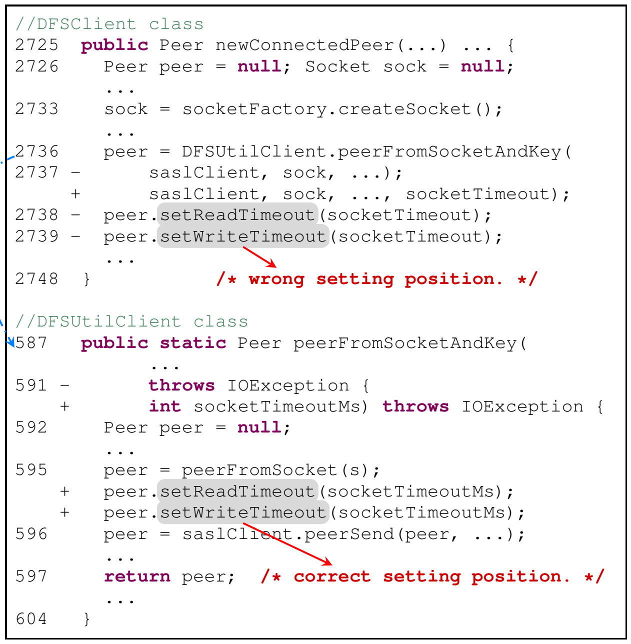
```
//HTable class
450  public Result[] get(List<Get> gets) ... {
       ...
456 -    batch(...);    /* override rpcTimeout */
    +    batch(..., readRpcTimeout);
       ...
468  }

474  public void batch(...) ... {
475    ...
476    AsyncRequestFuture ar = multiAp.submitAll(..);
       ...
481  }
482
    + public void batch(..., int timeout) ... {
    +  AsyncRequestFuture ar = multiAp.submitAll(
    +                               ..., timeout);
    +    ...
    + }

//AsyncProcess class
326  this.timeout = rpcTimeout;/*default value*/
     ...
576  public AsyncRequestFuture submitAll(...){
577    return submitAll(..., timeout);
578  }
     ...
590  public AsyncRequestFuture submitAll(
                   ..., int curTimeout) {
       ...
615  }
```

Figure 6. The call graph and partial patch for the HBase-16556 bug. To save space, we emit the patched lines for `delete`, `existsAll` functions. This is an example of incorrectly reused timeout value. The ⋯⋯→ represents how `rpcTimeout` flows into the second `submitAll` function, while the -·-→ represents the function call flows.

Incorrectly reused timeout value: Different operations in one system can have various triggering time and execution time. Thus, they often need different timeout variables to enforce their occurrences, executions, and terminations. Incorrectly reused timeout value bugs refer to those cases where one timeout variable is reused for different operations by mistake. We found eight bugs in this category.

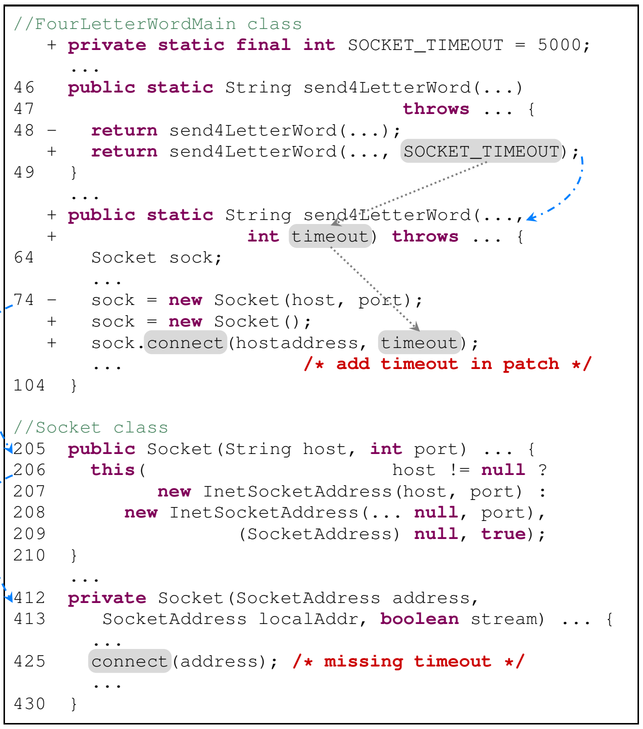
For example, the HBase-16556 bug shows how reusing the `rpcTimeout` in three different cases cause system performance degradation. The patch for the HBase-16556 bug is shown in Figure 6. In HBase, the `get`, `delete` and `existsAll` functions in `HTable` class call `batch` function to submit corresponding requests on line 456. The `batch` function in turn calls `submitAll` function on line 590 in `AsyncProcess` class. Without passing the timeout argument in the `batch` function, `submitAll` reuses the default timeout variable, `rpcTimeout`, for all requests. In fact, `rpcTimeout` is too large for `get` and `exitsAll` but too small for `delete`. Too long timeout degrades the application's performance. Too short timeout causes the `delete` operation to fail, which leads to the application failure.

Inconsistent timeout value: This category of timeout



```
//DFSClient class
2725  public Peer newConnectedPeer(...) ... {
2726    Peer peer = null; Socket sock = null;
        ...
2733    sock = socketFactory.createSocket();
        ...
2736    peer = DFSUtilClient.peerFromSocketAndKey(
2737 -        saslClient, sock, ...);
     +        saslClient, sock, ..., socketTimeout);
2738 -  peer.setReadTimeout(socketTimeout);
2739 -  peer.setWriteTimeout(socketTimeout);
        ...
2748  }            /* wrong setting position. */

//DFSUtilClient class
587   public static Peer peerFromSocketAndKey(
            ...
591  -      throws IOException {
     +      int socketTimeoutMs) throws IOException {
592     Peer peer = null;
        ...
595     peer = peerFromSocket(s);
     +  peer.setReadTimeout(socketTimeoutMs);
     +  peer.setWriteTimeout(socketTimeoutMs);
596     peer = saslClient.peerSend(peer, ...);
        ...
597     return peer;  /* correct setting position. */
        ...
604   }
```

Figure 7. The patch for the HDFS-10223 bug. An example of improper timeout scope. The -·-> represents the function call flows.



```
//FourLetterWordMain class
  + private static final int SOCKET_TIMEOUT = 5000;
    ...
    public static String send4LetterWord(...)
                                  throws ... {
  -   return send4LetterWord(...);
  +   return send4LetterWord(..., SOCKET_TIMEOUT);
    }
    ...
  + public static String send4LetterWord(...,
  +                     int timeout) throws ... {
      Socket sock;
      ...
  -   sock = new Socket(host, port);
  +   sock = new Socket();
  +   sock.connect(hostaddress, timeout);
      ...                       /* add timeout in patch */
    }

//Socket class
    public Socket(String host, int port) ... {
      this(                        host != null ?
            new InetSocketAddress(host, port) :
          new InetSocketAddress(... null, port),
                   (SocketAddress) null, true);
    }
    ...
    private Socket(SocketAddress address,
       SocketAddress localAddr, boolean stream) ... {
      ...
      connect(address); /* missing timeout */
      ...
    }
```

Figure 8. The call graph and patch for the Zookeeper-2224 bug. An example of missing timeout for network communication. The ·····> represent how SOCKET_TIMEOUT flows and be used for Socket connection, and the -·-> represents the function call flows.

bugs are caused by inconsistent settings to the same timeout variable. We found four bugs in this category. The Hadoop-11488 bug gives an example of setting two different values for `fs.s3a.connection.timeout` variable. The default value of `fs.s3a.connection.timeout` in `core-default.xml` is 5000, while the default value in the `Constants` class is 50000.

Stale timeout value: At different stages of an application execution, the timeout value needs to be updated to meet different execution requirements. A commonly seen mistake is to use a stale value. It leads to job failure or undesirable delay during the application's execution. We found three bugs in this category. The Zookeeper-593 bug gives an example of how not updating timeout value leads to a system failure. A client does not have access to the latest `sessionTimeout` value when it is updated by the server. Because of the staleness, the timeout value at the client side is different from the timeout value at the server side. The side that holds a smaller timeout value might terminate the session earlier, which leads to session failures.

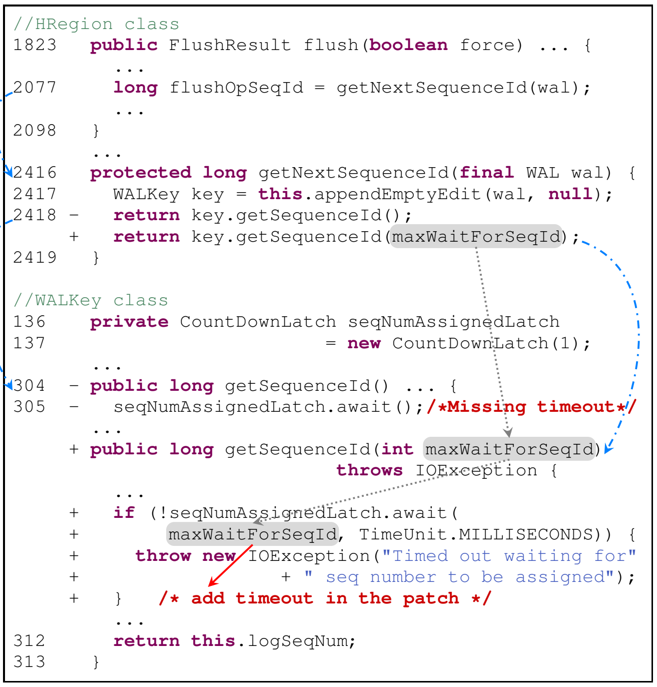
Improper timeout scope: In this category, timeout bugs are caused by being set at wrong locations. We found two bugs in this category. For example, Figure 7 shows the patch for the HDFS-10223 bug. The timeout mechanism is originally configured after `peer` is instantiated, that is on line 2738 and 2739. However, the `saslClient.peersend` method already sets a default system-wide TCP timeout for the connections, which makes all subsequent timeout settings invalid. The default system-wide TCP timeout is usually several hours long, causing undesirable delay for the application. In the patch, developers configure the timeout before `saslClient.peersend` operation in the `peerFromSocketAndKey` class.

Observations: Misused timeout value bug often occur when software developers do not perform extensive testing (e.g., unit testing, regression testing, and stress testing) on timeout configurations before releasing; or the software developers do not have a clear understanding about the system's timeout mechanisms [13]. Moreover, setting proper timeout value is challenging. Too small timeout values can cause system or application to fail while too large timeout values can cause performance degradation.

B. *Missing timeout checking*

We found 42 timeout bugs are caused by missing timeout checking. In distributed systems, different nodes interact with each other using either network communications or shared variables. Thus, we group missing timeout bugs into two groups: 1) missing timeout checking for network communications; and 2) missing timeout checking for intra-node synchronizations. Missing timeout checking can cause system hang or crash, which often does not produce any timeout related error messages.

Missing timeout for network communication: We found 26 bugs are caused by missing timeout checking for network communications, such as socket connections (including read and write), RPC calls, and HTTP requests. Figure 8 shows the Zookeeper-2224 bug, which gives an example of how missing timeout checking for socket connections software hang in Zookeeper. An application sends a request to the Zookeeper



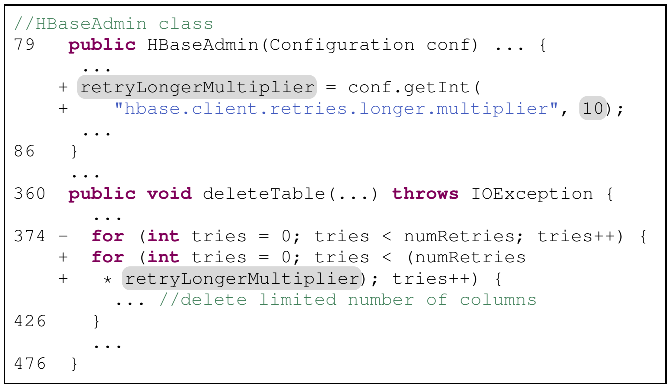
```
//HRegion class
1823  public FlushResult flush(boolean force) ... {
        ...
2077    long flushOpSeqId = getNextSequenceId(wal);
        ...
2098  }
      ...
2416  protected long getNextSequenceId(final WAL wal) {
2417    WALKey key = this.appendEmptyEdit(wal, null);
2418 -  return key.getSequenceId();
     +  return key.getSequenceId(maxWaitForSeqId);
2419  }

//WALKey class
136   private CountDownLatch seqNumAssignedLatch
137                             = new CountDownLatch(1);
      ...
304 - public long getSequenceId() ... {
305 -   seqNumAssignedLatch.await();/*Missing timeout*/
      ...
    + public long getSequenceId(int maxWaitForSeqId)
                           throws IOException {
        ...
    +   if (!seqNumAssignedLatch.await(
    +          maxWaitForSeqId, TimeUnit.MILLISECONDS)) {
    +     throw new IOException("Timed out waiting for"
    +                 + " seq number to be assigned");
    +   }    /* add timeout in the patch */
        ...
312     return this.logSeqNum;
313   }
```

Figure 9. The call graph and patch for the HBase-13971 bug. An example of missing timeout for synchronization. The ·······> represent how `maxWaitForSeqId` flows and be used for `Condition.await`, and the -·-> represents the function call flows.

server by calling `send4LetterWord` function on line 46. `send4LetterWord` creates a `Socket` instance on line 74 and connects to the Zookeeper's address on line 425. Without passing the timeout argument to the `send4LetterWord` function, the `Socket` connection misses timeout. When the network is congested, the required connection packets from this application to Zookeeper is dropped. Therefore, the application hangs, endlessly waiting for Zookeeper's response.

Missing timeout for Synchronization: We found 16 bugs that are caused by missing timeout for synchronization, including file system synchronization and process/thread synchronization. In Figure 9, the HBase-13971 bug shows how missing a timeout for `Condition.await` method causes system failure. In HBase, when the write cache in `MemStore` accumulates enough data, RegionServer needs to flush those write cache into a new `HFile` to disk with the largest sequence ID (line 2077). Before RegionServer gets the sequence ID on line 312, it is blocked by calling `CountDownLatch.await` on line 305. When a new write cache comes, it releases the `CountDownLatch`, and assigns the largest sequence ID to the RegionServer. However, if no more new write cache comes, and the current `MemStore` just hits the limit, the flushing is ongoing but the largest sequence ID is never assigned. Missing timeout on `CountDownLatch.await` on line 305 blocks the RegionServer. The consequence is that RegionServer becomes unavailable.

Observations: We observe that missing timeout bugs often occur when software developers do not consider the system failover mechanism carefully in advance. This group of bugs



```
//HBaseAdmin class
79   public HBaseAdmin(Configuration conf) ... {
       ...
   + retryLongerMultiplier = conf.getInt(
   +     "hbase.client.retries.longer.multiplier", 10);
       ...
86   }
     ...
360  public void deleteTable(...) throws IOException {
       ...
374 -  for (int tries = 0; tries < numRetries; tries++) {
    +  for (int tries = 0; tries < (numRetries
    +    * retryLongerMultiplier); tries++) {
         ... //delete limited number of columns
426    }
       ...
476  }
```

Figure 10. The patch for the HBase-3295 bug. An example of insufficient retry.

are difficult to diagnose because the affected systems often produce no relevant error message. Moreover, missing timeout often imposes serious impact to the affected system or application, which causes the system/application to hang or crash.

C. Improper timeout handling

When a timeout occurs, the system typically performs handling actions including dropping or retrying tasks. We observed that 16 out of 156 bugs are caused by improper timeout handling. We classify the identified improper handling cases into five groups.

Insufficient/missing retries: When a timeout occurs, systems often retry the operations. Sometimes, the retried operation needs to be repeated several times. Without a sufficient number of retries, jobs experience failures eventually. We found eight cases in this category.

For example, Figure 10 shows the patch for the HBase-3295 bug. When deleting a large table, the number of retries should be multiplied (e.g., 10 times of the retry number for deleting a normal-sized table), because HBase can only delete a limited number of columns in each retry. Without a sufficient number of retries, some columns of this large table cannot be deleted successfully.

The HDFS-4404 bug gives another example of how insufficient retries during timeout causes application failure. During failover, the HDFS client tries to connect to the NameNode and create a file. When a timeout occurs due to network congestion, the buggy code fails to perform any retry on file creation, which makes the client believe the HDFS system is unavailable. In this case, the application experiences a failure when creating a file.

Excessive retries: In contrast to insufficient retries, excessive retries waste resources. Retrying the same task many times prolongs the execution time and slows down the performance of the whole system. For excessive retries, we consider the following problem: when a timeout happens, fewer retries should have been taken than what is currently allowed by the program. We found three cases in this category.

For example, the Mapreduce-5616 bug shows how excessive retries cause failover mechanism slowing down. The default retry number for RPC client is 45. When a timeout occurs, the client has to retry 45 times with a 20 seconds timeout interval on each retry. So, the total 45 tries will cause the client to wait 900 seconds before the client is notified of the job failure. In the patch, the retry number is adjusted to 3. So the client can detect the job failure in a more timely fashion.

Incorrect retry: Sometimes, retrying the operation during timeout causes unexpected results. Instead, we should just abort the operation. For example, in the Cassandra-6427 bug, the correct semantic for the update operation requires at-most-once execution, which means when a timeout happens, the system should not retry the operation. Resubmitting the same task leads to undesired results.

Incomplete abort: Aborting those timeout tasks should be complete. Otherwise, the server system or the application will be affected by incomplete aborting. For example, in the Cassandra-7392 bug, when Cassandra server is overloaded, a large group of queries for reading operations timeout. The correct handling should be dropping the whole group of queries. However, Cassandra only drops the to-be-processed queries but still tries to finish all in-process queries. This reduces the throughput of the whole system, causing significant performance degradation.

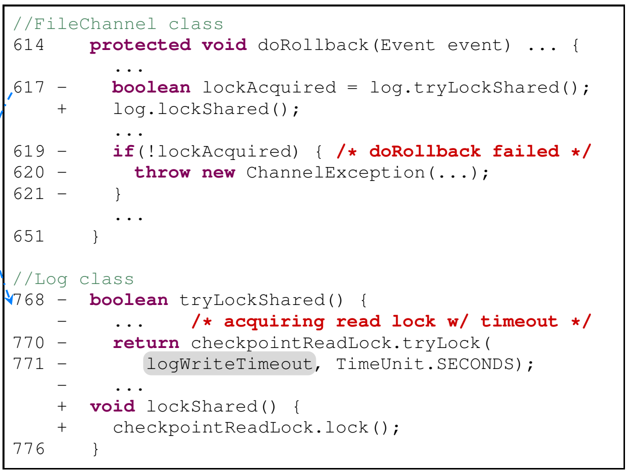
Incorrect abort: Aborting timeout tasks incorrectly can also cause problems. For example, in the Spark-18529 bug, when the `get` operation fails due to timeout, an `AssertionError` is thrown out, which terminates the whole Spark system. The correct handling should be just throwing out a timeout exception.

Observations: We observe that improper timeout handling bugs occur because the software developers do not fully understand the timeout mechanisms. However, it is challenging to implement the proper timeout handling mechanisms, which require the developers to have a deep understanding about the tradeoffs between different timeout handling schemes (e.g., aborting v.s. retry) and their impact to the systems and applications.

D. Unnecessary timeout protection

Software developers should not use timeout mechanisms as the failover mechanism sometimes. Under those circumstances, adding an unnecessary timeout causes system slowdown. We found seven bugs in this category.

For example, Figure 11 shows the patch for the Flume-2307 bug. After checkpoint successfully grabs the write lock, rollback starts and tries to grab the read lock on line 617 by calling `tryLockShared` function which in turn calls `tryLock` function on line 770. The `tryLock` operation is configured with a timeout variable, `logWriteTimeout` on line 771. However, if the checkpoint does not finish in `logWriteTimeout` interval, `tryLockShared` returns false, making rollback failure by throwing the `ChannelException` on line 620. The failure of the rollback causes the corresponding transactions stuck in a limbo. Those



```
//FileChannel class
614     protected void doRollback(Event event) ... {
          ...
617 -     boolean lockAcquired = log.tryLockShared();
    +     log.lockShared();
          ...
619 -     if(!lockAcquired) { /* doRollback failed */
620 -       throw new ChannelException(...);
621 -     }
          ...
651     }

//Log class
768 -   boolean tryLockShared() {
    -     ...    /* acquiring read lock w/ timeout */
770 -     return checkpointReadLock.tryLock(
771 -       logWriteTimeout, TimeUnit.SECONDS);
    -     ...
    +   void lockShared() {
    +     checkpointReadLock.lock();
776     }
```

Figure 11. The partial patch for the Flume-2307 bug. An example of unnecessary timeout. The `tryLockShared` was also called in `doPut`, `doTake`, `doCommit` functions. To save space, we only use `doRollback` as representative. The -·→ represents the function call flows.

unfinished transactions prevent Flume from clearing their log data. Finally, the log data fills up all the disk space. The patch simply replaces the `tryLock` function with the `lock` function, which removes the unnecessary timeout protection for acquiring a lock. This makes the rollback always a success, even though it happens after a long checkpoint.

The Hadoop-491 bug gives another example of how adding an unnecessary timeout causes the system slowdown. In this bug, Hadoop uses timeout mechanisms for all the steaming jobs. When there are streaming jobs that have long idle time, Hadoop arbitrarily kills their connections due to the expired timer. Hadoop then recreates new streaming jobs to replace the killed ones. The re-initialization overhead is heavy, which degrades the whole system's performance. In the patch, Hadoop classified the streaming jobs, revoking the timeout protection for those streaming jobs which have long idle time.

Observations: We observe that unnecessary timeout bugs often occur when software developers mistakenly use timeout retry mechanisms over operations which requires continuous or at-most-once-execution semantics.

E. Clock drifting

The timeout value is often computed by subtracting the start clock-time from the current clock-time. However, when the clocks are out-of-synchronization, the elapsed time is miscalculated, which generates a wrong timer value. We found seven bugs are related to clock drifting.

The HDFS-4307 bug gives an example of how clock drifting can cause either job/system failure or performance degradation, depending on whether the clock drifting causes a shortened or enlarged timer value. For example, when the clock is adjusted by `ntpd` or a system administrator, using the `System.currentTimeMillis` causes time jumping forward or backward unexpectedly. If time jumps forward, the elapsed time gets much longer. All the sockets abruptly expire, which leads to system failure; Otherwise, the elapsed time gets

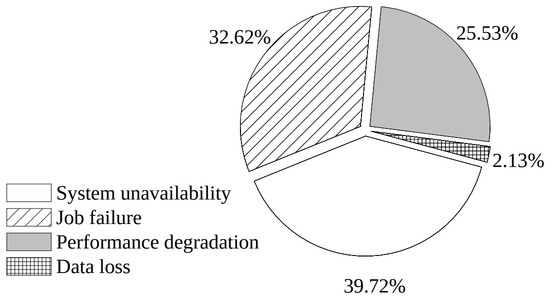
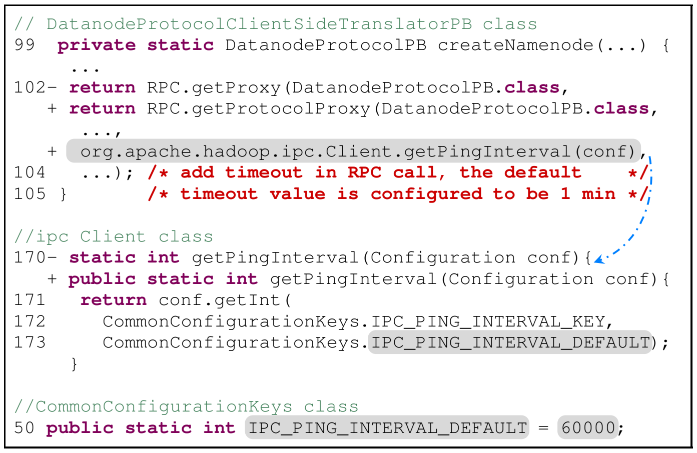


Figure 12. The statistics of the impact of timeout bugs.



```
// DatanodeProtocolClientSideTranslatorPB class
99  private static DatanodeProtocolPB createNamenode(...) {
      ...
102- return RPC.getProxy(DatanodeProtocolPB.class,
   + return RPC.getProtocolProxy(DatanodeProtocolPB.class,
      ...,
   +  org.apache.hadoop.ipc.Client.getPingInterval(conf),
104   ...); /* add timeout in RPC call, the default    */
105 }         /* timeout value is configured to be 1 min */

//ipc Client class
170- static int getPingInterval(Configuration conf){
   + public static int getPingInterval(Configuration conf){
171   return conf.getInt(
172     CommonConfigurationKeys.IPC_PING_INTERVAL_KEY,
173     CommonConfigurationKeys.IPC_PING_INTERVAL_DEFAULT);
    }

//CommonConfigurationKeys class
50 public static int IPC_PING_INTERVAL_DEFAULT = 60000;
```

Figure 13. The patch for the HDFS-4858 bug. When the bug occurs, the whole system is unavailable. The -·-> represents the function call flows.

much smaller, even possibly be negative. All the sockets are left in the cache for a longer period of time, which slows down the whole system.

Observations: Clock drifting in distributed systems is an important factor to consider in timeout calculation and handling, which further makes the timeout bugs prevalent and challenging in cloud computing environments.

IV. Timeout Bug Impact

In this section, we examine the impact of timeout bugs on both systems and applications. As shown in Figure 12, timeout bugs affect systems and/or applications in four major aspects: 40% timeout bugs cause system unavailability, 33% cause application job failures, 26% cause system/application performance degradation and 2% bugs cause application data loss.

A. Unavailability impact

We found 56 out of 156 bugs cause system/application unavailability. We illustrate how timeout bugs can cause the whole system unavailable using the HDFS-4858 bug as another example. As shown in Figure 13, this bug is caused by missing timeout checking for the connection between the DataNode and the secondary NameNode. In HDFS system, the DataNodes should register with both the NameNode and the secondary NameNode. If the NameNode does not work, the secondary NameNode takes its place. The bug occurs when the NameNode is down and the secondary NameNode happens to be rebooting for some reason, such as a power outage. Because the TCP reset packet does not go out due to the abrupt shutdown of network interface, the DataNode never knows the secondary NameNode is rebooted. When the secondary NameNode comes back again, the DataNode never reconnects and re-registers with the secondary NameNode. In this case, the DataNode hangs on sending heartbeats to the secondary NameNode, which causes the whole HDFS system to become unavailable. In the patch, timeout is configured for the socket connection between the DataNode and the secondary NameNode in the `DatanodeProtocolClientSideTranslatorPB` class. The timeout value is set to one minute on line 171-173 of `IPC client` class. If TCP reset packet is not sent, the connection is shut down when the timer expires. Then the DataNode reconnects and re-registers with the secondary NameNode.

The HDFS-3180 bug gives an example of how timeout bugs cause part of the system to become unavailable. The bug is caused by missing a timeout on the connection between the WebHDFS API and HDFS. WebHDFS is a RESTful API for remote applications to access HDFS cluster services. When the network is congested, the HDFS hangs waiting for a response from the WebHDFS connection, which causes WebHDFS service to become unavailable for all remote applications.

The Mapreduce-3186 bug is an example of how timeout bugs cause the application to become unavailable. This bug is caused by missing timeout on the connection between the ApplicationMaster and the ResourceManager. The ApplicationMaster is responsible for the execution of a single application. If the ResourceManager is restarted, the ApplicationMaster loses the contact with the ResourceManager. The ApplicationMaster hangs waiting for the response from the ResourceManager, which causes the application to become unavailable.

B. Application failure impact

We found 46 out of 156 bugs cause application failures. They are often caused by misused timeout or improper timeout handling. Let us use the Mapreduce-6263 bug as an example. This bug is caused by a misconfigured timeout value. Developers set ten seconds as the maximum job execution time. Specifically, YarnRunner connects to the ApplicationMaster, sends the “kill job” command, and waits ten seconds for the job to complete. However, when the job contains a large amount of data, ten seconds is too short for the ApplicationMaster to terminate it. YarnRunner then kills the ApplicationMaster by force, which causes the application failure.

C. Performance degradation impact

We found 36 out of 156 bugs cause performance degradation. Generally speaking, a too large timeout value can cause performance degradation. Misused timeout, unnecessary timeout, and clock drifting can also cause performance

```
// MRJobConfig class
+  public static final String
+    MR_CLIENT_TO_AM_IPC_MAX_RETRIES_ON_TIMEOUTS =
+    MR_PREFIX + "yarn.app.mapreduce.client-am.ipc.max-
     retries-on-timeouts";
+  public static final int
+    DEFAULT_MR_CLIENT_TO_AM_IPC_MAX_RETRIES_ON_TIMEOUTS
     = 3;

// ClientServiceDelegate class
+ this.conf.setInt(
+  CommonConfigurationKeysPublic.
+  IPC_CLIENT_CONNECT_MAX_RETRIES_ON_SOCKET_TIMEOUTS_KEY,
+  this.conf.getInt(MRJobConfig.
+  MR_CLIENT_TO_AM_IPC_MAX_RETRIES_ON_TIMEOUTS,
+  MRJobConfig.
+  DEFAULT_MR_CLIENT_TO_AM_IPC_MAX_RETRIES_ON_TIMEOUTS));

// CommonConfigurationKeysPublic class
379  public static final int IPC_CLIENT_CONNECT_MAX_
     RETRIES_ON_SOCKET_TIMEOUTS_DEFAULT = 45;

// mapred-default.xml
+  <property>
+    <name>yarn.app.mapreduce.client-am.ipc.
+           max-retries-on-timeouts</name>
+    <value>3</value>
+  </property>
```

Figure 14. The patch for the Mapreduce-5616 bug. An example of performance degradation impact.

degradation, including job execution slow down and system recovery delay.

The Mapreduce-5616 bug, shown by Figure 14, gives an example of how timeout bugs cause the job execution slowdown. This bug is caused by excessive retries. The developer introduces an RPC connection timeout on the connection from the client to the ApplicationMaster. In the buggy version, the retry number is set to be 45 by mistake (line 379). Each retry takes 20 seconds. As a result, the client has to wait for 15 minutes for the socket to be closed. To fix the problem, the developer configures the maximum retry number to be 3 in `mapred-default.xml` and `MRJobConfig` class.

The Yarn-3238 bug gives an example of how timeout bugs cause a slow system failover. This bug is caused by excessive reties in an improper timeout handling. When the client builds a connection with the NodeManager, it specifies a retry number for the connection when a timeout occurs. This connection, in turn, invokes `ipc.Client` class, where the retry policy is already configured (45 by default). The two-level retry mechanism causes the client have to wait a very long time for the connection to finally fail, making failover starts much later.

D. Data loss impact

Timeout bugs can also cause other severe problems. We found 3 out of 156 bugs can even cause data loss. For example, the Spark-8958 bug is caused by an unnecessary timeout. In Spark, the `ExecutorAllocationManager` class allocates and removes executors, which are agents responsible for executing tasks, dynamically based on the current workload. The executors store the corresponding tasks' data in the cache. An executor is removed when the executor has been

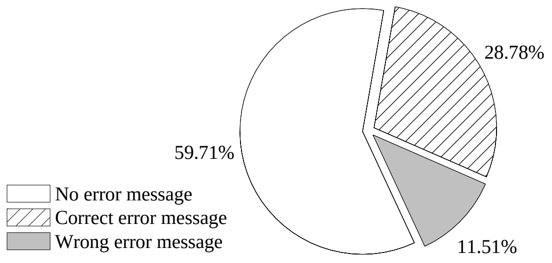


Figure 15. The statistics of diagnosability of timeout bugs.

```
//StorageProxy class
135   public static void mutate(...) {
        ...
143     try{
          ...
162     }
    +   catch (TimeoutException ex) {
    +     if (logger.isDebugEnabled()) {
    +       ...
    +       logger.debug("Write timeout {} for one
    +               (or more) of: ", ex.toString(), mstring);
    +     }
    +     throw ex;
    +   }
163     catch(IOException e)
        ...
606   }
```

Figure 16. The patch for the Cassandra-2532 bug. An example of no error message.

```
//CassandraServer class
1116   public void truncate(...) throws
1117 -                      UnavailableException  ... {
     +    UnavailableException, TimedOutException ... {
        ...
1134     catch (TimeoutException e) {
           ...
1137 -     throw (UnavailableException) new
1138 -                UnavailableException().initCause(e);
     +     throw new TimedOutException();
1139     }
         ...
1143   }
```

Figure 17. The patch for the Cassandra-3651 bug. An example of wrong error message.

idle for `cachedExecutorIdleTimeout` period of time. Removing the executor causes its cached data block to get lost. In the patch, the `cachedExecutorIdleTimeout` is set as `infinity`, making the executor never times out.

V. Timeout Bug Diagnosability

In this section, we examine the diagnosability of timeout bugs. As shown in Figure 15, we found only 40 out of 156 timeout bugs report the correct error message, which makes it difficult to diagnose the timeout bugs.

No error message: This type of bugs are often caused by missing timeout or clock drifting. We found 83 out of 156 bugs report no error message. For example, the Cassandra-2532 bug is shown in Figure 16. When a timeout occurs while writing data to storage, there is no error message. The Cassandra client is unaware of the write request failure. The patch is to add the corresponding code block for timeout error handling—catch the `TimeoutException` and record the error message.

The HDFS-8311 bug is another example, which is caused by missing timeout for the socket read operation. When the network congestion occurs, the HDFS client endlessly waits for a response from the DataNode. Under this circumstance, the HDFS client reports no error message.

Wrong error message: Sometimes applications report misleading error messages when a timeout occurs. Or timeout bugs cause other errors which report confusing messages. We found 16 out of 156 bugs in this category.

The Cassandra-3651 bug gives an example of how application reports misleading error messages shown by Figure 17. When `truncate` operations failed due to timeout, it throws an `UnavailableException` rather than the `TimeOutException`.

The Mapreduce-709 bug gives another example. Mapreduce is configured with node health check function, which is run to check each node's status periodically. The timeout variable `mapred.healthChecker.script.timeout` adds a time limit on the health checking process. If the health checking process exceeds the timeout value, the system throws an `IOException` instead of a `TimeoutException`.

VI. Related Work

Several recent empirical studies have looked at general bugs in distributed systems [14], [15], [16], [17]. Gunawi et al. studied 3000 issues in different distributed systems [14]. Huang et al. studied 4000 issues in Hadoop systems [15]. They both found that timeout bugs widely exist in distributed systems. However, their studies do not focus on studying the root cause, impact, and diagnosability of timeout bugs, which is the focus of our work.

There have also been empirical studies that focus on specific types of bugs in distributed systems, such as misconfigurations that cause failures in distributed systems [18], data corruptions in file systems and distributed systems [19], [20], general performance degradation and resource-waste issues in real-world distributed systems [21], [22], concurrency bugs caused by unexpected distributed timing [23]. Our work is complementary to above studies, providing the first comprehensive study about timeout bugs that occurred in real-world cloud server systems.

Previous work has also explored various performance diagnosis schemes. X-ray [24] applies program dependency analysis techniques to automatically identify configuration entries that are responsible for performance problems. Fournier et al. [25] proposed to analyze the blocking behavior using kernel-level system events to diagnose performance problems. PerfCompass [26] leveraged machine learning techniques to differentiate external faults from internal faults. PerfScope [27] identified buggy functions for system performance anomalies. However, our recent study [22] shows that existing solutions cannot effectively detect or diagnose performance anomalies caused by timeout bugs.

We believe that this paper could provide insights for the future development of timeout bug detection and diagnosis tools.

Work has also been done to detect or fix concurrency bugs [28], [29], [30], [31], [32], [33], [34], [35]. Existing work focuses on timing problems caused by missing synchronization operations or mis-using synchronization APIs/variables, such as data races [28], [29], atomicity violations [30], [31], order violations [32], and deadlocks [33], [34], [35]. Our study reveals an under-studied type of root causes for concurrency bugs: missing timeout, misused timeout, and unnecessary timeout. Our study provides insights for future development of detecting and fixing those bugs.

VII. Conclusion

We have presented a comprehensive characteristic study about 156 real-world timeout bugs which are discovered in 11 popular open source cloud server systems. Our study reveals a set of interesting findings: 1) 81% timeout problems are caused by either misused timeout values or missing timeout checking; 2) timeout problems have serious impact to both cloud server systems and applications, which can make system unavailable, cause application execution failures, bring significant performance degradation, and even data loss; and 3) existing timeout issues are difficult to diagnose with 71% bugs producing no error message or misleading error messages. As part of our on-going work, we plan to develop efficient timeout bug detection and diagnosis schemes to enhance the resilience of cloud server systems against timeout bugs.

VIII. Acknowledgements

We thank the anonymous reviewers for their valuable comments. This work was sponsored in part by NSF CNS1513942 grant and NSF CNS1149445 grant. Any opinions expressed in this paper are those of the authors and do not necessarily reflect the views of NSF.

References

[1] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, *Operating Systems: Three Easy Pieces*, May 2015.

[2] U. A. Khan and B. Rinner, "Online learning of timeout policies for dynamic power management," *TECS*, vol. 13, no. 4, p. 96, 2014.

[3] S. O. Luiz, A. Perkusich, B. M. Cruz, B. H. Neves, and G. M. d. S. Araujo, "Optimization of timeout-based power management policies for network interfaces," *CE*, vol. 59, no. 1, pp. 101–106, 2013.

[4] H. Zhu, H. Fan, X. Luo, and Y. Jin, "Intelligent timeout master: Dynamic timeout for sdn-based data centers," in *IM*, 2015.

[5] N.-N. Dao, J. Park, M. Park, and S. Cho, "A feasible method to combat against ddos attack in sdn network," in *ICOIN*, 2015.

[6] "Irreversible Failures: Lessons from the DynamoDB Outage," http://blog.scalyr.com/2015/09/irreversible-failures-lessons-from-the-dynamodb-outage/.

[7] "Apache JIRA," https://issues.apache.org/jira.

[8] "Hadoop core configuration," https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-common/core-default.xml.

[9] "Hadoop HDFS configuration," https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/hdfs-default.xml.

[10] "Hadoop Mapreduce configuration," https://hadoop.apache.org/docs/current/hadoop-mapreduce-client/hadoop-mapreduce-client-core/mapred-default.xml.

[11] "Hadoop Yarn configuration," https://hadoop.apache.org/docs/current/hadoop-yarn/hadoop-yarn-common/yarn-default.xml.

[12] "Apache Spark configuration," https://spark.apache.org/docs/latest/configuration.html.

[13] T. Xu, X. Jin, P. Huang, Y. Zhou, S. Lu, L. Jin, and S. Pasupathy, "Early detection of configuration errors to reduce failure damage," in *OSDI*, 2016.

[14] H. S. Gunawi, M. Hao, T. Leesatapornwongsa, T. Patana-anake, T. Do, J. Adityatama, K. J. Eliazar, A. Laksono, J. F. Lukman, V. Martin *et al.*, "What bugs live in the cloud?: A study of 3000+ issues in cloud systems," in *SOCC*, 2014.

[15] J. Huang, X. Zhang, and K. Schwan, "Understanding issue correlations: a case study of the hadoop system," in *SOCC*, 2015.

[16] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *SOSP*, 2001.

[17] L. Lu, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and S. Lu, "A study of linux file system evolution," in *FAST*, 2013.

[18] Z. Yin, X. Ma, J. Zheng, Y. Zhou, L. N. Bairavasundaram, and S. Pasupathy, "An empirical study on configuration errors in commercial and open source systems," in *SOSP*, 2011.

[19] P. Wang, D. J. Dean, and X. Gu, "Understanding real world data corruptions in cloud systems," in *IC2E*, 2015.

[20] Y. Zhang, A. Rajimwale, A. C. Arpaci-Dusseau, and R. H. Arpaci-Dusseau, "End-to-end data integrity for file systems: A zfs case study," in *FAST*, 2010.

[21] G. Jin, L. Song, X. Shi, J. Scherpelz, and S. Lu, "Understanding and detecting real-world performance bugs," in *PLDI*, 2012.

[22] D. J. Dean, P. Wang, X. Gu, W. Enck, and G. Jin, "Automatic server hang bug diagnosis: Feasible reality or pipe dream?" in *ICAC*, 2015.

[23] T. Leesatapornwongsa, J. F. Lukman, S. Lu, and H. S. Gunawi, "Taxdc: A taxonomy of non-deterministic concurrency bugs in datacenter distributed systems," in *ASPLOS*, 2016.

[24] M. Attariyan, M. Chow, and J. Flinn, "X-ray: Automating root-cause diagnosis of performance anomalies in production software," in *OSDI*, 2012.

[25] P. Fournier and M. R. Dagenais, "Analyzing blocking to debug performance problems on multi-core systems," in *SIGOPS*, 2010.

[26] D. J. Dean, H. Nguyen, P. Wang, and X. Gu, "Perfcompass: Toward runtime performance anomaly fault localization for infrastructure-as-a-service clouds."

[27] D. J. Dean, H. Nguyen, X. Gu, H. Zhang, J. Rhee, N. Arora, and G. Jiang, "PerfScope: Practical online server performance bug inference in production cloud computing infrastructures," in *SOCC*, 2014.

[28] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. Anderson, "Eraser: A dynamic data race detector for multi-threaded programs," in *SOSP*, 1997.

[29] P. Liu, O. Tripp, and C. Zhang, "Grail: Context-aware fixing of concurrency bugs," in *FSE*, 2014.

[30] G. Jin, L. Song, W. Zhang, S. Lu, and B. Liblit, "Automated atomicity-violation fixing," in *PLDI*, 2011.

[31] M. Vaziri, F. Tip, and J. Dolby, "Associating synchronization constraints with data in an object-oriented language," in *POPL*, 2006.

[32] G. Jin, W. Zhang, D. Deng, B. Liblit, and S. Lu, "Automated concurrency-bug fixing," in *OSDI*, 2012.

[33] Y. Lin and S. Kulkarni, "Automatic repair for multi-threaded program with deadlock/livelock using maximum satisfiability," in *ISSTA*, 2014.

[34] Y. Wang, T. Kelly, M. Kudlur, S. Lafortune, and S. A. Mahlke, "Gadara: Dynamic deadlock avoidance for multithreaded programs," in *OSDI*, 2008.

[35] D. Weeratunge, X. Zhang, and S. Jaganathan, "Accentuating the positive: atomicity inference and enforcement using correct executions," in *OOPSLA*, 2011.