Understanding and Discovering Configuration Dependencies in Modern Software Systems

1 Roadmap

The purpose of this document is for:

- defining terminology to make our communication more efficient (given that we have many folks in the team);
- keeping everyone on the same page about the roadmap;
- recording the progress and results as checkpoints.

Many of the write-ups will later be made into the final paper, but this document is not written for submission.

We plan to conduct the project in four phases:

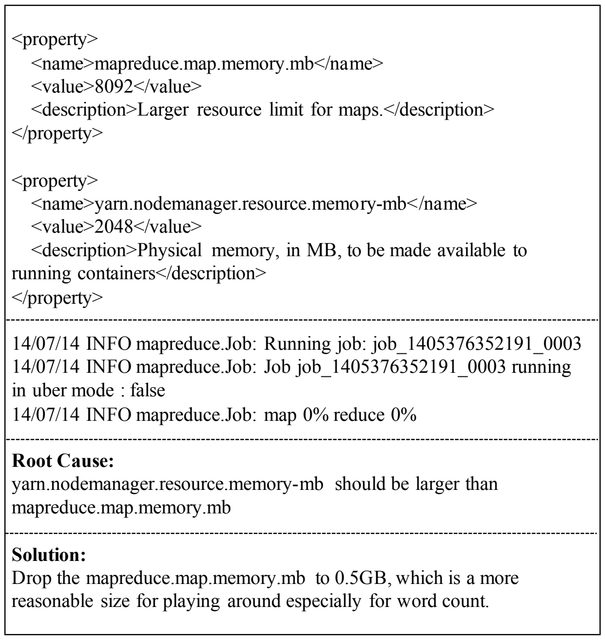
- **Phase 1 (§2; §3).** Collect potential configuration dependencies from text data (documentation, configuration files, and StackOverflow). We include both configuration dependencies within a single software components and across multiple components. The end results are a number of intra- and inter-component dependencies. We plan to include two software stacks: (1) Hadoop big data stack and (2) OpenStack cloud stack.
- **Phase 2 (§4).** Study how dependencies are formulated in the source code, for every potential configuration dependency collected from Phase 1. The end results are source code patterns. In this phase, we also remove the false cases from the first phase caused by categorization errors or ambiguity.
- **Phase 4 (§5; §7; §6).** Study how dependencies are checked, logged (if violated), and the impact of violation in the source code.
- **Phase 5 (§8).** Build an automatic tool which takes the source code of given systems as input, and discovers configuration dependencies based on the code patterns summarized in Phase 2. We plan to build the tool on top of Soot Java compiler framework. Note that we only plan to build the tool for Java code (we do not plan to deal with Python code—OpenStack will not be included in Phase 3).
- **Phase 6 (§8.2).** We build use cases with the configuration dependencies discovered by the tool. The use cases include: (1) better configuration design to eliminate unnecessary dependencies; (2) better documentation to cover dependencies; and (3) detecting misconfigurations that violate the dependencies.

1.1 Introduction

We just give 3 "dramatic" examples now in order to motivate the importance of the problem (configuration dependency).

The first example is some issues from JIRA. The dependency is `hbase.lease.recovery.dfs.timeout >= dfs.client.socket-timeout + dfs.heartbeat.interval + hbase.lease.recovery.pause`. The constraint is hard because it involves four parameters and two components. Even developers make mistakes in the process of dealing with this constraint. Developers use HBASE-8354, HBASE-8389, HBASE-8449, HBASE-13200 and other issues to fix bugs related the constraint. In HBASE-13200, improper configuration can lead to endless lease recovery during failover. When a node (DataNode + RegionServer) has machine/OS level failure, another RegionServer will try to do lease recovery for the log file. It will retry for every `hbase.lease.recovery.dfs.timeout` from the second time. When the hdfs configuration is not properly, the lease recovery time will exceeded the timeout. It leads to endless retries and preemptions until the final timeout and lease revovery can never succeed. Developers fixed the bug, but misconfiguration can still lead to the failure of the lease revovery.

The second example is a real problem from stackoverflow Question/24747427. The user tyied to run the Word Count Example to test his hadoop setup. But mapreduce job got stuck in 0%. And he tried some other simple jobs and each one of them stuck. The configuration and logs are show in



```
<property>
  <name>mapreduce.map.memory.mb</name>
  <value>8092</value>
  <description>Larger resource limit for maps.</description>
</property>

<property>
  <name>yarn.nodemanager.resource.memory-mb</name>
  <value>2048</value>
  <description>Physical memory, in MB, to be made available to
running containers</description>
</property>
```

```
14/07/14 INFO mapreduce.Job: Running job: job_1405376352191_0003
14/07/14 INFO mapreduce.Job: Job job_1405376352191_0003 running
in uber mode : false
14/07/14 INFO mapreduce.Job: map 0% reduce 0%
```

Root Cause:
yarn.nodemanager.resource.memory-mb should be larger than mapreduce.map.memory.mb

Solution:
Drop the mapreduce.map.memory.mb to 0.5GB, which is a more reasonable size for playing around especially for word count.

Figure 1. Dramatic Example. The figures shows the configuration and log information of the Question/24747427 from stackoverflow.

```
--- hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-
mapreduce-client-core/src/main/resources/mapred-default.xml
+++ hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-
mapreduce-client-core/src/main/resources/mapred-default.xml

<property>
   <name>mapreduce.job.maxtaskfailures.per.tracker</name>
-  <value>4</value>
+  <value>3</value>
   <description>The number of task-failures on a tasktracker of a given
job
-  after which new tasks of that job aren't assigned to it.
+  after which new tasks of that job aren't assigned to it. It
+  MUST be less than mapreduce.map.maxattempts and
+  mapreduce.reduce.maxattempts otherwise the failed task will
+  never be tried on a different node.
   </description>
</property>
```

Figure 2. Dramatic Example. The figures shows the patch of the issue MAPREDUCE-4604.

Figure 3. The dependency is really hard because the descriptions of parameter don't point the constraint, which can only be inferred by understanding the semantics of the descriptions. The dependency is also complex because the constraint will be dynamically checked to find whether the map size is less than the unallocated resource on the node. There are about 10 similar cases in stackoverflow, for example Question/26739242, Question/29001702 and Question/43567134.

The third example is an issue from JIRA. `mapreduce.job.maxtaskfailures.per.tracker` is the number of task-failures on a node manager of a given job after which new tasks of that job aren't assigned to it. It must be less than `mapreduce.map.maxattempts` and `mapreduce.reduce.maxattempts`. In previous versions, `mapreduce.reduce.maxattempts` defaults are set to 4 as well as `mapreduce.job.maxtaskfailures.per.tracker`. And the descriptions didn't point the constraint. This causes the AM to fail the job at the same time as it blacklists a node, thus never actually trying another node. Developers realized the bug and fixed it in issue MAPREDUCE-4604. The patch of the issue is shown in Figure 4.

1.2 Introduction

We just give 3 "dramatic" examples now in order to motivate the importance of the problem (configuration dependency).

The first example is some issues from JIRA. The dependency is `hbase.lease.recovery.dfs.timeout >= dfs.client.socket-timeout + dfs.heartbeat.interval + hbase.lease.recovery.pause`. The constraint is hard because it involves four parameters and two components. Even developers make mistakes in the process of dealing with this constraint. Developers use HBASE-8354, HBASE-8389, HBASE-8449, HBASE-13200 and other issues to fix bugs related the constraint. In HBASE-13200, improper configuration can lead to endless lease recovery during failover. When

```
<property>
   <name>mapreduce.map.memory.mb</name>
   <value>8092</value>
   <description>Larger resource limit for maps.</description>
</property>

<property>
   <name>yarn.nodemanager.resource.memory-mb</name>
   <value>2048</value>
   <description>Physical memory, in MB, to be made available to
running containers</description>
</property>
--------------------------------------------------------------------
14/07/14 INFO mapreduce.Job: Running job: job_1405376352191_0003
14/07/14 INFO mapreduce.Job: Job job_1405376352191_0003 running
in uber mode : false
14/07/14 INFO mapreduce.Job: map 0% reduce 0%
--------------------------------------------------------------------
```

Root Cause:
yarn.nodemanager.resource.memory-mb should be larger than mapreduce.map.memory.mb

Solution:
Drop the mapreduce.map.memory.mb to 0.5GB, which is a more reasonable size for playing around especially for word count.

Figure 3. Dramatic Example. The figures shows the configuration and log information of the Question/24747427 from stackoverflow.

a node (DataNode + RegionServer) has machine/OS level failure, another RegionServer will try to do lease recovery for the log file. It will retry for every `hbase.lease.recovery.dfs.timeout` from the second time. When the hdfs configuration is not properly, the lease recovery time will exceeded the timeout. It leads to endless retries and preemptions until the final timeout and lease revovery can never succeed. Developers fixed the bug, but misconfiguration can still lead to the failure of the lease revovery.

The second example is a real problem from stackoverflow Question/24747427. The user tyied to run the Word Count Example to test his hadoop setup. But mapreduce job got stuck in 0%. And he tried some other simple jobs and each one of them stuck. The configuration and logs are show in

Figure 3. The dependency is really hard because the descriptions of parameter don't point the constraint, which can only be inferred by understanding the semantics of the descriptions. The dependency is also complex because the constraint will be dynamically checked to find whether the map size is less than the unallocated resource on the node. There are about 10 similar cases in stackoverflow, for example Question/26739242, Question/29001702 and Question/43567134.

The third example is an issue from JIRA. `mapreduce.job.maxtaskfailures.per.tracker` is the number of task-failures on a node manager of a given job after which new tasks of that job aren't assigned to it. It must be less than `mapreduce.map.maxattempts` and `mapreduce.reduce.maxattempts`. In previous versions, `mapreduce.reduce.maxattempts` defaults are set to 4 as well as `mapreduce.job.maxtaskfailures.per.`

```
--- hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-
mapreduce-client-core/src/main/resources/mapred-default.xml
+++ hadoop-mapreduce-project/hadoop-mapreduce-client/hadoop-
mapreduce-client-core/src/main/resources/mapred-default.xml

<property>
   <name>mapreduce.job.maxtaskfailures.per.tracker</name>
-  <value>4</value>
+  <value>3</value>
   <description>The number of task-failures on a tasktracker of a given
job
-  after which new tasks of that job aren‘t assigned to it.
+  after which new tasks of that job aren‘t assigned to it. It
+  MUST be less than mapreduce.map.maxattempts and
+  mapreduce.reduce.maxattempts otherwise the failed task will
+  never be tried on a different node.
   </description>
</property>
```

Figure 4. Dramatic Example. The figures shows the patch of the issue MAPREDUCE-4604.

`tracker`. And the descriptions didn't point the constraint. This causes the AM to fail the job at the same time as it blacklists a node, thus never actually trying another node. Developers realized the bug and fixed it in issue MAPREDUCE-4604. The patch of the issue is shown in Figure 4.

2 Data Collection

2.1 Target Systems

For Hadoop stack, we select eight systems: Alluxio, HBase, HDFS, Mapreduce, Spark, Yarn, Zookeeper and Hadoop Core.

TODO: We need to decide the systems for OpenStack.

2.2 Collecting Configuration Dependencies

Since source code patterns of configuration dependencies have never been studied or known, we start from text data. We find that many configuration dependencies are described in documents (e.g., manuals), configuration metadata, and StackOverflow posts. So we mine the text data to extract configuration dependencies. Note that the mined configuration dependencies are just candidates instead of ground truth. We treat the source code as the ground truth and we will validate every dependency later in Phase 2.

2.2.1 Analyzing Structured Descriptions

We realize that the configuration metadata in the form of (e.g., `*-default.xml` in Hadoop) provides structured text data that describe configuration parameters. We apply the following two strategies to mine intra-software configuration dependencies from configuration metadata.

- **Strategy A.** If the description of one configuration parameter contains the name of another configuration parameter, then we consider these two potentially have dependencies.
- **Strategy B.** If the description of one configuration parameter contains the keyword of another configuration parameter, then we consider them potentially have dependencies.

We apply the following four strategies to mine inter-software configuration dependencies from configuration metadata.

- **Strategy C.** If the description of one configuration parameter contains the name of another configuration parameter, then we consider these two potentially have dependencies.
- **Strategy D.** If the description of one configuration parameter contains the keyword of another configuration parameter, then we consider them potentially have dependencies.
- **Strategy E.** If the description of one configuration contains the name of another application, then we will assume it to be potentially related to the other application. Thus, we will record it and manually check it to see whether it has any relation with the configuration parameters from other applications.
- **Strategy F.** We use neural embedding methods to embedding the description of each configuration parameter into a vector and then, for each parameter, we compute the cosine similarity between its vector and

the vectors of configuration parameters from other applications, and pick the most similar one as potentially correlated one.

2.2.2 Analyzing Unstructured Document Pages

We apply the following strategy to mine potential configuration dependencies from the user manual:

- **Strategy G.** If two configuration parameters from different applications occur in the same manual page, we consider them as potentially correlated.

2.2.3 Analyzing Online Forums

We find that crowdsourcing Q/A forums (e.g., StackOverflow) host posts in which users ask questions and misconfiguration is a common topic. We apply the following strategy:

- **Strategy H.** We dump the whole content of the platform. Then for each post, we search whether two configurations from different applications occur in the answer part and if that exists, we record them and view them as potentially correlated. The reason why we only take a look at the answer part is that answer part usually analyzes the problems. Thus, if two configurations are mentioned for analyzes, it is highly likely that they are correlated.

2.3 Summary

Hadoop			**OpenStack**		
	Intra	**Inter**		**Intra**	**Inter**
Alluxio	10	8	Nova		
HBase	15	11	Swift		
HDFS	48	20	Congress		
MapRed	29	24	General		
Spark	33	16			
Yarn	42	59			
Zookeeper	4	11			
Core	45	37			
Total	226	93			

Table 1. The number of intra- and inter-component configuration dependencies found in each application.

3 Categorization

3.1 Definition

Rather than impose a taxonomy *ex ante*, we build ours bottom-up based on studying real-world configuration dependencies and looking for patterns. We find that both intra- and inter-component configuration dependencies fall into a similar set of dependency categories. In general, we find that configuration dependencies can be categorized as follows:

Control dependency. We define a control dependency between between two configuration parameters A and B, if B's effect is dominated by A's value, denoted as

$$B \longrightarrow_c A$$

The most common form of control dependencies is that B will only be used by the program if A is enabled.

Value dependency. We define a value dependency between two configuration parameters A and B, if their values are independent—a change of one would affect the other.

$$B \longrightarrow_v A$$

We find that value dependencies are manifested through a diverse range of patterns:

- **Constraints**. A and B have to satisfy certain correctness constraints required by the program, including numeric (e.g., A has to be less than B), membership (e.g., A has to be a subset of B), and logic constraints (e.g., A and B cannot be enabled at the same time).
- **Composition**. A's value is composed by B. We find two common composition patterns: (1) A's value serves as the *default value* of B, and (2) A's value *overwrites* B's value.
- **Used together.** A and B are used together in binary operations, $\diamond_b$. Changing A's value affects the operation that involves B.
- **Resource competition.** A and B compete for the system resources including physical resources (e.g., CPU and memory) or OS abstractions (e.g., file descriptors and ports). Therefore, the values of A and B have to satisfy system-level constraints.

3.2 Overall Statistics

Intra-component dependency. As we can see from Table 1, Yarn, Hadoop Core and HDFS have the most Intra-component dependencies . [Chen: I have computed the number of parameters for each application also and it does not show the rule that more configurations imply more Intra-component dependencies , so we may need to reason why these three applications have most Intra-component dependencies from docs].

Inter-component dependency. In this section, we will analyze in detail about the code patterns for Inter-component dependencies . We have analyzed where each dependency case comes from and the result is summarized in Table 3 and

Dependency Type	**Intra Comp.**	**Inter Comp.**
Control dependency	126	18
Value dependency	100	75
→ Constraints	49	53
→ Composition	42	20
→ Used together	9	2
→ Resource competition		

Table 2. The number of dependency types for intra- and inter-component dependencies.

Application Pairs	# of Inter Comp.
Hadoop-Core, HDFS	16
Hadoop-Core, Yarn	16
Yarn, Alluxio	8
Hbase, ZooKeeper	8
Yarn, Mapreduce	21
Hadoop-Core, Spark	2
Hadoop-Core, Zoopkeeper	2
Hbase, HDFS	1
Hadoop-Core, Hbase	1
Hadoop-Core, Mapreduce	1
Yarn, Spark	13
Yarn, ZooKeeper	1
Yarn, HDFS	1
Mapreduce, Spark	1
Mapreduce, HDFS	1
Total	93

Table 3. The number of inter-component dependencies between different components.

the total number of each dependency type is presented in Table 1. In the following, we will analyze the code patterns of each dependency one by one. In addition, for better illustrations, we provide the concrete examples for each dependency type in Figure **??** and Figure **??**.

4 Formulation

For a further analysis of Intra-component dependencies , we classify each dependency case into it corresponding category defined in Section 3. The results are summarized in Table 2. In the following, we will analyze the code patterns for each category in detail.

4.1 Control Dependency

As we see from Table 2, Control dependency accounts for most of the dependencies, which is about 53.6% in Intra-component dependencies and 25.4% in Inter-component dependencies . Since in control dependency, parameter A defines the working scope of Parameter B, failure to recognize this kind of dependency will result in one parameter to lose effectiveness. There are 2 unique code patterns we have found for this kind of dependency. The first pattern is

```
if ( ParameterA == SomeValue )
  doWork(ParameterB);
```

where ParameterB will only be used by the program if ParameterA is SomeValue. The second pattern is

```
ClassA = getClass(ParameterA);
ClassA.Function(){
  doWork(ParameterB);
}
```

where ParameterA is used to initialize a class object and ParameterB only works in the scope of a specific class chosen by ParameterA. For better illustrations, we have shown the example codes for these patterns, which are shown in Figure 5.

4.2 Value Dependency

4.2.1 Constraints

This dependency accounts for 21.1% of total Intra-component dependencies and accounts for 33.3% of total Inter-component dependencies . Since value dependency defines the constraints over the values between two configuration parameters A and B, failure to recognize this kind of dependency will make the program throw exception, log error message or directly override users' provided configuration values.

Numeric Constraint. For numeric constraints, we find 2 unique patterns. The first pattern is

```
if MathFunc(ParameterA) op MathFunc(ParameterB)
```

where *MathFunc* denotes the parameter may be added or multiplied by some constant values and *op* denotes binary operators like > . The second pattern is

```
C= Max/Min(ParameterA,ParameterB)
```

, where either ParameterA is capped by ParameterB or ParameterB is capped by ParameterA. For better illustrations, we have shown the example codes for these patterns, which are shown in Figure 6.

Membership Constraint. For membership constraints, we find 1 unique pattern. The first pattern is

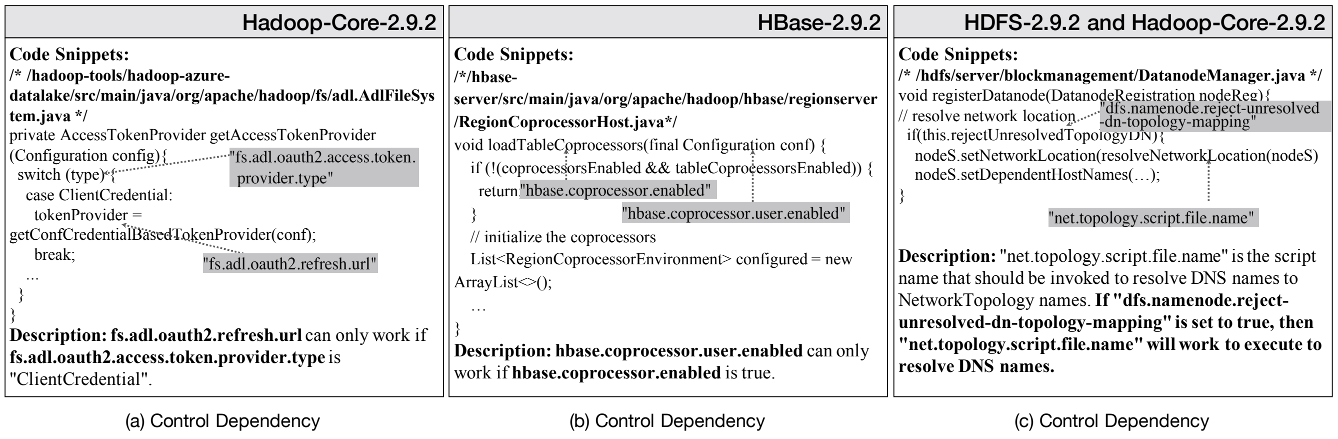
```
// ParameterA and ParameterB are both List
for (item in ParameterA):
  if (!ParameterB.contains(item))
    throw Exception();
```

where ParameterA and ParameterB are both Lists. For better illustrations, we have shown the example code for this pattern, which is shown in Figure 7.

Logic Constraint. For logic constraints, we find one unique pattern. The pattern is

```
if ( condition(ParameterA) && condition(ParameterB))
```

which means ParameterA and ParameterB have some satisfy some conditions at the same time. For better illustrations, we have shown the example code for this pattern in Figure 8.


Hadoop-Core-2.9.2

Code Snippets:

```
/* /hadoop-tools/hadoop-azure-datalake/src/main/java/org/apache/hadoop/fs/adl.AdlFileSystem.java */
private AccessTokenProvider getAccessTokenProvider
(Configuration config){
  switch (type) {
    case ClientCredential:
      tokenProvider =
getConfCredentialBasedTokenProvider(conf);
      break;
    …
  }
}
```

"fs.adl.oauth2.access.token.provider.type"

"fs.adl.oauth2.refresh.url"

Description: fs.adl.oauth2.refresh.url can only work if **fs.adl.oauth2.access.token.provider.type** is "ClientCredential".

(a) Control Dependency

HBase-2.9.2

Code Snippets:

```
/*/hbase-server/src/main/java/org/apache/hadoop/hbase/regionserver/RegionCoprocessorHost.java*/
void loadTableCoprocessors(final Configuration conf) {
  if (!(coprocessorsEnabled && tableCoprocessorsEnabled)) {
    return;
  }
  // initialize the coprocessors
  List<RegionCoprocessorEnvironment> configured = new
ArrayList<>();
  …
}
```

"hbase.coprocessor.enabled"

"hbase.coprocessor.user.enabled"

Description: hbase.coprocessor.user.enabled can only work if **hbase.coprocessor.enabled** is true.

(b) Control Dependency

HDFS-2.9.2 and Hadoop-Core-2.9.2

Code Snippets:

```
/* /hdfs/server/blockmanagement/DatanodeManager.java */
void registerDatanode(DatanodeRegistration nodeReg){
// resolve network location
  if(this.rejectUnresolvedTopologyDN){
    nodeS.setNetworkLocation(resolveNetworkLocation(nodeS)
    nodeS.setDependentHostNames(…);
}
```

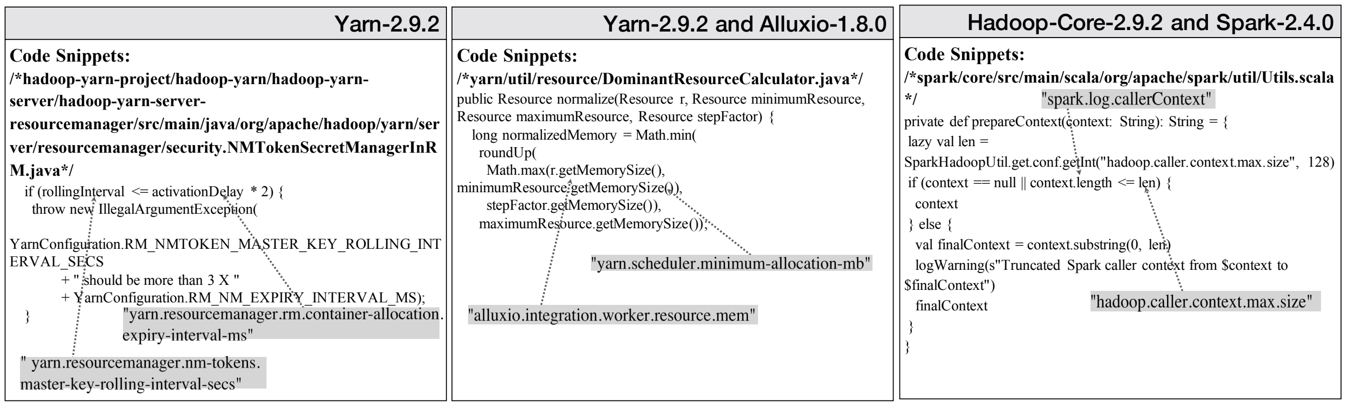
"dfs.namenode.reject-unresolved-dn-topology-mapping"

"net.topology.script.file.name"

Description: "net.topology.script.file.name" is the script name that should be invoked to resolve DNS names to NetworkTopology names. **If "dfs.namenode.reject-unresolved-dn-topology-mapping" is set to true, then "net.topology.script.file.name" will work to execute to resolve DNS names.**

(c) Control Dependency

Figure 5. Example of Control Dependency.


Yarn-2.9.2

Code Snippets:

```
/*hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java/org/apache/hadoop/yarn/server/resourcemanager/security.NMTokenSecretManagerInRM.java*/
  if (rollingInterval <= activationDelay * 2) {
    throw new IllegalArgumentException(

YarnConfiguration.RM_NMTOKEN_MASTER_KEY_ROLLING_INT
ERVAL_SECS
          + " should be more than 3 X "
          + YarnConfiguration.RM_NM_EXPIRY_INTERVAL_MS);
  }
```

"yarn.resourcemanager.rm.container-allocation.expiry-interval-ms"

" yarn.resourcemanager.nm-tokens.master-key-rolling-interval-secs"

(a) Numeric Constraint

Yarn-2.9.2 and Alluxio-1.8.0

Code Snippets:

```
/*yarn/util/resource/DominantResourceCalculator.java*/
public Resource normalize(Resource r, Resource minimumResource,
Resource maximumResource, Resource stepFactor) {
  long normalizedMemory = Math.min(
    roundUp(
      Math.max(r.getMemorySize(),
minimumResource.getMemorySize()),
      stepFactor.getMemorySize()),
    maximumResource.getMemorySize());
```

"yarn.scheduler.minimum-allocation-mb"

"alluxio.integration.worker.resource.mem"

(b) Numeric Constraint

Hadoop-Core-2.9.2 and Spark-2.4.0

Code Snippets:

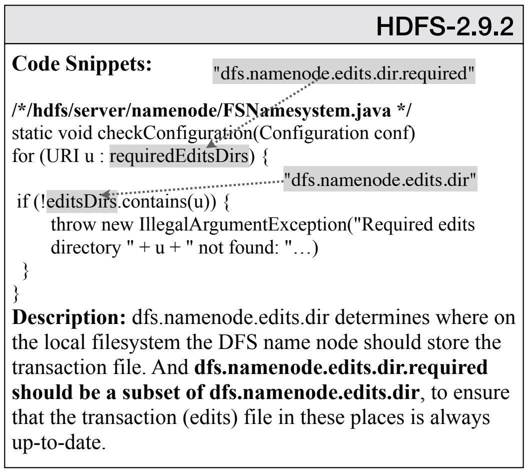
```
/*spark/core/src/main/scala/org/apache/spark/util/Utils.scala
*/
private def prepareContext(context: String): String = {
 lazy val len =
SparkHadoopUtil.get.conf.getInt("hadoop.caller.context.max.size", 128)
 if (context == null || context.length <= len) {
  context
 } else {
  val finalContext = context.substring(0, len)
  logWarning(s"Truncated Spark caller context from $context to
$finalContext")
  finalContext
 }
}
```

"spark.log.callerContext"

"hadoop.caller.context.max.size"

(c) Numeric Constraint

Figure 6. Example of Numeric Constraints.


HDFS-2.9.2

Code Snippets:

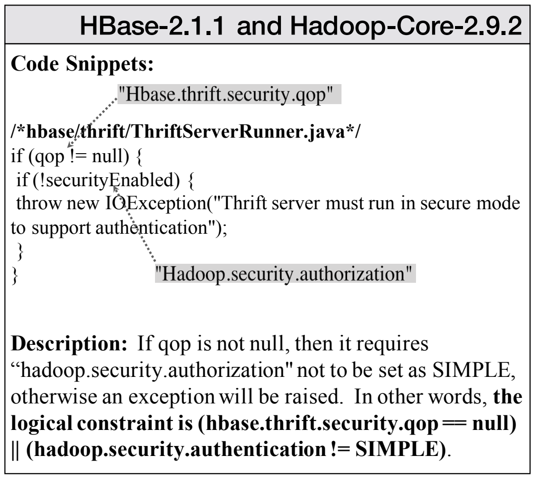
```
/*/hdfs/server/namenode/FSNamesystem.java */
static void checkConfiguration(Configuration conf)
for (URI u : requiredEditsDirs) {
 if (!editsDirs.contains(u)) {
    throw new IllegalArgumentException("Required edits
    directory " + u + " not found: "…)
  }
}
```

"dfs.namenode.edits.dir.required"

"dfs.namenode.edits.dir"

Description: dfs.namenode.edits.dir determines where on the local filesystem the DFS name node should store the transaction file. And **dfs.namenode.edits.dir.required should be a subset of dfs.namenode.edits.dir**, to ensure that the transaction (edits) file in these places is always up-to-date.

Figure 7. Example of Membership Constraints.


HBase-2.1.1 and Hadoop-Core-2.9.2

Code Snippets:

```
/*hbase/thrift/ThriftServerRunner.java*/
if (qop != null) {
 if (!securityEnabled) {
 throw new IOException("Thrift server must run in secure mode
to support authentication");
 }
}
```

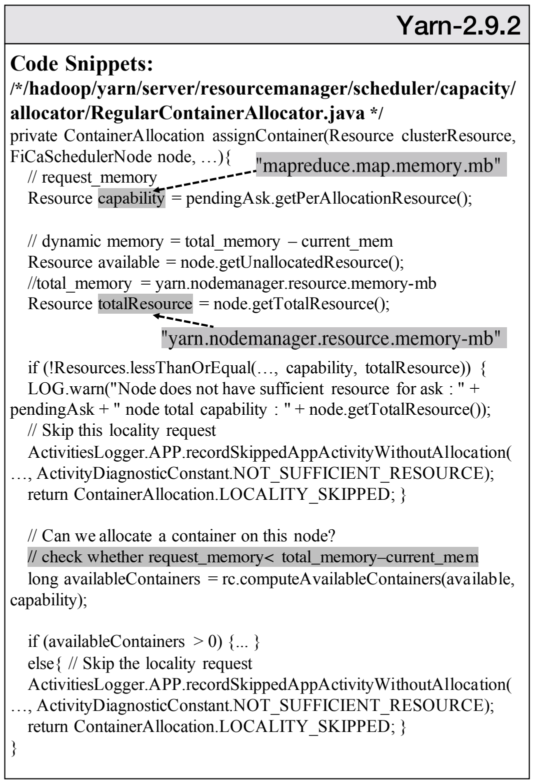
"Hbase.thrift.security.qop"

"Hadoop.security.authorization"

Description: If qop is not null, then it requires “hadoop.security.authorization" not to be set as SIMPLE, otherwise an exception will be raised. In other words, **the logical constraint is (hbase.thrift.security.qop == null) || (hadoop.security.authentication != SIMPLE)**.

Figure 8. Example of Logic Constraints.

Dynamic Constriant. Compared with some constraints checked statically, some constraints are checked dynamically and they are mostly resource-related dependencies. We find 10 cases of dynamic constriants.


Yarn-2.9.2
"mapreduce.map.memory.mb"
"yarn.nodemanager.resource.memory-mb"

Code Snippets:
/*/hadoop/yarn/server/resourcemanager/scheduler/capacity/allocator/RegularContainerAllocator.java */

```
private ContainerAllocation assignContainer(Resource clusterResource,
FiCaSchedulerNode node, …){
   // request_memory
   Resource capability = pendingAsk.getPerAllocationResource();

   // dynamic memory = total_memory – current_mem
   Resource available = node.getUnallocatedResource();
   //total_memory = yarn.nodemanager.resource.memory-mb
   Resource totalResource = node.getTotalResource();

   if (!Resources.lessThanOrEqual(…, capability, totalResource)) {
   LOG.warn("Node does not have sufficient resource for ask : " +
pendingAsk + " node total capability : " + node.getTotalResource());
   // Skip this locality request
   ActivitiesLogger.APP.recordSkippedAppActivityWithoutAllocation(
…, ActivityDiagnosticConstant.NOT_SUFFICIENT_RESOURCE);
   return ContainerAllocation.LOCALITY_SKIPPED; }

   // Can we allocate a container on this node?
   // check whether request_memory< total_memory–current_mem
   long availableContainers = rc.computeAvailableContainers(available,
capability);

   if (availableContainers > 0) {... }
   else{ // Skip the locality request
   ActivitiesLogger.APP.recordSkippedAppActivityWithoutAllocation(
…, ActivityDiagnosticConstant.NOT_SUFFICIENT_RESOURCE);
   return ContainerAllocation.LOCALITY_SKIPPED; }
}
```

Figure 9. Example of Dynamic Constriant.

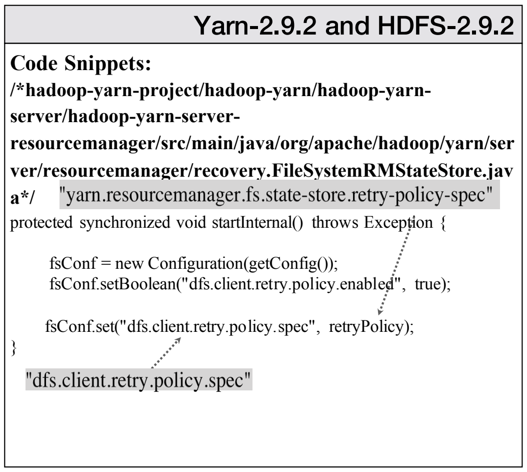
The example is `mapreduce.map.memory.mb` should be less than `yarn.nodemanager.resource.memory-mb`, and it will be checked statically and also dynamically. We can see the dynamic constriant in Figure 9.

`yarn.nodemanager.resource.memory-mb` is "Amount of physical memory, in MB, that can be allocated for containers". And `mapreduce.map.memory.mb` is the amount of memory to request from the scheduler for each map task.

When the request is ask for container on one node, it will first statically check the request rseource (`mapreduce.map.memory.mb`) should be less than the total resource (`yarn.nodemanager.resource.memory-mb`) of the node. Then it will check dynamically whether the available resource is enough. If available resource is less than request rseource, the task cannot work on this node.

4.2.2 Composition

This Dependency accounts for 16.9% of Intra-component dependencies and 31.7% of Inter-component dependencies . It either refers to default value or overwriting compositions as defined in Section 3. Failure to recognize this kind of dependency may not cause crash errors. However, it will change the values of some configuration parameters which users may not be aware of, thus leading to the system not behaving as expected from the users.


Yarn-2.9.2 and HDFS-2.9.2
"yarn.resourcemanager.fs.state-store.retry-policy-spec"
"dfs.client.retry.policy.spec"

Code Snippets:
/*hadoop-yarn-project/hadoop-yarn/hadoop-yarn-server/hadoop-yarn-server-resourcemanager/src/main/java/org/apache/hadoop/yarn/server/resourcemanager/recovery.FileSystemRMStateStore.java*/

```
protected synchronized void startInternal() throws Exception {

      fsConf = new Configuration(getConfig());
      fsConf.setBoolean("dfs.client.retry.policy.enabled", true);

     fsConf.set("dfs.client.retry.policy.spec", retryPolicy);
}
```

Figure 10. Example of Overriding Relationship.

Override . For overriding, we find two unique patterns. The first pattern is

`set(ParameterA,ParameterB)`

where ParameterB is used to override the value of ParameterA directly. The second pattern is

`set(C,ParamaterA) ; set(C,ParamaterB)`

where ParameterA and ParameterB are used to set a variable sequentially. Thus, the latter one will override the value of the first one. For better illustrations, we have shown the example codes for these patterns, which are shown in Figure 10.

Default Value . For default value relationship, we find 4 unique patterns. The first pattern is

```
<name>ParameterA</name>
<value>${ParameterB}</value>
```

where the ParameterB is used as the default value for ParameterA in the configuration file. The second pattern is

`get(ParameterA, MathFunc(ParameterB))`

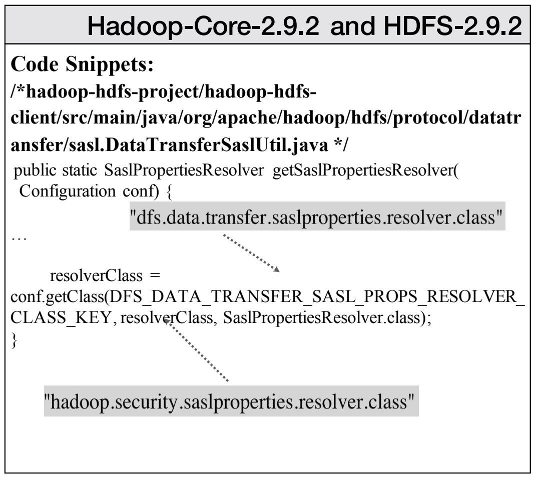
. In the *get* function, if the value of ParameterA is not available, then the value of ParameterB will be provided. Similarly, here *MathFunc* means ParameterB may be added or multiplied by some constant value. The third pattern is

```
if(notset(ParameterA))
set(ParameterA,ParameterB)
```

. The system first judges whether the value of ParameterA is available and if it is not, ParameterB will be used to set the value of ParameterA. The fourth pattern is

```
if(notset(ParameterA)) use ParameterB
else use ParameterA
```

If the the value of ParameterA is not available, the value of ParameterB will be used. Otherwise, the value of ParameterA is used. For better illustrations, we have shown the example codes for these patterns, which are shown in Figure 11.

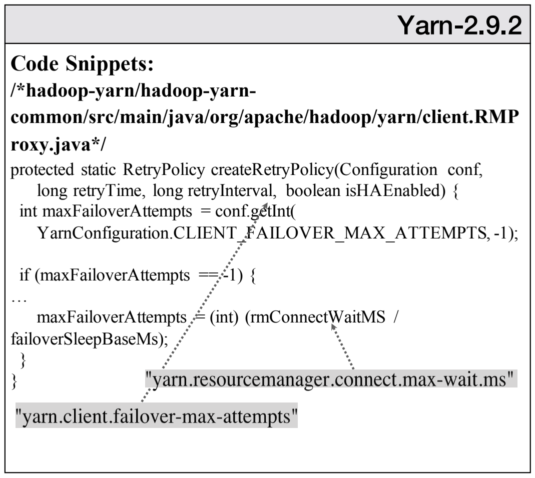

Hadoop-Core-2.9.2 and HDFS-2.9.2

Code Snippets:

/*hadoop-hdfs-project/hadoop-hdfs-client/src/main/java/org/apache/hadoop/hdfs/protocol/datatransfer/sasl.DataTransferSaslUtil.java */

```
 public static SaslPropertiesResolver  getSaslPropertiesResolver(
  Configuration  conf) {
                    "dfs.data.transfer.saslproperties.resolver.class"
…
      resolverClass  =
conf.getClass(DFS_DATA_TRANSFER_SASL_PROPS_RESOLVER_
CLASS_KEY, resolverClass,  SaslPropertiesResolver.class);
}

     "hadoop.security.saslproperties.resolver.class"
```

(a) Default Value Dependency


Yarn-2.9.2

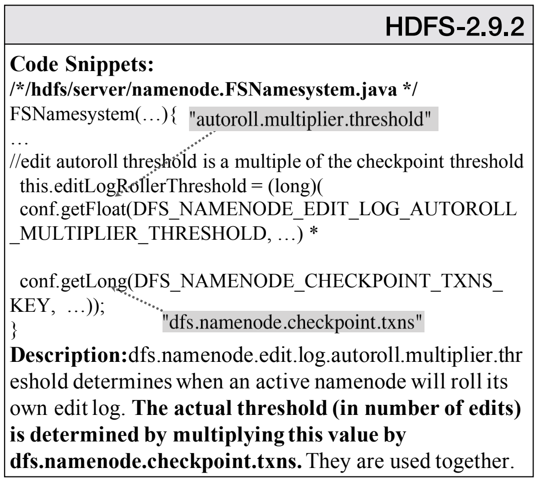
Code Snippets:

/*hadoop-yarn/hadoop-yarn-common/src/main/java/org/apache/hadoop/yarn/client.RMProxy.java*/

```
protected  static  RetryPolicy  createRetryPolicy(Configuration   conf,
     long retryTime,  long retryInterval,  boolean isHAEnabled)  {
  int maxFailoverAttempts  = conf.getInt(
     YarnConfiguration.CLIENT_FAILOVER_MAX_ATTEMPTS, -1);

  if (maxFailoverAttempts   == -1) {
…
     maxFailoverAttempts  = (int)  (rmConnectWaitMS   /
failoverSleepBaseMs);
  }
}                          "yarn.resourcemanager.connect.max-wait.ms"
 "yarn.client.failover-max-attempts"
```

(b) Default Value Dependency

Figure 11. Example of Default Value Relationship.


HDFS-2.9.2

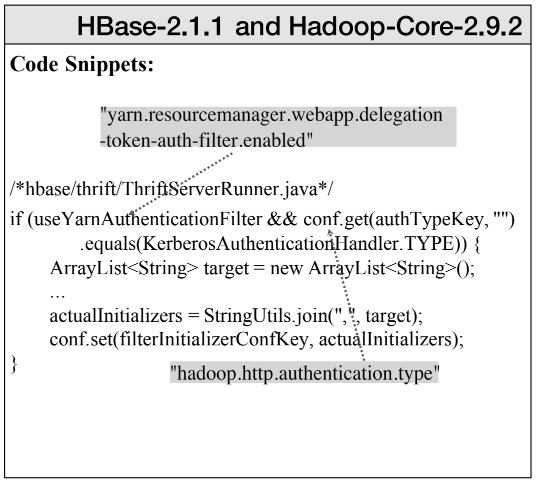
Code Snippets:

/*/hdfs/server/namenode.FSNamesystem.java */

```
FSNamesystem(…){  "autoroll.multiplier.threshold"
…
//edit autoroll threshold is a multiple of the checkpoint threshold
  this.editLogRollerThreshold = (long)(
  conf.getFloat(DFS_NAMENODE_EDIT_LOG_AUTOROLL
_MULTIPLIER_THRESHOLD, …) *

  conf.getLong(DFS_NAMENODE_CHECKPOINT_TXNS_
KEY,  …));
                    "dfs.namenode.checkpoint.txns"
}
```

Description:dfs.namenode.edit.log.autoroll.multiplier.threshold determines when an active namenode will roll its own edit log. **The actual threshold (in number of edits) is determined by multiplying this value by dfs.namenode.checkpoint.txns.** They are used together.

(a) Use-together Dependency


HBase-2.1.1 and Hadoop-Core-2.9.2

Code Snippets:

```
              "yarn.resourcemanager.webapp.delegation
              -token-auth-filter.enabled"

/*hbase/thrift/ThriftServerRunner.java*/
if (useYarnAuthenticationFilter && conf.get(authTypeKey, "")
         .equals(KerberosAuthenticationHandler.TYPE)) {
     ArrayList<String> target = new ArrayList<String>();
     …
     actualInitializers = StringUtils.join(",", target);
     conf.set(filterInitializerConfKey, actualInitializers);
}
                    "hadoop.http.authentication.type"
```

(b) Use-together Dependency

Figure 12. Example of Used Together Relationship.

4.2.3 Used together

This dependency accounts for 8.4% of total Intra-component dependencies and 6.3% of total Inter-component dependencies . Configurations which are involved in this dependency function together in the program. Thus, failure to recognize this dependency may cause the disability of some functionality from the systems.

There is one unique code pattern we have found for this kind of dependency.The pattern is

```
Result = ParameterA op ParameterB
```

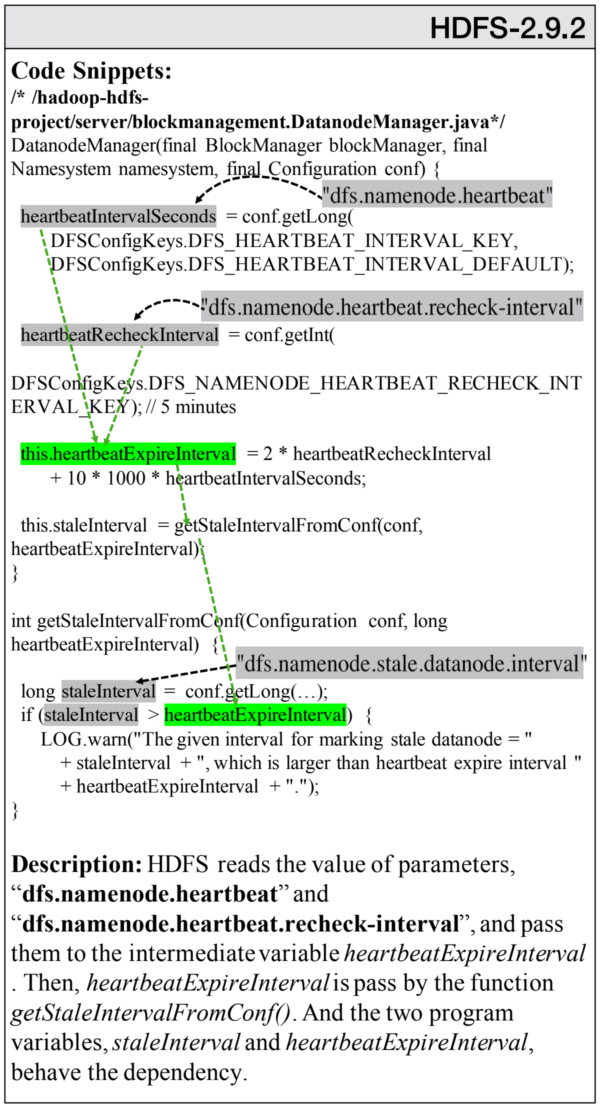
where *op* denotes binary operators like $*$ and $+$. For better illustrations, we have shown the example codes for this pattern in Figure 12.

4.3 Value Propagation

[Chen: This section is mainly used to discuss how to unify the code patterns from Intra-component dependencies and Inter-component dependencies (code patterns may be general nevertheless it is Intra-component dependencies or Inter-component dependencies). Also, the talk about the difference like how variable is passed.] We find two types of Value propagation for Intra-component dependencies and Inter-component dependencies . The first one is Procedure-related propagation; the second one is Message-related propagation.

Procedure-related propagation. Procedure-related propagation has four basic manifestation.

- **Straightly read from the conf file.** The target software straightly reads the value of parameters from configuration files. This type occupies the majority. For better illustrations, we have shown the example code in Figure 13.
- **Pass parameters into variables.** The target software reads the value of parameters and compose them into a variable. Then the variable will behave dependency



HDFS-2.9.2

Code Snippets:

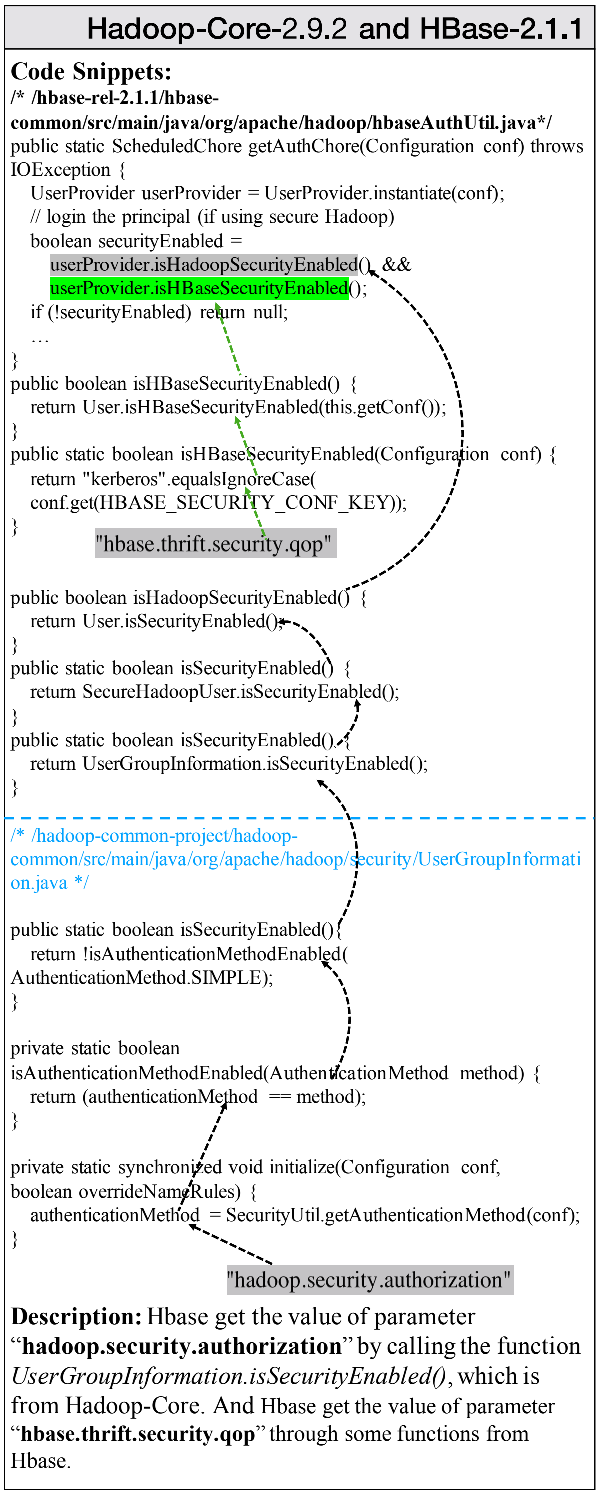
```
/* /hadoop-hdfs-
project/server/blockmanagement.DatanodeManager.java*/
DatanodeManager(final BlockManager blockManager, final
Namesystem namesystem, final Configuration conf) {
                                   "dfs.namenode.heartbeat"
  heartbeatIntervalSeconds = conf.getLong(
     DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_KEY,
     DFSConfigKeys.DFS_HEARTBEAT_INTERVAL_DEFAULT);

                          "dfs.namenode.heartbeat.recheck-interval"
  heartbeatRecheckInterval = conf.getInt(

DFSConfigKeys.DFS_NAMENODE_HEARTBEAT_RECHECK_INT
ERVAL_KEY); // 5 minutes

  this.heartbeatExpireInterval = 2 * heartbeatRecheckInterval
     + 10 * 1000 * heartbeatIntervalSeconds;

  this.staleInterval = getStaleIntervalFromConf(conf,
heartbeatExpireInterval);
}

int getStaleIntervalFromConf(Configuration conf, long
heartbeatExpireInterval) {
                              "dfs.namenode.stale.datanode.interval"
  long staleInterval = conf.getLong(…);
  if (staleInterval > heartbeatExpireInterval) {
    LOG.warn("The given interval for marking stale datanode = "
      + staleInterval + ", which is larger than heartbeat expire interval "
      + heartbeatExpireInterval + ".");
}
```

Description: HDFS reads the value of parameters, "**dfs.namenode.heartbeat**" and "**dfs.namenode.heartbeat.recheck-interval**", and pass them to the intermediate variable *heartbeatExpireInterval* . Then, *heartbeatExpireInterval* is pass by the function *getStaleIntervalFromConf()*. And the two program variables, *staleInterval* and *heartbeatExpireInterval*, behave the dependency.

Figure 13. Example of value-prepagation.

with other parameter. For better illustrations, we have shown the example code in Figure 13.

- **Pass parameters by functions in component.** The target software reads the value of parameters from configuration files and passes parameters by functions in component. For better illustrations, we have shown the example code in Figure 13 and Figure 14.
- **Pass parameters by functions out of component (API calls).** The value of parameters are passed in the way, where one component calls functions out of the component (API calls). For better illustrations, we have shown the example code in Figure 14.



Hadoop-Core-2.9.2 and HBase-2.1.1

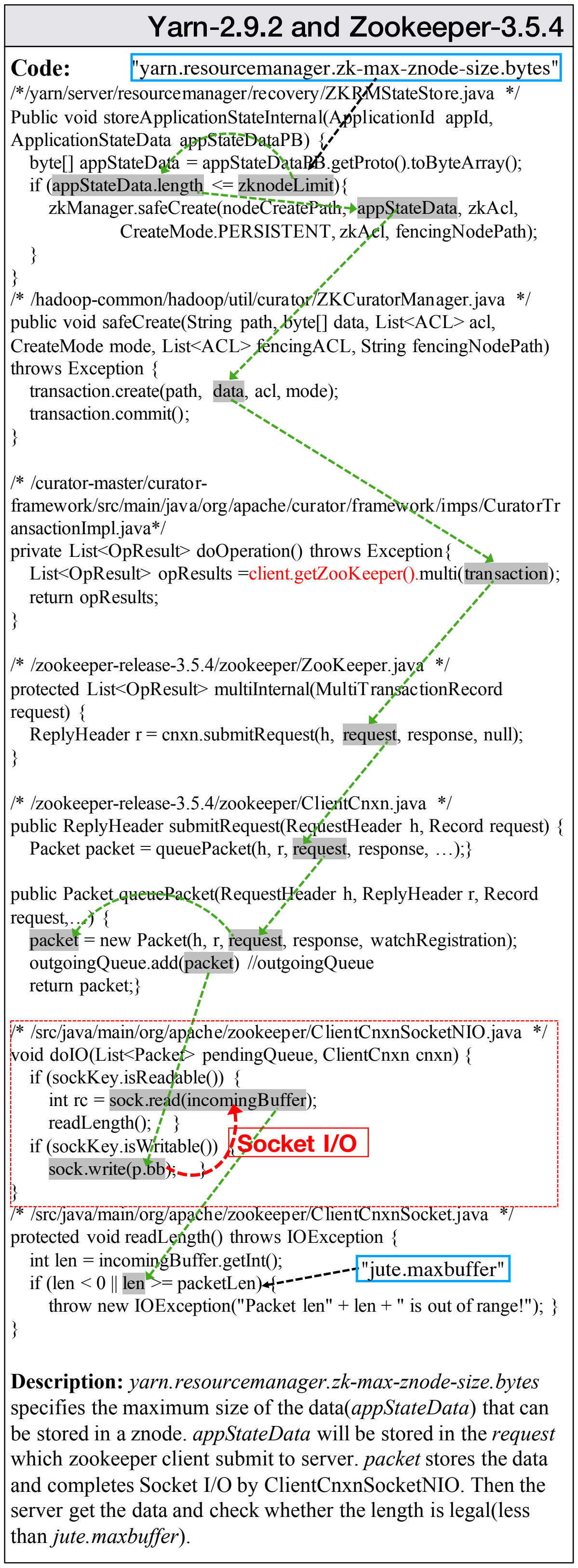
Code Snippets:

```
/* /hbase-rel-2.1.1/hbase-
common/src/main/java/org/apache/hadoop/hbaseAuthUtil.java*/
public static ScheduledChore getAuthChore(Configuration conf) throws
IOException {
   UserProvider userProvider = UserProvider.instantiate(conf);
   // login the principal (if using secure Hadoop)
   boolean securityEnabled =
      userProvider.isHadoopSecurityEnabled() &&
      userProvider.isHBaseSecurityEnabled();
   if (!securityEnabled) return null;
   …
}
public boolean isHBaseSecurityEnabled() {
   return User.isHBaseSecurityEnabled(this.getConf());
}
public static boolean isHBaseSecurityEnabled(Configuration conf) {
   return "kerberos".equalsIgnoreCase(
   conf.get(HBASE_SECURITY_CONF_KEY));
}
              "hbase.thrift.security.qop"

public boolean isHadoopSecurityEnabled() {
   return User.isSecurityEnabled();
}
public static boolean isSecurityEnabled() {
   return SecureHadoopUser.isSecurityEnabled();
}
public static boolean isSecurityEnabled() {
   return UserGroupInformation.isSecurityEnabled();
}
----------------------------------------------------------------
/* /hadoop-common-project/hadoop-
common/src/main/java/org/apache/hadoop/security/UserGroupInformati
on.java */

public static boolean isSecurityEnabled(){
   return !isAuthenticationMethodEnabled(
AuthenticationMethod.SIMPLE);
}

private static boolean
isAuthenticationMethodEnabled(AuthenticationMethod method) {
   return (authenticationMethod == method);
}

private static synchronized void initialize(Configuration conf,
boolean overrideNameRules) {
   authenticationMethod = SecurityUtil.getAuthenticationMethod(conf);
}
                          "hadoop.security.authorization"
```

Description: Hbase get the value of parameter "**hadoop.security.authorization**" by calling the function *UserGroupInformation.isSecurityEnabled()*, which is from Hadoop-Core. And Hbase get the value of parameter "**hbase.thrift.security.qop**" through some functions from Hbase.

Figure 14. Example of value-prepagation.

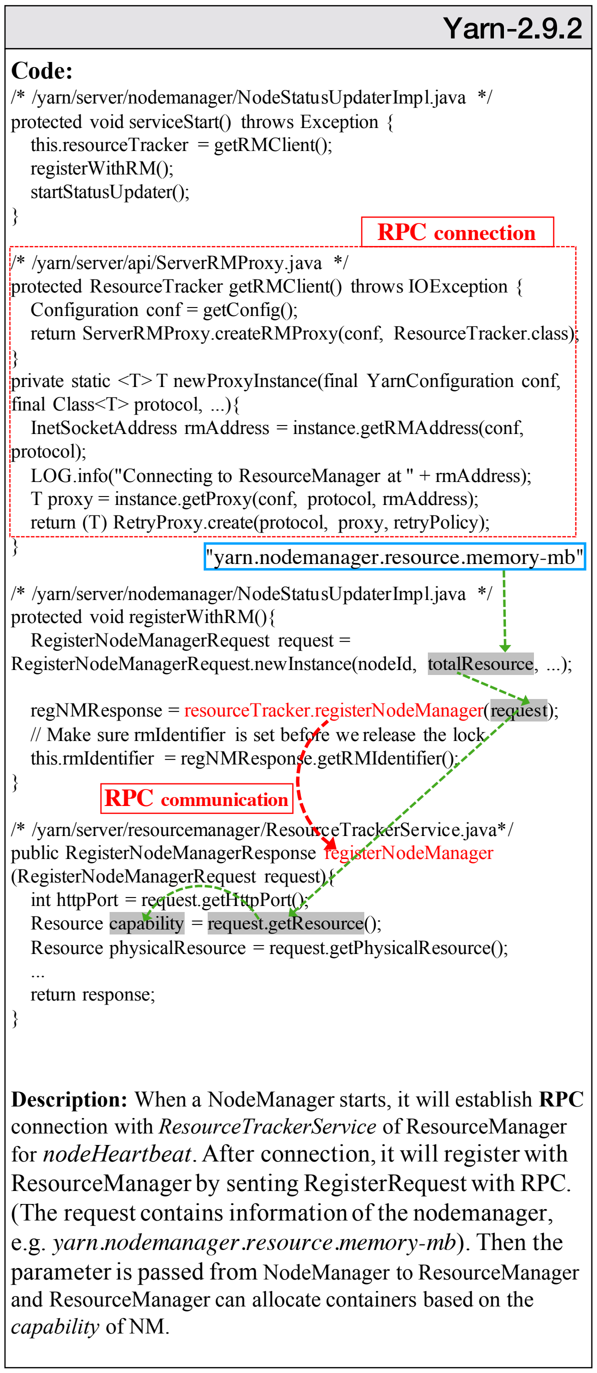
Message-related propagation. Message-related propagation passes the information of parameters by Socket I/O or Remote Procedure Call. Then the parameters are passed across components. For better illustrations, we have shown the example code in Figure 15 and Figure 16.


Yarn-2.9.2 and Zookeeper-3.5.4
Code:
"yarn.resourcemanager.zk-max-znode-size.bytes"
Socket I/O
"jute.maxbuffer"

```
/*/yarn/server/resourcemanager/recovery/ZKRMStateStore.java */
Public void storeApplicationStateInternal(ApplicationId appId,
ApplicationStateData appStateDataPB) {
   byte[] appStateData = appStateDataPB.getProto().toByteArray();
   if (appStateData.length <= zknodeLimit){
      zkManager.safeCreate(nodeCreatePath, appStateData, zkAcl,
                CreateMode.PERSISTENT, zkAcl, fencingNodePath);
   }
}
/* /hadoop-common/hadoop/util/curator/ZKCuratorManager.java */
public void safeCreate(String path, byte[] data, List<ACL> acl,
CreateMode mode, List<ACL> fencingACL, String fencingNodePath)
throws Exception {
   transaction.create(path, data, acl, mode);
   transaction.commit();
}

/* /curator-master/curator-
framework/src/main/java/org/apache/curator/framework/imps/CuratorTr
ansactionImpl.java*/
private List<OpResult> doOperation() throws Exception{
   List<OpResult> opResults =client.getZooKeeper().multi(transaction);
   return opResults;
}

/* /zookeeper-release-3.5.4/zookeeper/ZooKeeper.java */
protected List<OpResult> multiInternal(MultiTransactionRecord
request) {
   ReplyHeader r = cnxn.submitRequest(h, request, response, null);
}

/* /zookeeper-release-3.5.4/ClientCnxn.java */
public ReplyHeader submitRequest(RequestHeader h, Record request) {
   Packet packet = queuePacket(h, r, request, response, …);}

public Packet queuePacket(RequestHeader h, ReplyHeader r, Record
request,…) {
   packet = new Packet(h, r, request, response, watchRegistration);
   outgoingQueue.add(packet) //outgoingQueue
   return packet;}

/* /src/java/main/org/apache/zookeeper/ClientCnxnSocketNIO.java */
void doIO(List<Packet> pendingQueue, ClientCnxn cnxn) {
   if (sockKey.isReadable()) {
      int rc = sock.read(incomingBuffer);
      readLength(); }
   if (sockKey.isWritable()) {
      sock.write(p.bb); }
}
/* /src/java/main/org/apache/zookeeper/ClientCnxnSocket.java */
protected void readLength() throws IOException {
   int len = incomingBuffer.getInt();
   if (len < 0 || len >= packetLen){
      throw new IOException("Packet len" + len + " is out of range!"); }
}
```

Description: *yarn.resourcemanager.zk-max-znode-size.bytes* specifies the maximum size of the data(*appStateData*) that can be stored in a znode. *appStateData* will be stored in the *request* which zookeeper client submit to server. *packet* stores the data and completes Socket I/O by ClientCnxnSocketNIO. Then the server get the data and check whether the length is legal(less than *jute.maxbuffer*).

Figure 15. Example of value-prepagation.



```
/* /yarn/server/nodemanager/NodeStatusUpdaterImpl.java */
protected void serviceStart() throws Exception {
   this.resourceTracker = getRMClient();
   registerWithRM();
   startStatusUpdater();
}
/* /yarn/server/api/ServerRMProxy.java */
protected ResourceTracker getRMClient() throws IOException {
   Configuration conf = getConfig();
   return ServerRMProxy.createRMProxy(conf, ResourceTracker.class);
}
private static <T> T newProxyInstance(final YarnConfiguration conf,
final Class<T> protocol, ...){
   InetSocketAddress rmAddress = instance.getRMAddress(conf,
protocol);
   LOG.info("Connecting to ResourceManager at " + rmAddress);
   T proxy = instance.getProxy(conf, protocol, rmAddress);
   return (T) RetryProxy.create(protocol, proxy, retryPolicy);
}

/* /yarn/server/nodemanager/NodeStatusUpdaterImpl.java */
protected void registerWithRM(){
   RegisterNodeManagerRequest request =
RegisterNodeManagerRequest.newInstance(nodeId, totalResource, ...);

   regNMResponse = resourceTracker.registerNodeManager(request);
   // Make sure rmIdentifier is set before we release the lock
   this.rmIdentifier = regNMResponse.getRMIdentifier();
}

/* /yarn/server/resourcemanager/ResourceTrackerService.java*/
public RegisterNodeManagerResponse registerNodeManager
(RegisterNodeManagerRequest request){
   int httpPort = request.getHttpPort();
   Resource capability = request.getResource();
   Resource physicalResource = request.getPhysicalResource();
   ...
   return response;
}
```

Description: When a NodeManager starts, it will establish **RPC** connection with *ResourceTrackerService* of ResourceManager for *nodeHeartbeat*. After connection, it will register with ResourceManager by senting RegisterRequest with RPC. (The request contains information of the nodemanager, e.g. *yarn.nodemanager.resource.memory-mb*). Then the parameter is passed from NodeManager to ResourceManager and ResourceManager can allocate containers based on the *capability* of NM.

Figure 16. Example of value-prepagation.

4.4 Implicit Patterns

TODO: Timing is one such example.

Further, we also analyze the dependent cases where we could not get any code patterns. [Chen: this section needs more efforts and do not have results yet].

Check	Early Check	Runtime Check	No Check
Control dependency	0	0	0
Value dependency	0	0	0
→ Constraints	0	0	0
→ Composition	0	0	0
→ Used together	0	0	0
→ Resource competition	0	0	0

Table 4. Checking of Violations for Intra-component dependencies and Inter-component dependencies .

Handling	Exception	Overruling with Logging	Overruling without Lo
Constraints	37	9	30
→ Numeric	12+9	6+2	5+25
→ Membership	2+2	0+0	0+0
→ Logic	10+2	1+0	0+0

Table 5. Handling of Violations for Intra-component dependencies and Inter-component dependencies .

Handling	Crashing	Logging	None
Constraints	34	15	36
→ Numeric	9+9	0+12	24+12
→ Membership	2+2	0+0	0+0
→ Logic	10+2	1+2	0+0

Table 6. Manifestations of Violations for Intra-component dependencies and Inter-component dependencies .

5 Checking

TODO: Answer the following three research questions:

- Do software check the dependencies?
- When do software check the dependencies (do they do it during initialization or at the runtime)?

In this section, we focus on studying whether those Intra-component dependencies and Inter-component dependencies are checked by the software. In addition, for those checked dependencies, we study whether they are checked during initialization time or checked during runtime. The collected numbers for each dependency type is summarized in Table 4. [Chen: to be filled later].

6 Handling Violation

TODO: How does software handle the violations of configuration dependencies?

In this section, we analyze in detail how the system behaves if the configuration dependency is violated. More specifically, we will analyze how the system behaves internally for the violation (i.e. how the system handles the violation) which we denote as violation handling and how the system behaves externally for the violation (i.e. what the users observe from the programs with violations) which we denote as manifestation of violations. In the next sections, we will describe the internal and external behaviors in detail one by one.

We study how the programs handle when the configuration dependency is violated. Generally, there are five kinds of behaviors. The first is exception which means if the dependency is violated, the program will throw an exception and crash. The second is overruling without logging which means the program will override the values set by user and keep the program running without telling users the configuration values are changed. The third is overruling with logging which means the program will override the values while notifying users at the same time. The fourth is logging which means the program will simply log the violation while doing nothing to solve it. The fifth is ignoring which means the program does nothing to neither solve the violation nor log the event. For all the Intra-component dependencies and Inter-component dependencies , we have analyzed how they are handled when the relations are violated. The statistics are presented in Table 5[Chen: fill later].

We further analyze how the system behave externally in case of violations of configuration dependencies. There are three external manifestations in total. The first is crashing, the second is printing logging messages and the last is no symptoms. For all the Intra-component dependencies and Inter-component dependencies , we have analyzed their external manifestations and the results are presented in Table 6.[Chen: fill later]

7 Feedback

TODO: Do software provide good feedback when the constraints are violated to help users resolve the problems efficiently? In this section, we study when the configuration dependencies are violated, whether the programs will give enough feedback for users to solve it. To be more specific, we consider there are four cases of feedback.

The first case is that software provides enough feedback for users to solve the problem. For enough feedback, we mainly refer to that the software logs contains the full name or substring of all the involved configurations and clearly give out the dependency relation. For example, if the logs say that configuration A should be larger than configuration B, we consider it as providing enough feedback for users.

The second case is that software provides useful feedback for users. For useful feedback, we refer to that some (not all of) involved configurations are mentioned in the logs when configuration dependencies are violated. So the relationship is not clearly identified in the logs. But we can also know there is something wrong with the involved configurations. So it is useful to solve the dependency problem.

The third case is that software provides inadequate feedback for users. For inadequate, we refer to that no involved configuration is not mentioned in the logs. In this case, it's

Feedback	Enough	Useful	Inadequate	None
Constraints	25	16	18	43
→ Numeric	16+0	1+10	2+11	17+26
→ Membership	0+1	1+0	1+1	0+0
→ Logic	5+3	4+0	2+1	0+0

Table 7. Feedback from software for violations of Intra-component dependencies and Inter-component dependencies .

difficult for users to find the root cause, even the involved configuration. However, the inadequate logs might also help diagnosing with some diagnosis tools.

The fourth case is that software provides no feedback at all. For this case, we refer to that the system either prints no log or simply throws an exception without telling any reasons.

For all the Intra-component dependencies and Inter-component dependencies , we have analyzed their feedback and the results are summarized in Table 7.[Chen: filled later].

8 Tooling

TODO: Build a tool that can discover new configuration dependencies based on the code pattern we studied.

8.1 Design and Implementation

8.2 Use Cases

TODO: We have discussed a few uses cases, including:

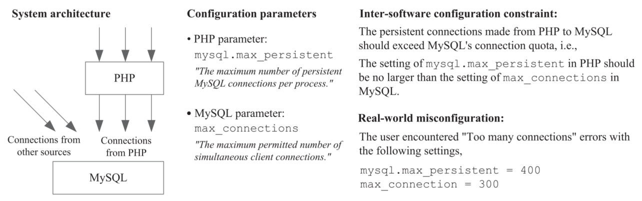
- enhancing documentation
- check violations in existing configuration files
- improving configuration design

9 Related Work

TODO: This seems to be directly moved from CQ's old project report. Let's keep it as is.

We are the first to provide a comprehensive study to understand different kinds of configuration issues both within each application and across multiple applications. Therefore, there is no significant previous work directly related to our study. However, some works have considered configuration issues for limited relationships or in different domains. In this section we briefly discuss these works.

One recent work related to ours is Mohammed et al. [?]. Although they also consider cross-stack configuration errors, they do not consider configuration dependency issues. They only consider when receiving the error messages how to quickly locate the exact layer where the error happens when the system is running in multi-layer environment. Besides this work, Zhang et al. [?] study the configuration dependency issues. However, they only consider dependencies within one application while we also consider multiple applications. Another paper related to our work is by



"The maximum number of persistent MySQL connections per process."

"The maximum permitted number of simultaneous client connections."

The persistent connections made from PHP to MySQL should exceed MySQL's connection quota, i.e.,

The setting of `mysql.max_persistent` in PHP should be no larger than the setting of `max_connections` in MySQL.

The user encountered "Too many connections" errors with the following settings,

```
mysql.max_persistent = 400
max_connection = 300
```

Figure 17. A real-world example of a cross-component configuration error in a LAMP-based Web server from [?].

Vinod et al. [?]. They consider configuration dependency issues in multiple software component stacks. However, the dependency they care is very limited. They consider two parameters are related only if their values are the same or one value is the substring of the other. Note that our goal is to conduct a comprehensive study of configuration dependencies. Another difference is that, their work has only considered enterprise applications, for which they have a running system in a organization and also error logs of that system. However, in our work we rely on open source system and crowd sourcing data to detect dependencies. Due to these reasons, their system can not be directly used for our problem. In addition, Holl et al. [?] have considered multiple product line system of system services (SoS). In such services, various kinds of dependencies exist between product lines. Like our work, they also consider configuration dependencies between different components. However, they only consider some calling dependencies which means one component tries to call one function from the other component by setting the corresponding configuration value. Thus, from our perspective, they are still on the level of dependencies between different applications (i.e., how different product lines rely on each other), instead of investigating the dependencies among different configuration parameters.

References

[] Gerald Holl, Daniel Thaller, Paul Grünbacher, and Christoph Elsner. 2012. Managing Emerging Configuration Dependencies in Multi Product Lines. In *Proceedings of the 6th International Workshop on Variability Modelling of Software-Intensive Systems (VAMOS'12)*. Leipzig, Germany.

[] Vinod Ramachandran, Manish Gupta, Manish Sethi, and Soudip Roy Chowdhury. 2009. Determining Configuration Parameter Dependencies via Analysis of Configuration Data from Multi-tiered Enterprise Applications. In *Proceedings of the 6th International Conference on Autonomic Computing and Communications (ICAC'09)*. Barcelona, Spain.

[] Mohammed Sayagh, Noureddine Kerzazi, and Bram Adams. 2017. On Cross-stack Configuration Errors. In *Proceedings of the 39th International Conference on Software Engineering (ICSE'17)*. Buenos Aires, Argentina.

[] Tianyin Xu and Yuanyuan Zhou. 2015. Systems Approaches to Tackling Configuration Errors: A Survey. *ACM Computing Surveys (CSUR)* 47, 4 (July 2015).

[] Jiaqi Zhang, Lakshmi Renganarayana, Xiaolan Zhang, Niyu Ge, Vasanth Bala, Tianyin Xu, and Yuanyuan Zhou. 2014. EnCore: Exploiting

System Environment and Correlation Information for Misconfiguration Detection. In *Proceedings of the 19th International Conference on Architecture Support for Programming Languages and Operating Systems (ASPLOS'14)*. Salt Lake City, UT.