Heading off Correlated Failures through Independence-as-a-Service

Ennan Zhai, Ruichuan Chen, David Isaac Wolinsky, and Bryan Ford

Presenter: Ron Wright
Motivation

• Cloud services depend on redundancy to ensure high reliability
• However, components that appear to be independent may share subtle dependencies, leading to unexpected correlated failures
• Redundant systems may contain risk groups (RGs), or sets of components that can cause a service outage if all the components fail simultaneously
What Can Go Wrong?

The availabilities of racks A, B, and C all depend on the availability of the aggregation switch. This common dependency introduced an unexpected RG.
Documented Examples

• Amazon AWS
  – One glitch on an EBS server disabled entire service across Amazon’s US-East region
  – This, in turn, caused correlated failures among EC2 instances utilizing the EBS server, which disabled applications designed for EC2 redundancy

• Google Storage
  – “Close to 37% of failures are truly correlated”
  – No tools to identify failure correlations systematically

• iCloud
  – A storm in Dublin disabled both Amazon and Microsoft clouds in that region for hours
Independence-as-a-Service (INDaaS)

• Architecture that proactively collects and audits structural dependency data to evaluate independence of redundant systems before any failures occur
  – *Dependency acquisition modules* collect dependency data
  – *Auditing modules* quantify independence of redundant systems and pinpoint common dependencies that may cause correlated failures
Main Contributions

1. Evaluates independence of redundant systems before or during deployment
2. Provides fault graph analysis to enable the evaluation of dependencies at multiple levels of detail
3. Uses scalable fault graph analysis
4. Supports efficient PIA through private set intersection cardinality
5. Provides realistic case studies with a prototype implementation
Architecture

Auditing Client, Alice

Step 1

Independence Auditing (Agent-Side)

Step 6

Step 5

Auditing Agent

Step 2

Independence Auditing (DataSource-Side)

Step 4

Step 3

Dependency Acquisition

Step 3

Dependency Data Source 1

Step 5

Independence Auditing (DataSource-Side)

Step 2

Dependency Data Source 2
Dependency Data Representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Dependency Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td><code>&lt;src=&quot;S&quot; dst=&quot;D&quot; route=&quot;x,y,z&quot;/&gt;</code></td>
</tr>
<tr>
<td>Hardware</td>
<td><code>&lt;hw=&quot;H&quot; type=&quot;T&quot; dep=&quot;x&quot;/&gt;</code></td>
</tr>
<tr>
<td>Software</td>
<td><code>&lt;pgm=&quot;S&quot; hw=&quot;H&quot; dep=&quot;x,y,z&quot;/&gt;</code></td>
</tr>
</tbody>
</table>

Network dependencies of S1 and S2:
- `<src="S1" dst="Internet" route="Tor1,Core1"/>`
- `<src="S1" dst="Internet" route="Tor1,Core2"/>`
- `<src="S2" dst="Internet" route="Tor1,Core1"/>`
- `<src="S2" dst="Internet" route="Tor1,Core2"/>`

Hardware dependencies of S1 and S2:
- `<hw="S1" type="CPU" dep="Intel(R)X5550@2.6GHz"/>`
- `<hw="S1" type="Disk" dep="SED900"/>`
- `<hw="S2" type="CPU" dep="Intel(R)X5550@2.6GHz"/>`
- `<hw="S2" type="Disk" dep="SED900"/>`

Software dependencies of S1 and S2:
- `<pgm="QueryEngine1" hw="S1" dep="libc6,libgccl"/>`
- `<pgm="Riak1" hw="S1" dep="libc6,libsvn1"/>`
- `<pgm="QueryEngine2" hw="S2" dep="libc6,libgccl"/>`
- `<pgm="Riak2" hw="S2" dep="libc6,libsvn1"/>`
Structural Independence Auditing (SIA)

• Assumes data sources are willing to share full dependency data with each other
• Involves generating a dependency graph, finding and ranking risk groups, and generating an audit report
Dependency Graph

OR gate: a failure propagates upwards, if any of the subsidiary components fails.
AND gate: a failure propagates upwards, only if all of the subsidiary components fail.

(a) Component-set
- Data Source E1
  - A1
  - A2
  - A3
  E1's component-set = \{A1, A2\}

(b) Fault-set
- Data Source E2
  - A1
  - A2
  - A3
  E2's component-set = \{A2, A3\}

Pr(Redundancy fails)

Pr(E1 fails) = 0.1
Pr(E2 fails) = 0.2
Pr(A1 fails) = 0.3
Pr(A2 fails) = 0.2
Pr(A3 fails) = 0.3

E1's fault-set = \{Pr(A1 fails)=0.1, Pr(A2 fails)=0.2\}
E2's fault-set = \{Pr(A2 fails)=0.2, Pr(A3 fails)=0.3\}

1. Redundancy deployment fails
2. S1 fails, S2 fails
3. S1's hardware fails, S1's network fails, S1's software fails, S2's software fails, S2's network fails, S2's hardware fails
4. CPU1 fails, Disk1 fails, CPU2 fails, Disk2 fails
5. Path1 fails, Path2 fails
6. libsvn1 fails, libc6 fails, libgccl fails

3/10/2015
Heading off Correlated Failures through Independence-as-a-Service
Risk Groups in Dependency Graphs

OR gate: a failure propagates upwards, if any of the subsidiary components fails.
AND gate: a failure propagates upwards, only if all of the subsidiary components fail.

(a) Component-set
E1’s component-set = \{A1, A2\}
E2’s component-set = \{A2, A3\}

(b) Fault-set
E1’s fault-set = \{Pr(A1 fails)=0.1, Pr(A2 fails)=0.2\}
E2’s fault-set = \{Pr(A2 fails)=0.2, Pr(A3 fails)=0.3\}

(c) Fault graph

\{A1, A3\}
\{A1, A2\}
\{A1, A2, A3\}
\{A2\}
\{A2, A3\}

\{TOR1 fails\}
\{Core1 fails, Core2 fails\}

3/10/2015
Heading off Correlated Failures through Independence-as-a-Service
Algorithms for Finding Risk Groups

• Minimal RG algorithm
  – Directly computes minimal RGs using reverse breadth-first traversal
  – Pros
    • Results are exact
  – Cons
    • Algorithm is NP hard!

• Failure sampling algorithm
  – Randomly assigns 0s and 1s to basic events to test for deployment failure and generate the appropriate RGs
  – Pros
    • Linear time complexity
  – Cons
    • Non-deterministic
    • No guarantee that any RG is minimal
Ranking Risk Groups

• Size-based ranking
  – Ranks RGs based on the number of components in each RG
  – The smaller the number of components in the RG, the higher the rank

• Failure probability ranking
  – Ranks RGs based on their relative importance, $I_c = \frac{\Pr(C)}{\Pr(T)}$
    • $\Pr(C)$ represents probability of any given failure event $C$
    • $\Pr(T)$ represents probability of any given failure event $T$
  – $\Pr(T)$ computed by inclusion-exclusion principle involving all minimal RGs of $T$
Pr(T) = Pr(A1 and A3 fail, or A2 fails) = 0.1 \cdot 0.3 + 0.2 - 0.1 \cdot 0.3 \cdot 0.2 = 0.224

\[ \Pr(\text{Redundancy fails}) \]

\[ \Pr(\text{E1 fails}) \quad \Pr(\text{E2 fails}) \]

\[ \Pr(\text{A1 fails}) = 0.1 \quad \Pr(\text{A2 fails}) = 0.2 \quad \Pr(\text{A3 fails}) = 0.3 \]

- \[ I_{A2 \text{ fails}} = \Pr(\text{A2 fails}) / \Pr(T) = 0.2 / 0.224 = \underline{0.8929} \]
- \[ I_{A1 \text{ fails}, A3 \text{ fails}} = \Pr(\text{A1 fails, A3 fails}) / \Pr(T) = 0.1 \cdot 0.3 / 0.224 = \underline{0.1339} \]

Therefore, the RG \{A2 fails\} is ranked higher than the RG \{A1 fails, A3 fails\}. 
Generating the Audit Report

• Let R denote a specific redundancy deployment
• Let \( c_i \) denote the \( i \)-th RG in R’s RG-ranking list
• Size-based ranking algorithm
  \[ \text{indep}(R) = \sum_{i=1}^{n} \text{size}(c_i) \]
• Failure probability ranking algorithm
  \[ \text{indep}(R) = \sum_{i=1}^{n} I_{c_i} \]
• Computed independence scores, returned to the client, can be used to choose the most independent deployment for a particular service, for example
Private Independence Auditing (PIA)

- Allows auditing to take place, even across two cloud providers unwilling to share full dependency data with each other
- Trust assumptions
  1. Auditing clients may be malicious and would like to know as much as possible about the providers’ dependency data
  2. Cloud providers and auditing agents are honest but curious
  3. No collusion among cloud providers and auditing agents
Jaccard similarity

• Let $S_i$ denote the $i$-th dataset

$$J(S_0, \ldots, S_{k-1}) = \frac{|S_0 \cap \ldots \cap S_{k-1}|}{|S_0 \cup \ldots \cup S_{k-1}|}$$

• Above computation useful for small datasets

• Low similarity for $J$ close to 0, high similarity for $J$ close to 1, significant correlation for $J$ greater than or equal to 0.75
MinHash

- An approximation to Jaccard similarity, which is useful for large datasets
- Let $h^{(1)}(\cdot), \ldots, h^{(m)}(\cdot)$ denote $m$ different hash functions
- MinHash constructs a vector $\left\{ h_{\text{min}}^{(i)}(S) \right\}_{i=1}^{m}$ and computes Jaccard similarity as $J(S_0, \ldots, S_{k-1}) = \frac{\delta}{m} + O\left(\frac{1}{\sqrt{m}}\right)$, where
  - $\delta$ denotes the number of datasets satisfying $h^{(i)}_{\text{min}}(S_1) = \cdots = h^{(i)}_{\text{min}}(S_{k-1})$
  - $h^{(i)}_{\text{min}}(S)$ denotes an item $e \in S$ with the smallest value $h^{(1)}(e)$
P-SOP

All parties share the above datasets with each other

Key assumptions:
- Each party encrypts the datasets using commutative encryption
- Each party permutes its dataset elements using a fixed permutation function
- All parties agree on the same hash function

$S_k = \text{the original dataset } k$

$S_k^{p_1,p_2,p_3,...} = \text{dataset } k \text{ hashed, encrypted, and permuted by } p_1; \text{ then encrypted and permuted by } p_2; \text{ then encrypted and permuted by } p_3; \text{ etc.}$
Dependency Graph & Audit Report

• Each provider first generates local dependency graph at component-set level
• Each provider *normalizes* generated component-set $S_i$ using two types of components with common correlated failures
  – Third-party routing elements (e.g., ISP routers)
    • Accessible IP addresses used as unique identifiers
  – Third-party software packages (e.g., OpenSSL)
    • Standard names plus software versions used as unique identifiers
• Report consists of rankings of Jaccard similarities
On each dependency acquisition node:
- NSDMiner used for network dependencies
- lshw used for hardware dependencies
- apt-rdepends used for software dependencies
PIA Implementation & Deployment

Heading off Correlated Failures through Independence-as-a-Service

With NetworkX

SSH links
Network Dependency Case Study

{Rack 5, Rack 29} most independent deployment

(a) Common network dependency.
Hardware Dependency Case Study

Top-ranking RGs:
1. {Server2}
2. {Switch1}
3. {Core1, Core2}
4. {VM7, VM8}

(b) Common hardware dependency.
Software Dependency Case Study

(c) Common software dependency.

<table>
<thead>
<tr>
<th>Rank</th>
<th>2-Way Redundancy</th>
<th>Jaccard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cloud2 &amp; Cloud4</td>
<td>0.1419</td>
</tr>
<tr>
<td>2</td>
<td>Cloud2 &amp; Cloud3</td>
<td>0.1547</td>
</tr>
<tr>
<td>3</td>
<td>Cloud1 &amp; Cloud4</td>
<td>0.2081</td>
</tr>
<tr>
<td>4</td>
<td>Cloud1 &amp; Cloud3</td>
<td>0.2939</td>
</tr>
<tr>
<td>5</td>
<td>Cloud3 &amp; Cloud4</td>
<td>0.3489</td>
</tr>
<tr>
<td>6</td>
<td>Cloud1 &amp; Cloud2</td>
<td>0.5059</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rank</th>
<th>3-Way Redundancy</th>
<th>Jaccard</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Cloud2 &amp; Cloud3 &amp; Cloud4</td>
<td>0.1128</td>
</tr>
<tr>
<td>2</td>
<td>Cloud1 &amp; Cloud2 &amp; Cloud4</td>
<td>0.1207</td>
</tr>
<tr>
<td>3</td>
<td>Cloud1 &amp; Cloud3 &amp; Cloud4</td>
<td>0.1353</td>
</tr>
<tr>
<td>4</td>
<td>Cloud1 &amp; Cloud2 &amp; Cloud3</td>
<td>0.1536</td>
</tr>
</tbody>
</table>
Performance Evaluation Configuration

Performance of INDaaS was evaluated on 40 workstations containing Intel Xeon Quad Core HT 3.7 GHz CPUs and 16 GB of RAM.

<table>
<thead>
<tr>
<th></th>
<th>Topology A</th>
<th>Topology B</th>
<th>Topology C</th>
</tr>
</thead>
<tbody>
<tr>
<td># switch ports</td>
<td>16</td>
<td>24</td>
<td>48</td>
</tr>
<tr>
<td># core routers</td>
<td>64</td>
<td>144</td>
<td>576</td>
</tr>
<tr>
<td># agg switches</td>
<td>128</td>
<td>288</td>
<td>1,152</td>
</tr>
<tr>
<td># ToR switches</td>
<td>128</td>
<td>288</td>
<td>1,152</td>
</tr>
<tr>
<td># servers</td>
<td>1,024</td>
<td>3,456</td>
<td>27,648</td>
</tr>
<tr>
<td>Total # of devices</td>
<td>1,344</td>
<td>4,176</td>
<td>30,528</td>
</tr>
</tbody>
</table>
SIA Performance Evaluation Results

(Roughly) linear computation performance seen for failure sampling algorithm in all topologies.

Tradeoff exists between linear time complexity and logarithmic percentage of minimal RGs detected.
PIA Performance Evaluation Results

- 1024-bit keys were used for all types of encryption

\[ \text{P-SOP consistently outperforms KS computation-wise.} \]

- Large overhead difference
  - Small overhead difference

\[ \text{P-SOP (2), P-SOP (3), P-SOP (4), KS (2), KS (3), KS (4)} \]

\[ \text{Number of elements in each provider's dataset} \]

\[ \text{Total traffic sent (MB)} \]

\[ \text{Computational time (seconds)} \]

(a) Bandwidth overhead.

(b) Computational overhead.
Comparing Performance of SIA & PIA

In both cases
- P-SOP outperforms KS
- $10^6$ rounds of random sampling outperform minimal RG algorithm
- Minimal RG algorithm and KS do not scale well

In both cases
- P-SOP outperforms KS
- $10^6$ rounds of random sampling outperform minimal RG algorithm
- Minimal RG algorithm and KS do not scale well
Comments & Criticisms

• Pros
  – Risk group ranking makes it easy for users to identify potential correlated failures in deployment configurations
  – Flexible in allowing cloud providers to decide whether to share their dependency data with other cloud providers

• Cons
  – For large enough deployments, in some cases, failure sampling algorithm may run longer with much fewer minimal RGs than the minimal RG algorithm
  – Cannot be used for complex dependency acquisition
  – Trust assumptions may not hold in reality (e.g., cloud providers may behave maliciously)
  – Cannot have fault-set level dependency graphs and failure probability-based ranking without accurate failure probability information
  – INDaaS is not fault tolerant in itself (e.g., the P-SOP nodes in PIA and the auditing agent are single points of failure)
Piazza Comments & Criticisms

• Pros
  – Dependency acquisition modules are pluggable
  – Fault graphs serve as intuitive models
  – Useful for people who have no prior knowledge of correlated failures in system

• Cons
  – Only considers static dependencies
  – Failure probabilities, required by INDaaS, may be difficult to obtain, and their accuracy is questionable
  – Cloud providers may not have enough incentives to share data
  – Auditing is time-consuming
Thank you!
Minimal RG Algorithm

Start

Gather all basic events

Enqueue basic events into initially empty Q

Is Q empty?

Yes
End

No
Dequeue front event from Q

Were the event's RGs already generated?

Yes

Is input gate of event AND?

Yes

Assertion failure!

No

Yes

Add all RGs resulting from all possible combinations of Cartesian products of children’s RGs

Add all RGs from children

Does event represent basic event?

Yes
Generate RG containing only the basic event

No

Is input gate of event OR?

Yes
Enqueue all parent events into Q if any exist

No

Is input gate of event AND?

Yes

Advantage
Results are exact.

Disadvantage
Algorithm is NP hard!

No

Assertion failure!
Failure Sampling Algorithm

Start -> Gather all basic events

Add RG containing basic events assigned 1 values if top event has value 1

Enqueue basic events into initially empty Q

Is Q empty?

No -> Terminate?

Yes -> Dequeue front event from Q

Does the event have a value?

Yes -> Set value as AND of all children’s values

No -> Set value as OR of all children’s values

Randomly assign 1 or 0 as its value

Does event represent basic event?

Yes -> Add RG containing basic events assigned 1 values if top event has value 1

No -> Enqueue all parent events into Q if any exist

Is input gate of event OR?

Yes -> Start

No -> End

Is input gate of event AND?

Yes -> Gather all basic events

No -> Assertion failure!

Advantage
Linear time complexity

Disadvantages
• Non-deterministic
• No guarantee that any RG is minimal