Performance Debugging for Distributed Systems of Black Boxes

PUBLISHED IN: PROCEEDINGS OF THE 19TH ACM SOSP 2003
SIMON KINDSTRÖM, PASCAL VOGEL
2016-12-01
Contents

1. Background
2. Problem Definition
3. Research Goals
4. Proposed Solution
5. Experiments
6. Conclusion
Background

• Large-scale distributed systems are difficult to debug.

• Black box components (= software components with nontransparent inner workings) increase difficulty.

• Performance of a black box distributed system must be analyzed on system level not on component level.

• Tools for identifying performance bottlenecks without need for highly skilled experts are required.
Problem Definition

• Distributed system can be modeled as graph of communicating nodes.

• Nodes = computers; edges = connections

• External request leads to activities in the graph along a causal path.

• Assumption: latencies are caused by node traversals (no significant network delay).

Source: Aguilera et al. 2003
Research Goals

Goals

1. Find high-impact causal path patterns (= those which amount for significant latency as observed by users).

2. Identify nodes on high-impact patterns which add significant latency to the patterns.

→ Identification of performance bottlenecks.

Constraints

1. Minimal knowledge of semantics of applications.

2. No modifications to applications, messages, etc.

3. No significant impact on system performance.
Proposed Solution

1. **Collect** complete trace of all inter-node messages for a system under load.
   - Simple in theory: only timestamp, sender, receiver and call/return necessary.
   - Real-world challenges: large trace sets, hardware cost, passive network tracing.

2. **Analyze** the gathered data using one of two algorithms.
   - *Nesting algorithm*: identify causal paths by looking for nesting relationships (only works for RPC-based systems).
   - *Convolution algorithm*: uses signal processing to find causal paths (works for all message-based systems).

3. **Visualize** the results.
Proposed Solution: Nesting Algorithm

- Finds causal patterns by analyzing how calls are nested.
- **Nested property**
  Call B ↔ C is nested within call A ↔ B if A calls B and B calls C before returning to A.
- Can be inferred from timestamps.
- Only works with RPC-based communication (needs to know if message is call or return).

Source: Aguilera et al. 2003
Proposed Solution: Nesting Algorithm

1. Find call pairs in the trace.
   - A → B, B → A  →  (A, B, 1, 11)
   - B → C, C → B  →  (B, C, 3, 5)
   - B → D, D → B  →  (B, D, 7, 9)

2. Find all possible nestings of one call pair in another, and estimate the likelihood of each candidate nesting via scoring.
   - (A, B, 1, 11) encloses both (B, C, 3, 5) and (B, D, 7, 9).

3. Pick the most likely parent candidate for the causing call for each call pair.
   - Only one possible parent: (A, B, 1, 11)

4. Derive call paths from the causal relationships.
   - A → B → C; D

Source: Aguilera et al. 2003
Nesting Algorithm: Find Call Pairs

First step: find all call pairs and their possible parent call pairs.

1. procedure FindCallPairs
2. for each trace entry \((t_1, \text{CALL/RET}, \text{sender A, receiver B, callid id})\)
3.   case CALL:
4.       store \((t_1, \text{CALL, A, B, id})\) in \(T_{\text{opencalls}}\)
5.   case RETURN:
6.       find matching entry \((t_2, \text{CALL, B, A, id})\) in \(T_{\text{opencalls}}\)
7.       if match is found then
8.           remove entry from \(T_{\text{opencalls}}\)
9.           update entry with return message timestamp \(t_2\)
10.          add entry to \(T_{\text{callpairs}}\)
11.          entry.parents := \{ all callpairs \((t_3, \text{CALL, X, A, id}_2)\) in \(T_{\text{opencalls}}\) with \(t_3 < t_2\) \}
Nesting Algorithm: Score Causal Nestings

**Intermediate result:** $T_{\text{callpairs}}$ containing all call pairs in the trace and their possible parent calls.

**Problem:** one child call might have many potential parent calls.

**Solution:** score those parents by likelihood of being the actual causal parent.

**Scoring approach** for each potential nesting (A, B, C):

→ Analyze **prevalence** of a delay between two call messages in a potentially-causal relationship in the trace dataset.
Nesting Algorithm: Score Causal Nestings

Scoreboard

- **Index**: time difference between parent call A ↔ B and subsequent child call B ↔ C
- **Score value**: $\frac{1}{\text{Number of potential parents}} \times \text{occurrence of delay}$
- **Example**: four potential parent child pairings.

<table>
<thead>
<tr>
<th>Delay</th>
<th>Timestamp $\Delta$</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>Medium delay</td>
<td>$t_3 - t_1$</td>
<td>0.5 + 0.5 = 1</td>
</tr>
<tr>
<td></td>
<td>$t_4 - t_2$</td>
<td></td>
</tr>
<tr>
<td>Long delay</td>
<td>$t_4 - t_1$</td>
<td>0.5</td>
</tr>
<tr>
<td>Short delay</td>
<td>$t_3 - t_2$</td>
<td>0.5</td>
</tr>
</tbody>
</table>
Nesting Algorithm: Score Causal Nestings

```plaintext
1  procedure ScoreNestings
2   for each child (B, C, t₂, t₃) in T_callpairs
3      for each parent (A, B, t₁, t₄) in child.parents
4         scoreboard[A, B, C, t₂ - t₁] += (1/|child.parents|)
```
Nesting Algorithm: Score Causal Nestings

Intermediate result: each nesting is now scored by likelihood of being causally related in the scoreboard.

Next step: find and assign the actual parent/child relationships.

```
1 procedure FindNestedPairs
2   for each child (B, C, t₂, t₃) in T_\text{callpairs}
3       maxscore := 0
4       for each p (A, B, t₁, t₄) in child.parents
5           score[p] := scoreboard[A, B, C, t₂ - t₁] * penalty
6           if (score[p] > maxscore) then
7               maxscore := score[p]
8               parent := p
9           parent.children := parent.children U \{ child \}
```
Nesting Algorithm: Find Call Path

**Intermediate result:** all parent/child relationships are assigned.

**Next step:** build a path from the discovered causal relationships.

```plaintext
1 procedure FindCallPaths
2 initialize has table Tpaths
3 for each callpair (A, B, t₁, t₂)
4     if callpair.parents = Ø then
5         root := new path starting at A
6         root.edges := { CreatePathNode(callpair, t₁) }
7     if root is in Tpaths then update its latencies
8     else add root to Tpaths
```
Nesting Algorithm: Find Call Path

```plaintext
procedure CreatePathNode(callpair (A, B, t₁, t₄), tₚ)
    node := new node with name B
    node.latency := t₄ - t₁
    node.call_delay := t₁ - tₚ
    for each child in callpair children
        node.edges := node.edges U { CreatePathNode(child, t₁) }
    return node
```
Proposed Solution: Convolution Algorithm

1. Select root node
2. For each destination \( j \) from node \( i \) create a vertex \( x_j \) and an edge between \( x_i \) and \( x_j \).
3. At node \( j \) find the sets of messages with source \( j \) that seem to be caused by \( i \).
   - Each set has the same destination node \( k \) and delay \( d \) between incoming and outgoing messages from \( j \).
   - Find causation by using convolution given the indicator function.
4. Add edge between \( x_j \) and \( x_k \) with label \( d \).
5. Continue recursively.
   - Indicator function for messages \( V \) from one node to another.
   - \( s(t) = 1 \) if \( V \) has messages in interval \([t - \epsilon, t + \epsilon]\), \( 0 \) otherwise.

\[
(f * g)(t) \overset{\text{def}}{=} \int_{-\infty}^{\infty} f(\tau) g(t - \tau) \, d\tau
\]
Proposed Solution: Convolution Algorithm

• Spikes
  • $N$ standard deviations above the mean.
  • Join close spikes together by requiring at least one point that is less than $S$ standard deviations above the mean.
  • $S < N$

• Discretization
  • $O(m + S)$ space complexity
  • $O(e^m + e^S\log S)$
    • Second factor dominates
## Algorithm Comparison

<table>
<thead>
<tr>
<th>Nesting Algorithm</th>
<th>Convolution Algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Nesting requires more information.</td>
<td>• Convolution might give less information about the actual paths.</td>
</tr>
<tr>
<td>• Some information can be however be inferred.</td>
<td>• Convolution might not discover rare events.</td>
</tr>
<tr>
<td></td>
<td>• Convolution has a much larger time complexity.</td>
</tr>
</tbody>
</table>
Visualization

What can be visualized?

- Node latency
- Including children
- Total latency
- Message count
Visualization: Nesting Algorithm

Source: Aguilera et al. 2003
Visualization: Convolution Algorithm

Source: Aguilera et al. 2003
Obtaining Traces

- Traces are key to both algorithms.
- Black box approach: feasible?
- Trace collection has potentially large overhead but scales well.
- Two approaches to trace collection:

<table>
<thead>
<tr>
<th>Passive</th>
<th>Active</th>
</tr>
</thead>
<tbody>
<tr>
<td>Port mirroring</td>
<td>No longer truly black box</td>
</tr>
<tr>
<td>Packet sniffing</td>
<td>Some applications already perform logging</td>
</tr>
<tr>
<td>Problems</td>
<td>Java EE</td>
</tr>
<tr>
<td>Message boundaries</td>
<td>Bean-components</td>
</tr>
<tr>
<td>Large amount of data</td>
<td>Large overhead</td>
</tr>
</tbody>
</table>

- Other traces: also usable but no proof given.
- Traces are merged and postprocessed into uniform format.
  Challenges: clock skew, duplicate entries, node-naming inconsistencies.
Experiments: Traces

- No real-life logs
- Traces from active logging
- *maketrace*
  - *tracelet* templates
- Pet store example *Java EE program*
  - Emulating multiple clients
- *Received-Header*
  - Non-RPC based
  - SMTP
Testing: Traces

- maketrace
  - Add 200ms delay to single node
- Java EE
  - Add 50ms delay to single node
- Received-Header
  - Test different time resolution
  - Only convolution
Testing: Other

- Accuracy
  - Ratio of false positive and false negative to the truth
- Pathological cases
  - Habitual behavior of the messages sent
- Parallelism
- Delay variation
- Message loss
- Time skew
- Execution cost
Setup: maketrace
Result: maketrace

Source: Aguilera et al. 2003
Result: maketrace

Source: Aguilera et al. 2003
Result: Java EE
Result: Received Header

- Time quantum of 30s
  - All spikes at 0
- Time quantum of 5s
  - Most at 0
- Nodes named with arbitrary number

Source: Aguilera et al. 2003
Testing: Accuracy of Nesting Algorithm

• **Setup**
  
  • Ground truth generated with nesting algorithm.
    
    → Tag each trace message.
  
  • Run without tags and compare with ground truth.

• **Result**
  
  • Large variety of false positives used with low frequency.
  
  • By pruning low frequency paths it’s possible to increase performance.
  
  • Some false negatives
    
    → Paths which were executed but not found by the algorithms.
Testing: Accuracy of Nesting Algorithm

Source: Aguilera et al. 2003
Testing: Pathological Cases

These cases will be used for testing the accuracy of the nesting algorithm:

- **Children parallel**
  - B calls C twice in parallel

- **Children 0/2**
  - B calls C twice in series in one pattern
  - B has no calls to C in another pattern

- **Children d/cc**
  - B calls C twice in series in one pattern
  - B calls D in another pattern

- **Penalty Breaker**
  - Two paths with multiple calls to the same child, one path with no calls
  - The two longer paths have identical delay
Testing: Pathological Cases

(a) Children-parallel

(b) Children-0/2

(c) Children-d/cc

(d) Penalty-breaker

Source: Aguilera et al. 2003
Result: Parallelism

Source: Aguilera et al. 2003
Result: Standard Deviation

Source: Aguilera et al. 2003
Testing: Message Loss

- Mimicking real behavior with an overflowing queue.
- Result:

![Graph showing false negative rate vs. drop percentage for different strategies: Children-parallel, Children-d/cc, Penalty-breaker, Multi-tier, and Children-0/2. Source: Aguilera et al. 2003.](image)
Result: Clock Skew

Source: Aguilera et al. 2003
Result: Convolution Algorithm

• Varying time quantum between 5s and 720s.

• Compare ground truth with Received-headers.

• 21%-29% false positives.
  • 0% if paths with less than 100 messages are pruned.

• False negatives for frequent paths are 0.
### Result: Execution Cost

<table>
<thead>
<tr>
<th>Trace</th>
<th>Length (messages)</th>
<th>Duration (secs.)</th>
<th>Mean per-node parallelism</th>
<th>MBytes</th>
<th>CPU secs.</th>
<th>(\mu) (secs.)</th>
<th>MBytes</th>
<th>CPU secs.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-tier (short)</td>
<td>20,164</td>
<td>50</td>
<td>1.793</td>
<td>1.5</td>
<td>0.23</td>
<td>0.01</td>
<td>0.2</td>
<td>6684</td>
</tr>
<tr>
<td>Multi-tier “normal”</td>
<td>202,520</td>
<td>500</td>
<td>1.641</td>
<td>13.8</td>
<td>2.27</td>
<td>0.01</td>
<td>0.2</td>
<td>6709</td>
</tr>
<tr>
<td>Multi-tier “added-delay”</td>
<td>196,438</td>
<td>500</td>
<td>1.744</td>
<td>13.4</td>
<td>2.31</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-tier (long)</td>
<td>2,026,658</td>
<td>5000</td>
<td>1.612</td>
<td>136.8</td>
<td>23.97</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-tier, parallelism-low</td>
<td>769,638</td>
<td>5,000</td>
<td>1.146</td>
<td>54.0</td>
<td>7.54</td>
<td>0.01</td>
<td>0.2</td>
<td>6684</td>
</tr>
<tr>
<td>Multi-tier, parallelism-medium</td>
<td>770,344</td>
<td>500</td>
<td>5.116</td>
<td>54.2</td>
<td>11.15</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Multi-tier, parallelism-high</td>
<td>775,254</td>
<td>50</td>
<td>45.057</td>
<td>132.1</td>
<td>233.61</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PetStore “normal”</td>
<td>252,024</td>
<td>1,999</td>
<td>1.322</td>
<td>19.8</td>
<td>3.34</td>
<td>0.02</td>
<td>26</td>
<td>12780</td>
</tr>
<tr>
<td>PetStore “const-delay”</td>
<td>234,036</td>
<td>2,000</td>
<td>1.313</td>
<td>18.4</td>
<td>2.92</td>
<td>0.02</td>
<td>25</td>
<td>6301</td>
</tr>
<tr>
<td>PetStore “normal” (full)</td>
<td>1,345,538</td>
<td>10,799</td>
<td>1.331</td>
<td>97.1</td>
<td>17.12</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PetStore “const-delay” (full)</td>
<td>1,288,223</td>
<td>10,799</td>
<td>1.318</td>
<td>93.2</td>
<td>16.41</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Email headers</td>
<td>81,044</td>
<td>5.1 \times 10^6</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
<td>131</td>
<td>2106</td>
</tr>
<tr>
<td>Email headers</td>
<td>81,044</td>
<td>5.1 \times 10^6</td>
<td></td>
<td></td>
<td></td>
<td>30</td>
<td>36</td>
<td>338</td>
</tr>
</tbody>
</table>

Source: Aguilera et al. 2003
Conclusion

- Two very different methods.
  - Acceptable performance possible even with imperfect traces.
  - Convolution algorithm requires a large amount of time.
  - Hard to obtain traces in a true black box manner.
Questions?

THANK YOU FOR YOUR ATTENTION