Performance Debugging
for Distributed Systems of Black Boxes

Distributed Information Processing, Fall 2015

Summarized by Kyung-Min Kim and Kyoung-Woon On

Contents

- Introduction & Related work
- Problem Settings
- Approach
- Algorithms
- Experiments & Results
- Conclusions

Introduction & Related work

Introduction

- Complex distributed systems are built from black box components
- These systems may have performance problems
- Distributed systems with black box component are hard to debug
- We need to design tools that isolate performance bottlenecks in black-box distributed systems

Related work

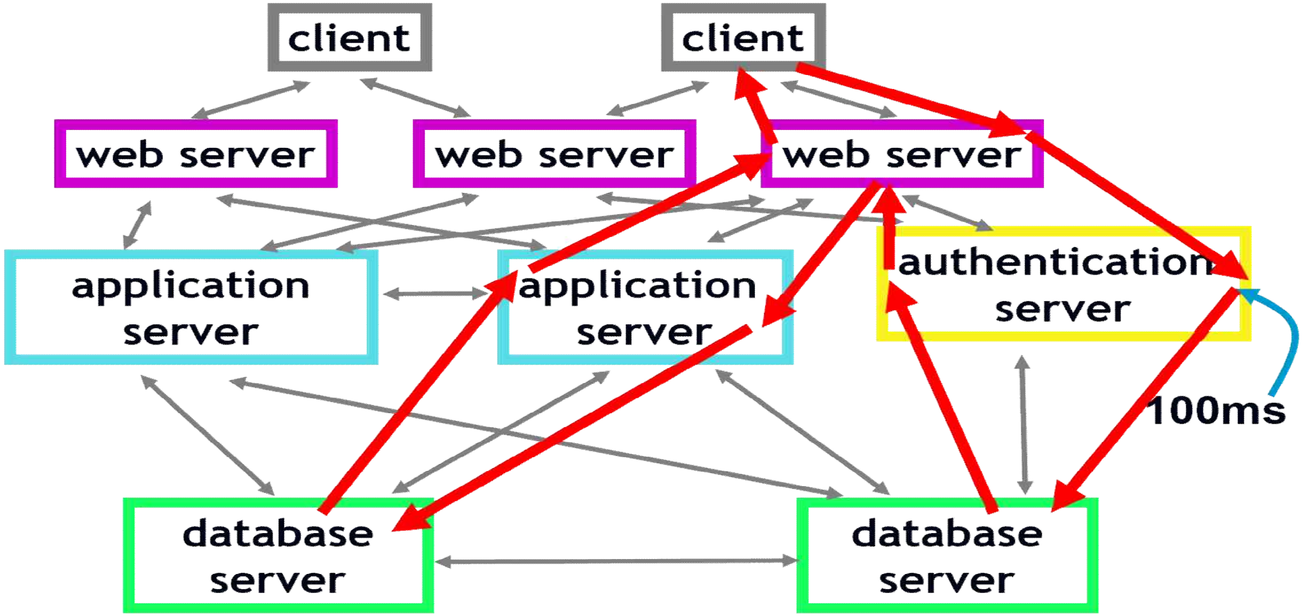
- Systems that trace end-to-end causality via modified middleware
  - Magpie (Microsoft Research)
  - Pinpoint (Stanford/Berkeley)
  - Products such as AppAssure, PerformaSure, OptiBench

- Systems that make inferences from traces
  - Intrusion detection (Zhang & Paxson, LBL)
  - They uses traces + statistics to find compromised systems

Performance Debugging for Distributed Systems of Black Boxes

Problem Settings

Problem Settings



- Situation : an external request to the system causes activities in the graph along a causal path
- Assumption : all latencies can be ascribed to the node traversals

Goals

- Isolating performance bottlenecks
  - Find high-impact causal path patterns
    - Causal path
    - High-impact

  - Identify high-latency nodes on high-impact patterns
    - Add significant latency to these patterns

- Without modifications or semantic knowledge

Performance Debugging for Distributed Systems of Black Boxes

Approach

Approach

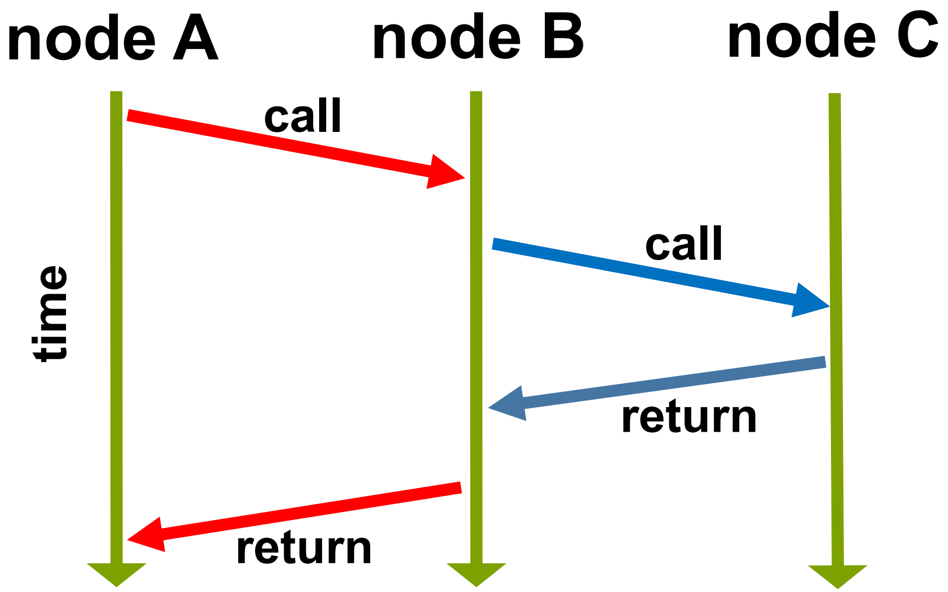
- Obtain traces of messages between components
- Analyze traces using algorithms
  - Nesting: faster, more accurate, limited to RPC-style systems
  - Convolution: works for all message-based systems
- Visualize results and highlight high-impact paths

Performance Debugging for Distributed Systems of Black Boxes

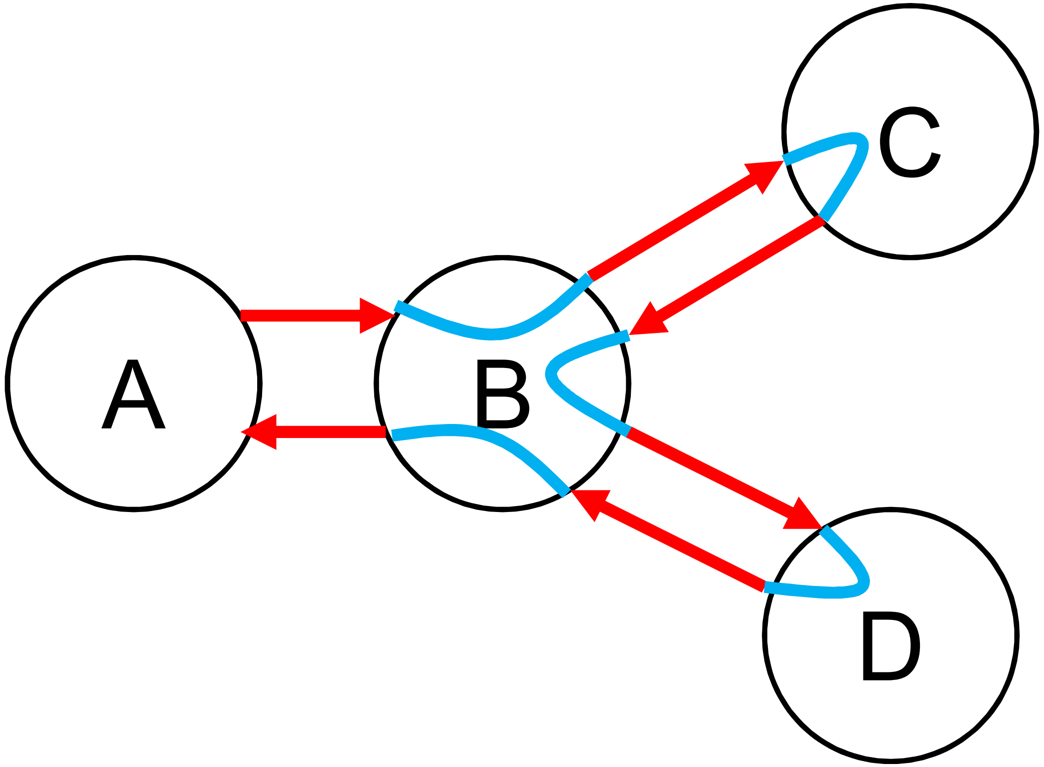
Algorithms

The nesting algorithm

- RPC-style communication
- Infers causality from “nesting” relationships
  - Suppose A calls B and B calls C before returning to A
  - Then the B↔C call is “nested” in the A↔B call
- Uses statistical correlation

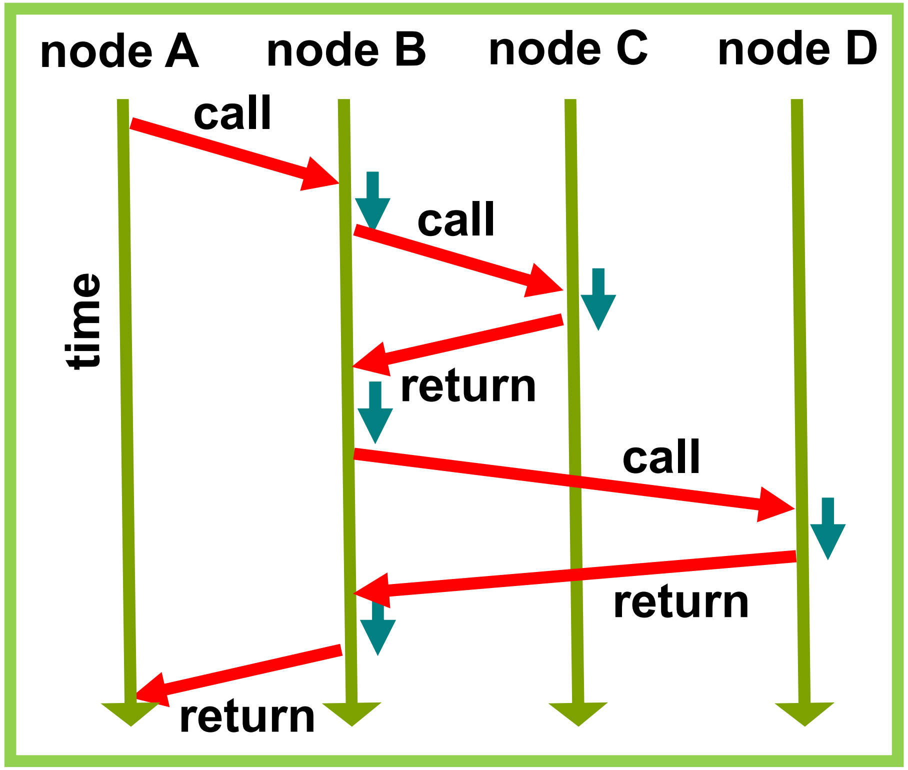


Nesting: an example causal path in detail



Steps of the nesting algorithm

1. Find call pairs in the trace
   - (A⇒B, B⇒A), (B⇒D, D⇒B), (B⇒C, C⇒B)
2. Find and score all nesting relationships
   - B→C nested in A→B
   - B→D also nested in A→B
3. Pick best parents
4. Derive call paths
   - A→B→[C ; D]

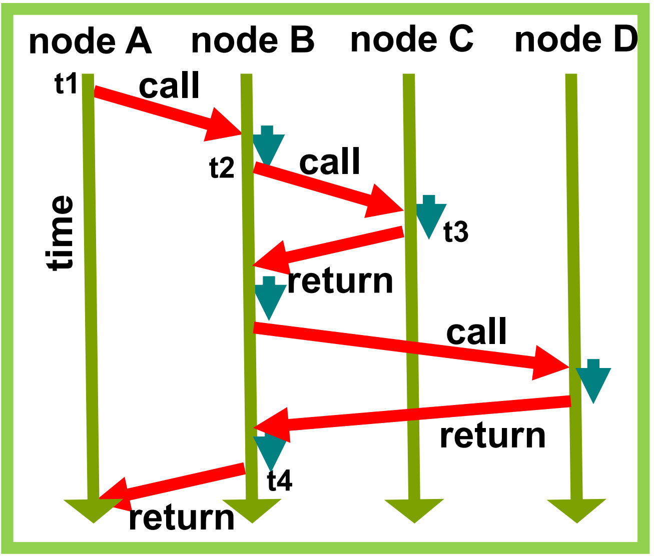


Pseudo-code for the nesting algorithm (1/3)

- Detects calls pairs and find all possible nestings of one call pair in another

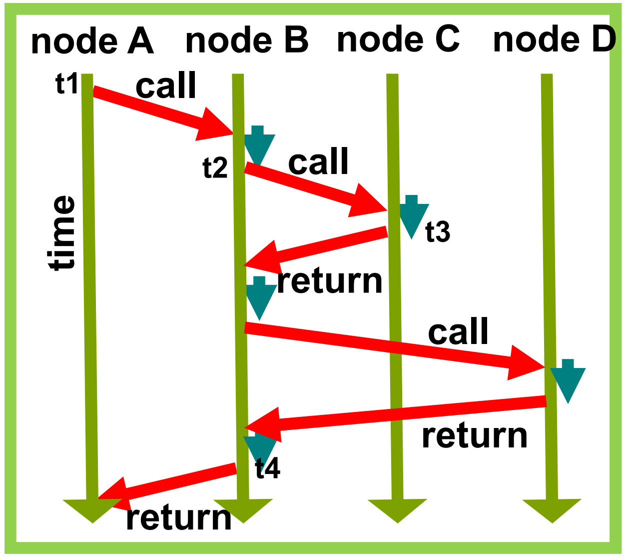
```
procedure FindCallPairs
for each trace entry (t1, CALL/RET, sender A, receiver B, callid id)
    case CALL:
        store (t1,CALL,A,B,id) in Topencalls
    case RETURN:
        find matching entry (t2, CALL, B, A, id) in Topencalls
        if match is found then
           remove entry from Topencalls
           update entry with return message timestamp t2
          add entry to Tcallpairs
          entry.parents := {all callpairs (t3, CALL, X, A, id2) in Topencalls with t3 < t2}
```

FindCallPairs (1/2)



- Trace entry
  - {(A, B, id1), (B, C, id2), (C, B, id2…}
- Topencalls
  - A set of not yet paired traces
  - Hash table structure
- For each trace in trace entry, find matching entry in Topencalls using sender, receiver, callid information and save pair into Tcallpairs
- All possible parents information is obtained by finding precedent call pairs in Topencalls

FindCallPairs (2/2)



- $T_{opencalls}$

Time	CALL/RET	Sender	Receiver	ID
t1	CALL	A	B	ID1
t2	CALL	B	C	ID2

- When we process (C, B, ID2)..
  - (B, C, ID2) in $T_{opencalls}$ is matched
  - Find call pairs (-, B) in Topencalls with an earlier call timestamp
    - There is (A, B, ID1) with earlier timestamp
    - So (A, B, ID1) becomes the parents of the call pair (B, C, ID2)

Pseudo-code for the nesting algorithm (2/3)

- Pick the most likely candidate for the causing call for each call pair

```
procedure ScoreNestings
for each child (B, C, t2, t3) in Tcallpairs
  for each parent (A, B, t1, t4) in child.parents
    scoreboard[A, B, C, t2-t1] += (1/|child.parents|)

procedure FindNestedPairs
for each child (B; C; t2; t3) in call pairs
   maxscore := 0
   for each p (A, B, t1, t4) in child.parents
      score[p] := scoreboard[A, B, C, t2-t1]*penalty
      if (score[p] > maxscore) then
         maxscore := score[p]
         parent := p
   parent.children := parent.children U {child}
```

ScoreNestings (1/2)

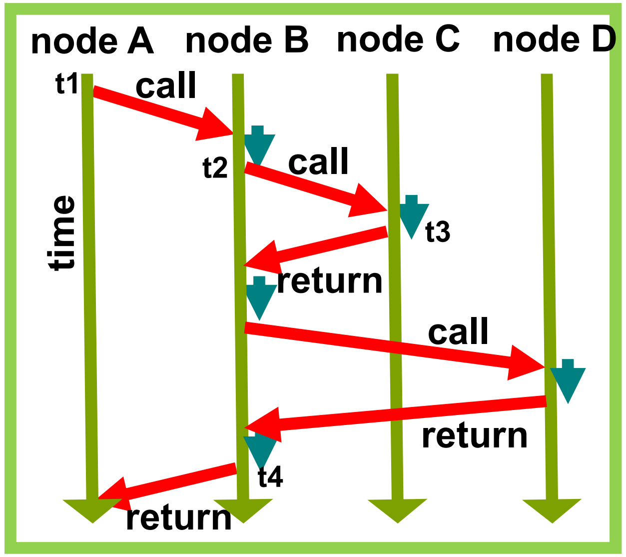
- $T_{callpairs}$

Sender	Receiver	Time1	Time2
A	B	t1	t4
B	C	t2	t3
…			

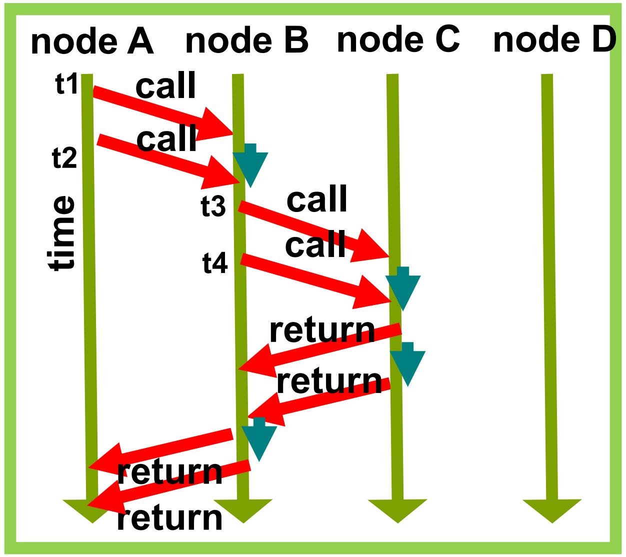
- For each child pair in Tcallpairs, find parent and store score into scoreboard
  - (A, B, ID1) is a parent of (B, C, ID2)
  - Scoreboard

Node1	Node2	Node3	Delta
A	B	C	t2-t1

  - If there are many parents for the child, score will be (1/|parents|) and definitely most probable causal parent pair will get highest score



ScoreNestings (2/2)



- Ambiguous case
- Each B to C call pair can have two different parent (A to B)
- In Scoreboard, there are 4 possible entries
  - Long-length delay : (A, B, C, t4-t1)
  - Short-length delay : (A, B, C, t3-t2)
  - Medium-length delay : (A, B, C, t3-1) & (A, B, C, t4-t2)
  - Score of Medium-length delay > score of long&short-length delay

Pseudo-code for the nesting algorithm (3/3)

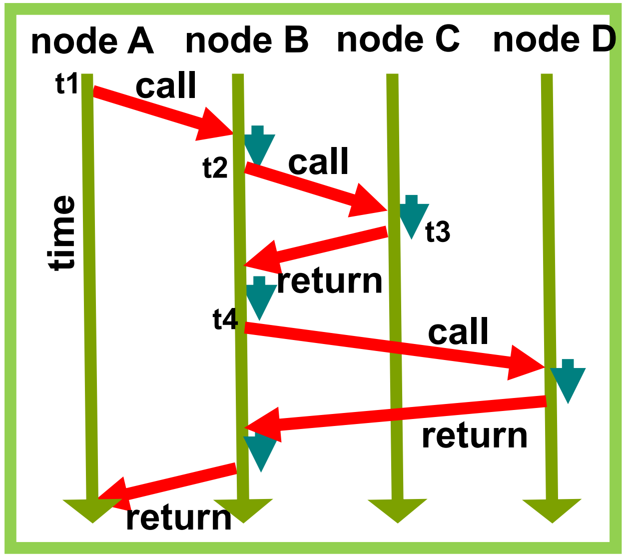
- Derive call paths from the causal relationships

```
procedure FindCallPaths
initialize hash table Tpaths
for each callpair (A, B, t1, t2)
  if callpair.parents = null then
      root := { CreatePathNode(callpair, t1) }
      if root is in Tpaths then update its latencies
      else add root to Tpaths
function 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
```

FindCallPaths

- Find the most parent node
  - **for each** callpair (A, B, t1, t2)

    if callpair.parents = null **then**



- **Node A** becomes root node

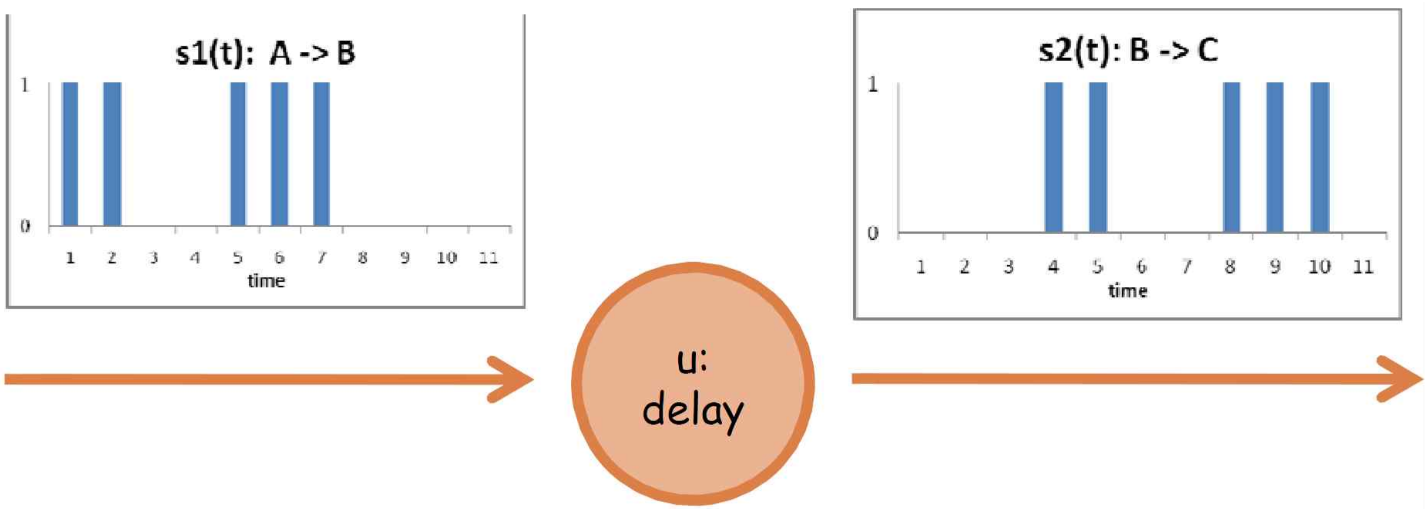
- Make a path by adding child nodes to the edges of root node
  - A->B->C;D

The “convolution algorithm”

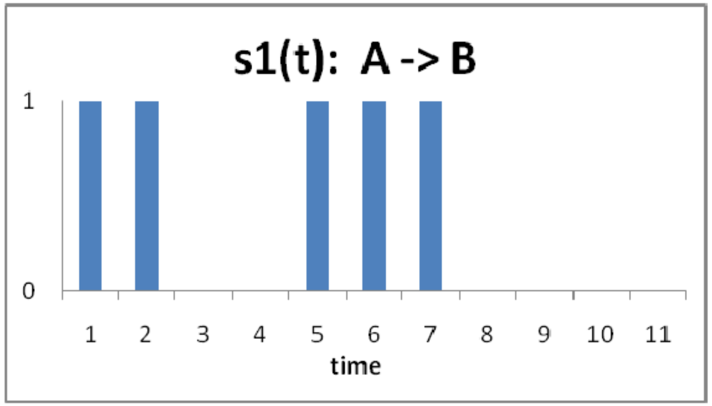
- Finds causal relationships
  - Considering the aggregation of multiple messages.
  - Separates a whole system trace into a set of per-edge traces.
- Convert traces into time signals (per-edge traces)
  - Use signal processing techniques to find the cross correlations between signals
  - Can be used on traces of free-form message-based communications

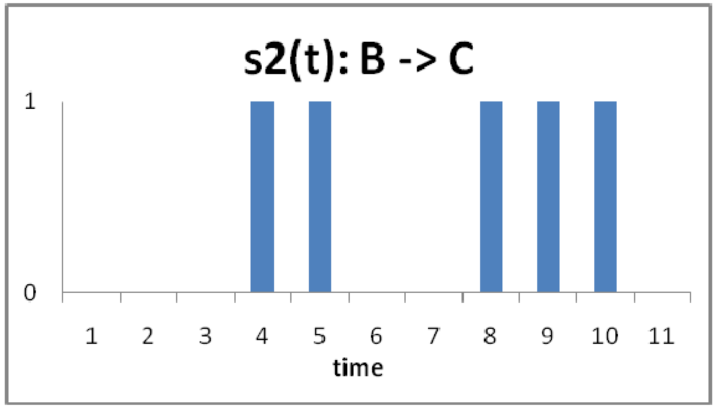
The “convolution algorithm”

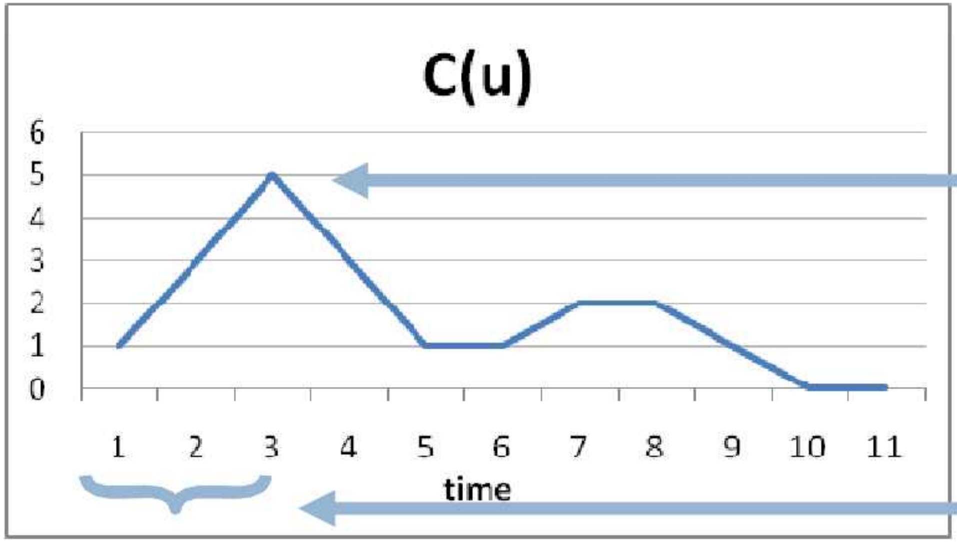
- Look for time-shifted similarities
  - Compute cross correlation by convolution
    - $C(t) = S_1 \otimes S_2(t) = \int_{-\infty}^{\infty} S_1(u) S_2(t-u) du$
  - Find peaks in C(t)
  - Time shift of peak indicate delay
  - Considering the aggregation of multiple messages.
  - Separates a whole system trace into a set of per-edge traces.



The “convolution algorithm”







Peaks suggest causality between A→B and B→C

delay

The “convolution algorithm”

- Detect the spikes (peaks)
  - Compute mean and standard deviations of C
  - Spike if in is a local maximum N (e.g., 4) standard deviations above the mean
  - Require at least one point that is less than S (e.g., 3) standard deviations above the mean between spikes, where S < N
  - Chose largest to represent the spike

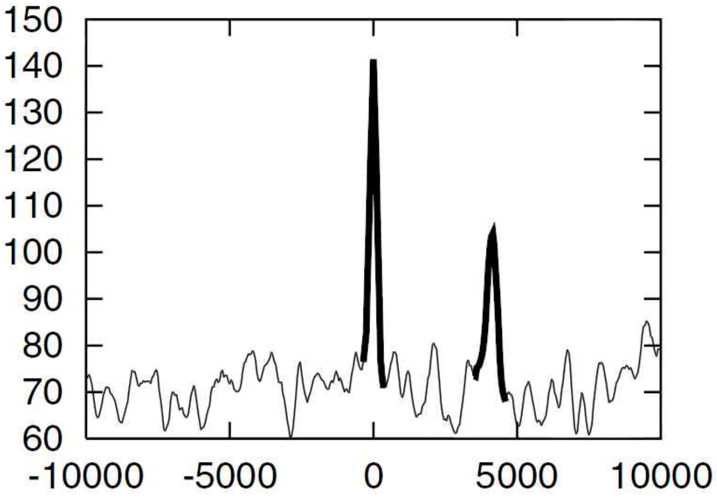


Figure 7: Example of convolution output, showing two spikes with bold lines. The x-axis represents the time shift; the y-axis roughly estimates the number of messages matching a given shift.

The “convolution algorithm”

- Time complexity: O(em+eSlogS)
  - m = # message
  - e = # edge in output graph
  - s = # time steps in trace
- Need to choose time step size
  - Must be shorter than delays of interest
  - Too coarse: poor accuracy
  - Too fine: long running time
- Robust to noise in trace
- Run-time is dependent on the trace duration and time quantum, not the trace length

Comparison of the two algorithms

	Nesting Algorithm	Convolution Algorithm
Communication style	RPC only	RPC or free-form messages
Rare events	Yes, but hard	No
Level of Trace detail	<timestamp, sender, receiver> + call/return tag	<timestamp, sender, receiver>
Time and space complexity	Linear space Linear time	Linear space Polynomial time
Visualization	RPC call and return combine	Less compact

Performance Debugging for Distributed Systems of Black Boxes

Experiments and Results

Maketrace

- Synthetic trace generator
- Needed for testing
  - Validate output for known input
  - Check corner cases
- Uses set of causal path templates (tracelet)
  - All call and return messages, with latencies
  - Gaussian delay between messages

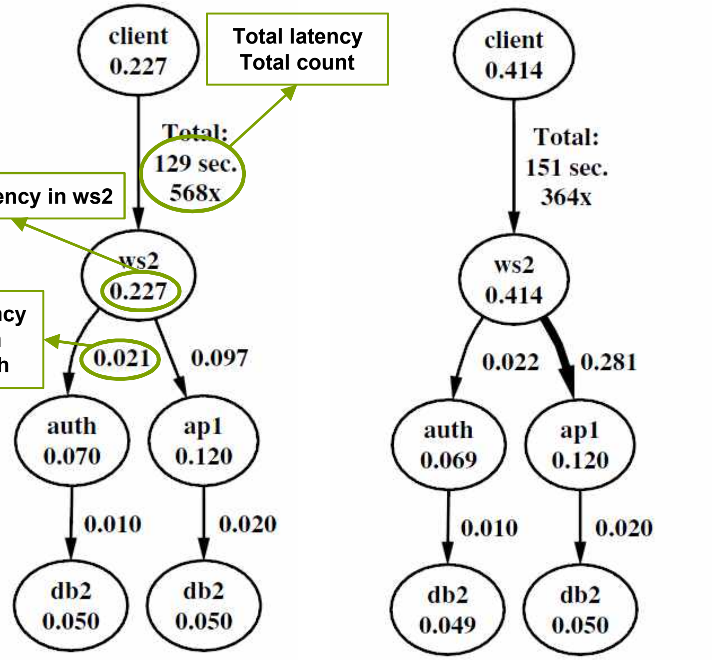
- Recipe to combine paths
  - Parallelism, start/stop times for each path
  - Duration of trace

Desired results for one trace

- Causal paths
  - How often
  - How much time spent
- Nodes
  - Host/component name
  - Time spent in node and all of the nodes it calls
- Edges
  - Time parent waits before calling child

Measuring Added Delay

- Added 200msec delay in WS2
- The nesting alg. detects the added delay, and so does the convolution algorithm



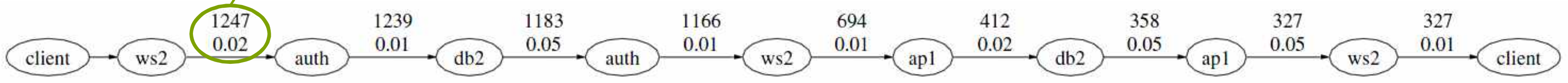
Total latency
Total count

Mean latency in ws2

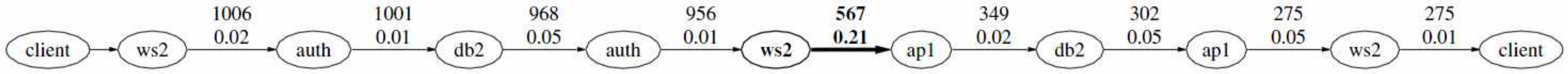
Mean latency between ws2&auth

(a) Normal trace (b) Added-delay trace

Ws2->auth count
Delay between ws2&auth



(a) Normal trace



(b) Added-delay trace

Results: Petstore

- Sample EJB application
- J2EE middleware for Java
  - Instrumentation from Stanford's PinPoint project
- 50msec delay added in mylist.jsp

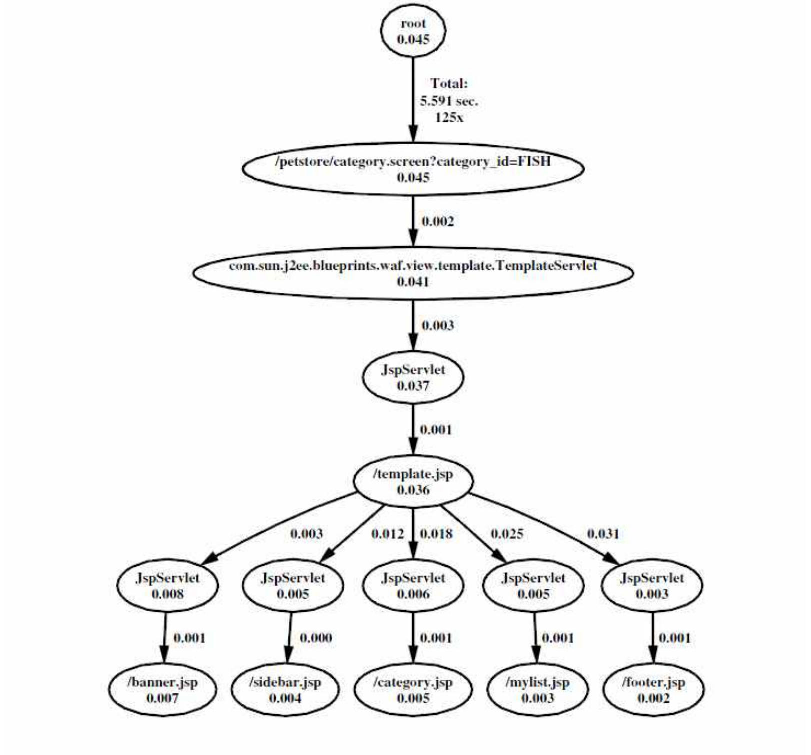


Figure 14: PetStore results, normal configuration (nesting algorithm)

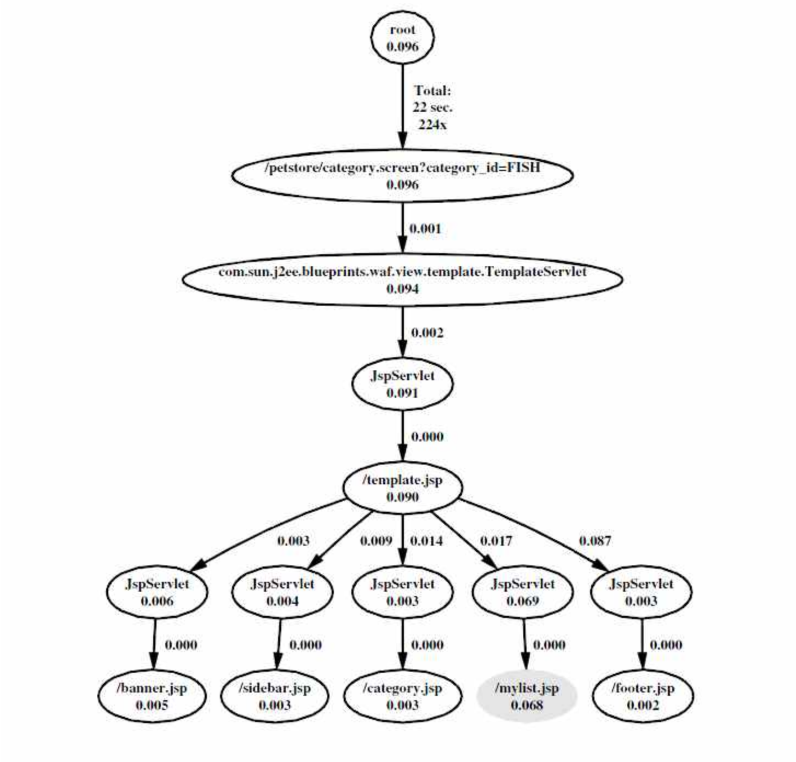


Figure 15: PetStore results, constant-delay config. (nesting algorithm)

Validation of accuracy

- False negative rate for top N pattern
  - Is bounded in most cases my 1/N

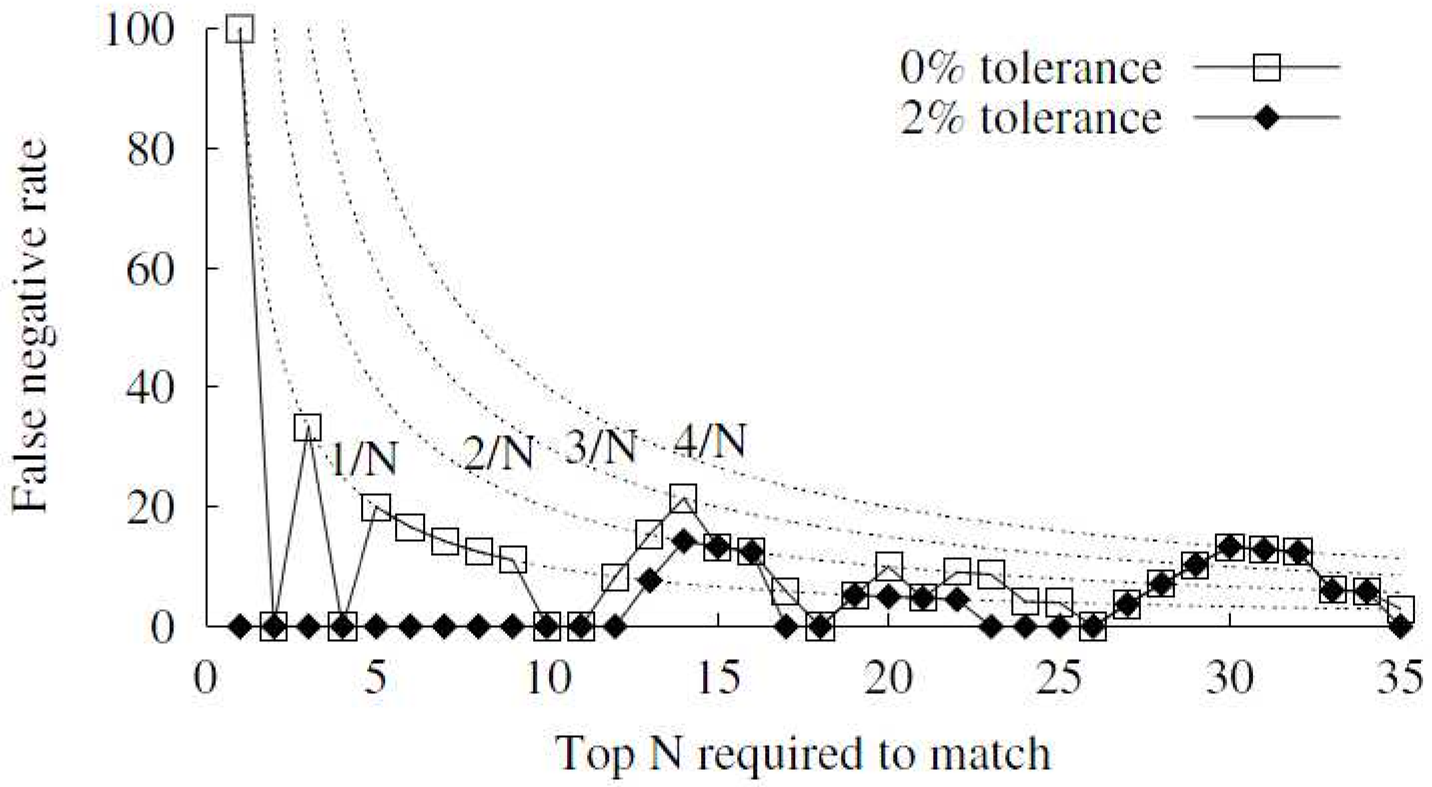
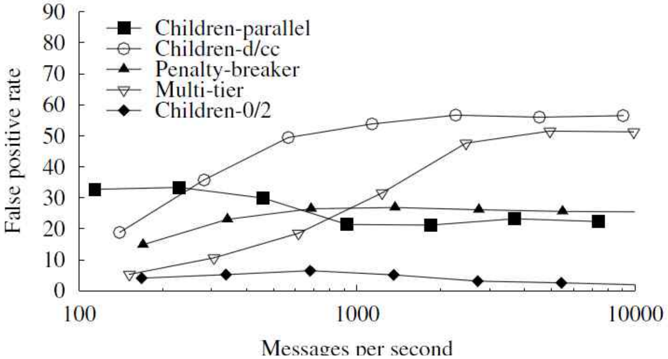


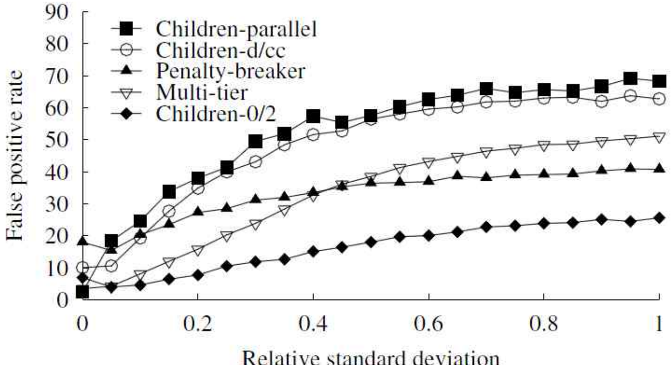
Figure 17: **False negative path pattern rate vs. pattern pruning**

accuracy

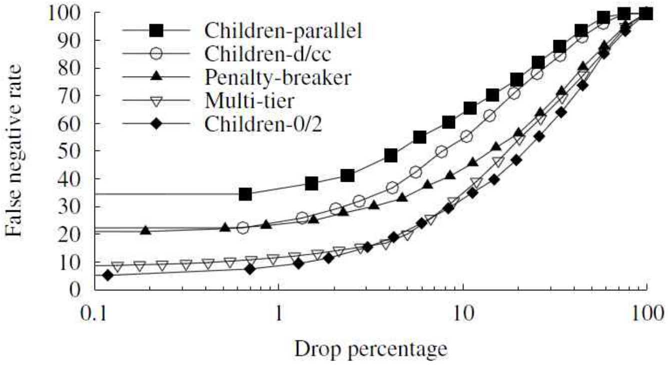
- Trace parallelism



- Delay variation



- Message drop rate



Performance Debugging for Distributed Systems of Black Boxes

Conclusions

Conclusions

- Looking for bottlenecks in black box systems
- Finding causal paths is enough to find bottlenecks
- Algorithms to find paths in traces really work
  - We find correct latency distribution
  - Two very different algorithms get similar results
  - Passively collected traces have sufficient information

Thank you