Pinpoint: Problem Determination in Large, Dynamic Internet Services

Mike Chen, Emre Kiciman, Eugene Fratkin
mikechen@cs.berkeley.edu
{emrek, fratkin}@cs.stanford.edu

ROC Retreat, 2002/01
Motivation

- Systems are large and getting larger
  - 1000’s of replicated HW/SW components used in different combinations
  - composable services further increase complexity

- Systems are dynamic
  - frequent changes
    - software releases, new machines
  - resources are allocated at runtime
    - e.g. load balancing, QoS, personalization

- Difficult to diagnose failures
  - what is really happening in the system?
  - how to tell what’s different about failed requests?
Existing Techniques

- **Dependency models/graphs**
  - Detect failures, check all components that failed requests depend on
  - Problem:
    - Need to check all dependencies (large # of false positives)
    - Hard to generate and keep up-to-date

- **Monitoring & alarm correlation**
  - Detect non-functioning components and often generates alarm storms
    - filter alarms for root-cause analysis
  - Problem:
    - need to instrument every component
    - hard to detect interaction faults and latent faults
The Pinpoint Approach

- **Trace *real* client requests**
  - record every component used in a request
  - record success/failure and performance of requests
  - can be used to build dynamic dependency graphs to visualize what is *really* going on

- **Statistical analysis**
  - search for components that “cause” failures
  - data mining techniques

- **Built into middleware**
  - requires no application code changes
  - application knowledge only for end-to-end failure detection
Examples

- Identify faulty components

- Anomaly detection
Framework

Communications Layer
(Tracing & Internal F/D)

Components
A
B
C

Requests
#1
#2
#3

External F/D

Logs
1, success
2, fail
3, ...

1,A
1,C
2,B..

Statistical Analysis

Predicted Faults
Prototype Implementation

- Built on top of J2EE platform
  - Sun J2EE 1.2 single-node reference implementation
  - added logging of Beans, JSP, & JSP tags
  - detect exceptions thrown out of components
  - required no application code changes

- Layer 7 network sniffer in Java
  - TCP timeouts, HTTP errors, malformed HTML

- PolyAnalyst statistical analysis
  - bucket analysis & dependency discovery
  - offline analysis
Experimental Setup

- Demo app: J2EE Pet Store
  - e-commerce site w/\sim 30 components
- Load generator
  - replay trace of browsing
  - approx. TPCW WIPSo load (~50% ordering)
- Fault injection
  - 6 components, 2 from each tier
  - single-components faults and interaction faults
  - includes exceptions, infinite loops, null calls
- 55 tests, 5 min runs
  - performance overhead of tracing/logging: 5%
Observations about the PetStore App

- large # of components used in a dynamic page request: median 14, min 6, max 23
- large sets of tightly coupled components that are always used together
### Metrics

- **Precision**: identified/predicted, \( (C/P) \)
- **Recall**: identified/actual, \( (C/A) \)
- **Accuracy**: whether all actual faults are correctly identified (recall == 100%)
  - boolean measure
4 Analysis Techniques

- **Pinpoint**: clusters of components that statistically correlate with failures
- **Detection**: components where Java exceptions were detected
  - union across all failed requests
  - similar to what an event monitoring system outputs
- **Intersection**: intersection of components used in failed requests
- **Union**: union of all components used in failed requests
Results: Accuracy/Precision vs Technique

- Pinpoint has high accuracy with relatively high precision
Prototype Limitations

- **Assumptions**
  - client requests provide good coverage over components and combinations
  - requests are autonomous (don’t corrupt state and cause later requests to fail)
    - however, dependency graphs are useful to identify shared state

- **Currently can’t detect the following:**
  - faults that only degrade performance
  - faults due to pathological inputs
    - help programmers debug by recording and replaying failed requests
Future Work

- Visualization of dynamic dependency
  - at various granularity: components, machine, tier
- Capture additional differentiating factors
  - URLs, cookies, DB tables
  - helps to identify independent faults
- Study the effects of transient failures
- Performance analysis
- Online statistical analysis
- Looking for real systems with real applications
  - Oceanstore? WebLogic/WebSphere?
Conclusions

- Dynamic tracing and statistical analysis give improvements in accuracy & precision
  - Handles dynamic configurations well
  - Without requiring application code changes
  - Reduces human work in large systems
  - But, need good coverage of combinations and autonomous requests
Thank you

- Acknowledgements: Aaron Brown, Eric Brewer, Armando Fox, George Candea, Kim Keeton, and Dave Patterson
Backup slides
Results: Accuracy under Interaction Faults

![Graph showing average accuracy under interaction faults. The x-axis represents the number of interacting components, ranging from 1 to 4. The y-axis represents the average accuracy, ranging from 0% to 100%. The graph includes four lines for different methods: Pinpoint, Detection, Intersection, and Union. The Pinpoint method shows the highest accuracy at all component counts, followed by the Detection method, then the Intersection method, and finally the Union method which shows the lowest accuracy.]
Problem Determination

- Analogy: trying to locate a car accident on Golden Gate Bridge
  - on a foggy day
  - using a toy model
  - on a clear day