Pinpoint: Problem Determination in Large, Dynamic Internet Services

Mike Chen, Emre Kıcımán, 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

- **Systems are dynamic**
  - resources are allocated at runtime
  - e.g. load balancing, personalization

- **Difficult to diagnose failures**
  - how to tell what’s different about failed requests?
Current Techniques

- **Dependency Models**
  - Detect failures, check all components that failed requests depend on
  - Problem:
    - Need to check all dependencies
    - 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
The Pinpoint Approach

- Trace many *real* client requests
  - Record every component used in a request
  - Detect success/failure of requests
  - Can be used as dynamic dependency graphs

- Statistical Correlation
  - Search for components that “cause” failures

- Built into middleware
  - Requires no application code changes
  - Application knowledge only for end-to-end failure detection
Framework

Components:
- A
- B
- C

Communications Layer (Tracing & Internal F/D)

Requests:
- #1
- #2
- #3

External F/D

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

Detected Faults

Statistical Analysis
Prototype Implementation

- Built on top of J2EE platform
  - Sun J2EE 1.2 single-node reference impl.
  - Added logging of Beans, JSP, & JSP tags
  - Detect exceptions thrown out of components
  - Required no application code changes

- Layer 7 network sniffer
  - TCP timeouts, malformed HTML, app-level string searches

- PolyAnalyst statistical analysis
  - Bucket analysis & dependency discovery
Experimental Setup

- Demo app: J2EE Pet Store
  - e-commerce site w/~30 components

- Load generator
  - replay trace of browsing
  - Approx. TPCW WIPS0 load (~50% ordering)

- Fault injection parameters
  - Trigger faults based on combinations of used components
  - Inject exceptions, infinite loops, null calls

- 55 tests with single-components faults and interaction faults
  - 5-min runs of a single client (J2EE server limitation)
Application Observations

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

- **Precision**: $C/P$
- **Recall**: $C/A$
- **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
Pinpoint has high accuracy with relatively high precision
Pinpoint 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)

- **Currently can’t detect the following:**
  - faults that only degrade performance
  - faults due to pathological inputs
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

Future Work
- Test with real distributed systems with real applications
  - Oceanstore? WebLogic/WebSphere?
- Capture additional differentiating factors
- Visualization of dynamic dependency
- Performance analysis
- Online data analysis