Model Based Validation

Presented by: Rich Martin

Joint work with: Andrew Tjang, Fabio Olivera
Thu D. Nguyen, Ricardo Bianchini,

Rutgers University

Presented at Ask.com, Piscataway, May 2006

Availability of Computer Systems

- Poor availability
  - – Typical PC 1 nine (90%, downtime: ~1 month/yr)
  - – Department server: 2 nines (99% ~3 days/yr)
  - – Large service: 2-3 nines (99%-99.9%, ~10 hours/yr)
  - – Mega service: 4-5 nines (99.99%-99.999%, ~ 30min/yr)
- Can high availability become ubiquitous?
  - Make at least 1 order of magnitude cheaper.
    - – Don’t want pay $5,000/line of code
    - – Don’t want to hire a 1000 PhDs

How to improve?

- Top 2 sources of unavailability
  - Software bugs
  - Human/system interaction
    - operator in broadest sense
- These are unchanged for decades
    - Tandem Survey 1984 - DBMS admin survey, 2006
- Unlikely to improve without new paradigms, metrics and techniques
- Resulting high cost will limit applications

Human-Aware System Design

- The human is part of the system
  - Human mistakes a primary cause of failures in systems

- Make human-system interaction a first-class design concern
  - Understand operator actions and mistakes
  - Techniques to avoid, tolerate, diagnose, and correct mistakes

- How is this related to Human-Computer Interaction (HCI)?
  - (Re) design system with human mistakes in mind
  - HCI efforts focused on ease-of-use and cognitive models
  - Complementary since we are exploring system support for human operation of complex systems
    - E.g. better interfaces are good, but a human-mistake tolerant system is even better

Talk Outline

- Motivation
- Human aware system design
- Our approach: Validation
- A language overview
  - Language and implementation
- Using A
- Evaluation
- Conclusions & future work

Our Approach: Validation

- Previous work: Component Validation
  - Avoid mistakes by testing a component before use
  - Replica + trace based
  - Assumed we had working replicas to compare
  - Assumed we had workloads to exercise
  - Human factors study: 60% mistakes caught, 40% missed

- This talk: Model based validation
  - Build a model of correct behavior
  - Check the model against the running system before, during and after a human interaction
  - Take action when system deviates from the model

Approach

- (1) Building models of a correct system
  - Relevant concerns:
    - Performance, Resource Allocation/Exhaustion, Connectivity, Security, Configuration, Content
  - High level modeling paradigms
  - A new language to specify correctness: the A language
    - General as possible to allow multiple modeling paradigms
    - Language facilitates multiple people contributing to a model
  - Encode steps to performing a human task
  - An A program realizes the model
- (2) Checking the model
  - Job of the A program runtime

Approach, cont

- Checking the model
  - Compile an A program and run it
  - A language runtime:
    - Gathers state from real system and performs checks
    - Presents operators a set of possible tasks
    - Within each task set of actions and checks
    - Outputs when assertions failed
    - Outputs when actions not completed in time
    - Open: Actuation when assertions failed.

A Language Users and Goals

- Target: describe complex computer systems and human interactions
    - Many interacting hardware/software components
- A Programmers:
    - System designers
    - Senior operations staff
- Users:
    - Operations staff
    - System auditors
- Uses:
    - Description of correct behavior
    - Alarms and alerts
    - Future: actuation, diagnosis

Contributions

- Why not use a custom solution?

  E.g. Ad hoc scripts and alarms
- A: Formalizes correctness checking
  - Tractable to translate models into working code
  - Easier to reason about coverage, complexity
- Run-time: Single observation point
  - Easier development, verification, debugging
- Approach helps force proactivity as opposed to reactivity

Talk Outline

- Motivation
- Human aware system design
- Our approach: Validation
- 3 Model paradigms
- A language overview
  - Language and implementation
- Using A
- Evaluation
- Conclusions & future work

Approaching A programs

- Goal: a higher-level structure
  - E.g libraries for various tasks
    - Analogy: Collections, Strings in programming languages
- What are the modeling paradigms?
  - No one model captures everything
  - Our approach: graph representations of different aspects of the system

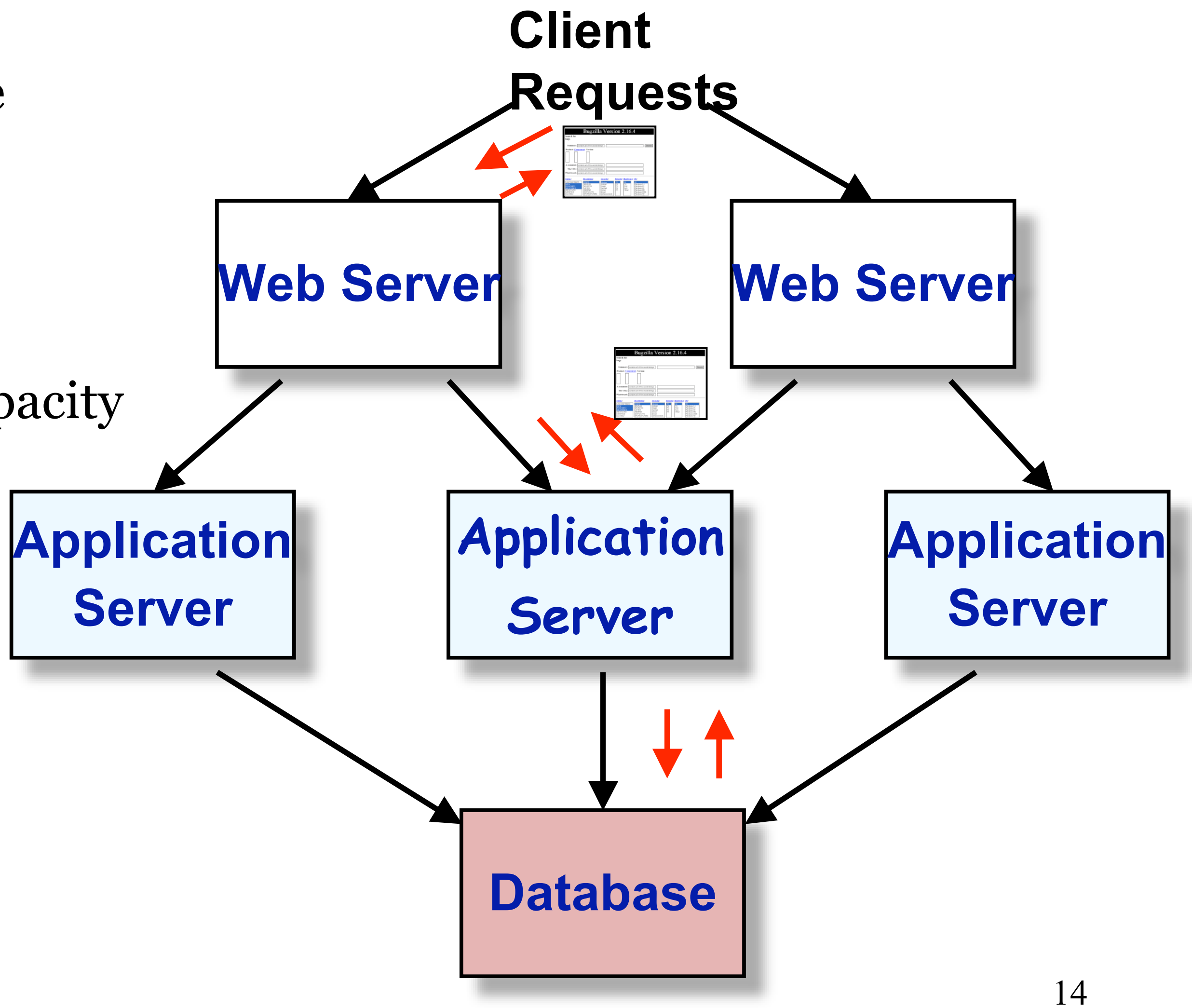
3 Modeling Paradigms

Paradigm: method to express invariants

- Flow
  - Nodes are computations, edges are messages
  - Assertions: throughput, latency, connectivity, capacity
- Sub-Component
  - Nodes are computations, edges are sub-components
  - Assertions: type/number of subs failed implies overall is is failed
- Security
  - Access Control Matrix
  - Nodes are users and resources
  - Edges: allows/access
  - Assertions: sets of allowable edges

Flow model

- Applies to any message passing system
- Reason about flows of messages
- Load introduces flow
- Elements introduce capacity restrains



Flow Assertions

- High-level flow assertion concepts:

  X connected to Y

  Flow <= capacity of a component

  Flow >= 0

  Flow in == flow out

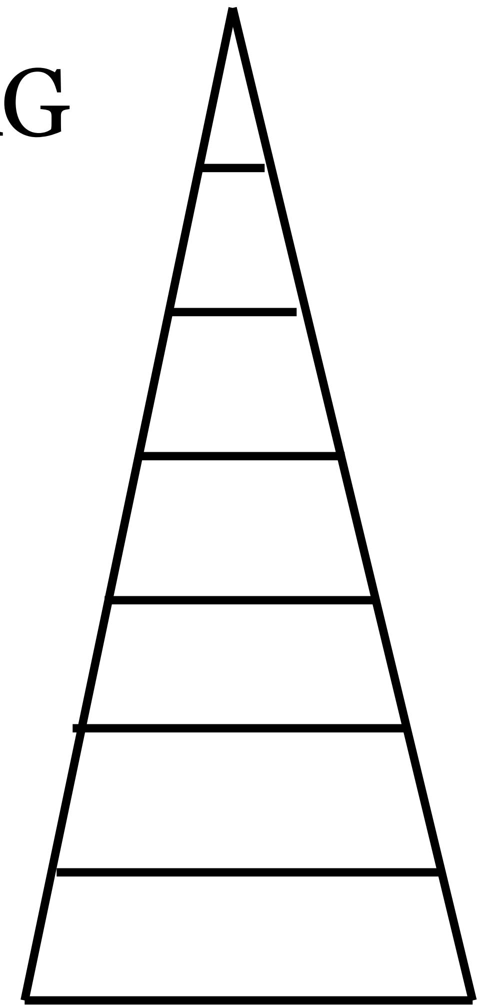
  Flow.regression < max_slope

  Flow.std_dev < max_deviation

- Each of these must be fleshed out with low-level assertions
  - E.g. what does "assert connected" mean?
    - Ping, HTTP response, Special heartbeat, etc.

Sub-Component Model

DAG



Entire Service working?

Sites, ISP, Power, AC OK?

Web, Application Servers, Databases

Nodes, Links, Switches, Routers

Process, Files, Tables

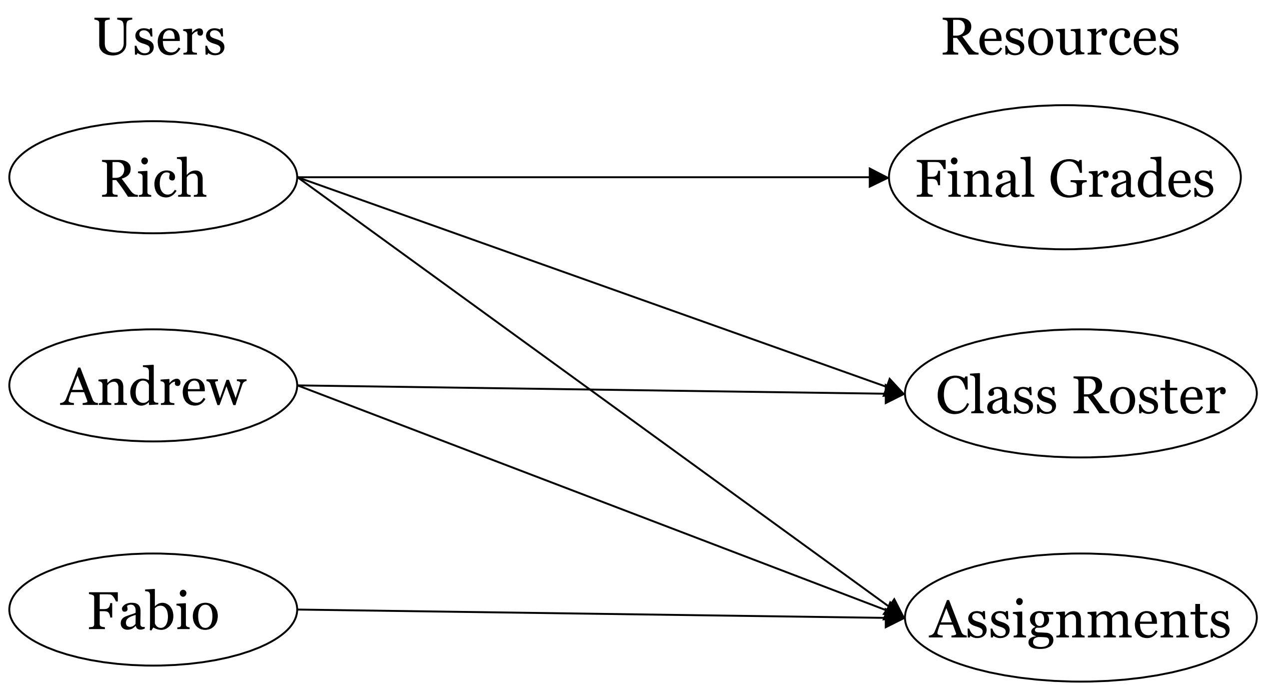
CPU, Disk, memory

Busses, Cache, Channels

Security Model

Access Control Matrix

Bipartite Graph



Talk Outline

- Motivation
- Human aware system design
- Our approach: Validation
- 3 Model paradigms
- A language overview
  - Language and implementation
- Using A
- Evaluation approaches
- Conclusions & future work

A: Program Structure

- General purpose assertions bound to run-time system state
- Libraries for specific objects and properties
  - Connectivity
  - Flow
  - Capacity
  - Latency
  - Users
  - Resources

A: Language Abstractions

- System state: (3 types)
  - Elements (hardware/software components)
  - Static Input (Configuration files)
  - Stream output (Logs)
- Bindings
  - Elements/Configs/Logs bound to real system objects
- Assertions
- Tasks (Sequential human execution)

Example A code

```
element loadbalancer {
   (IP address);
   stat net.requests;
   stat net.responses;
}

element webserver {
   (IP address);
   stat requests;
   stat responses;
   stat throughput;
   stat utilization;
   element CPU;
}

element CPU {
   stat utilization;
   stat idle;
}
```

```
ws1::webserver(“/192.168.1.1/");
ws2::webserver("/192.168.1.2/");
wsboth::webserver("/[192.168.1.1|192.168.1.2]/");
wsall::webserver("/192.168.*/") ;
lb::loadbalancer("192.168.0.1");

assert overload (wsall..CPU.utilization < 0.90)  { }
else {
      log(“A webserver is overloaded”)
}
assert balanced
 (ws1.CPU.utilization == {20} ws2.CPU.utilization) {
      freq=1s;
      ON;
} else {
      log(“Backends are not balanced”);
}
```

Element

Example A code

```
element loadbalancer {
   (IP address);
   stat net.requests;
   stat net.responses;
}

element webserver {
   (IP address);
   stat requests;
   stat responses;
   stat throughput;
   stat utilization;
   element CPU;
}

element CPU {
   stat utilization;
   stat idle;
}
```

```
ws1::webserver("/192.168.1.1/");
ws2::webserver("/192.168.1.2/");
wsboth::webserver("/[192.168.1.1|192.168.1.2]/");
wsall::webserver("/192.168.*/") ;
lb::loadbalancer("192.168.0.1");

assert overload (wsall..CPU.utilization < 0.90)
else {
      log("A webserver is overloaded");
}
assert balanced
(ws1.CPU.utilization == {20} ws2.CPU.utilization) ( {
      freq=1s;
      ON;
} else {
      log("Backends are not balanced");
}
```

Example A code

Element

Bindings

```
element loadbalancer {
   (IP address);
   stat net.requests;
   stat net.responses;
}

element webserver {
   (IP address);
   stat requests;
   stat responses;
   stat throughput;
   stat utilization;
   element CPU;
}

element CPU {
   stat utilization;
   stat idle;
}
```

```
ws1::webserver("/192.168.1.1/");
ws2::webserver("/192.168.1.2/");
wsboth::webserver("/[192.168.1.1|192.168.1.2]/");
wsall::webserver("/192.168.*/") ;
lb::loadbalancer("192.168.0.1");

assert overload (wsall..CPU.utilization < 0.90)
else {
      log("A webserver is overloaded");
}
assert balanced
(ws1.CPU.utilization == {20} ws2.CPU.utilization) {
      freq=1s;
      ON;
} else {
      log("Backends are not balanced");
}
```

Example A code

Element

Bindings

Assertion

```
element loadbalancer {
   (IP address);
   stat net.requests;
   stat net.responses;
}

element webserver {
   (IP address);
   stat requests;
   stat responses;
   stat throughput;
   stat utilization;
   element CPU;
}

element CPU {
   stat utilization;
   stat idle;
}
```

```
ws1::webserver("/192.168.1.1/");
ws2::webserver("/192.168.1.2/");
wsboth::webserver("/[192.168.1.1|192.168.1.2]/");
wsall::webserver("/192.168.*/") ;
lb::loadbalancer("192.168.0.1");

assert overload (wsall..CPU.utilization < 0.90)
else {
      log("A webserver is overloaded");
}
assert balanced
 (ws1.CPU.utilization == {20} ws2.CPU.utilization) {
      freq=1s;
      ON;
} else {
      log("Backends are not balanced");
}
```

Aggregates

```
assert overload (wsall..CPU.utilization < 0.90) {
   freq=1s;
   ON;
}{
   log(“A webserver is overload”)
}
```

Aggregate operator

- Aggregate supports replication
- Typed at binding time
- Operator applied to each element
- Assertion fails if any fails (implicit and)

Assertion Hierarchy

```
assert loadOK ( load_1_OK && load_2_OK)
else {
   //Action Block
}
assert load_1_OK (ws1.cpu.utilization <= 0.80)
else {
   log("workstation 1 overloaded")
}
```

- Can specify assertion name in an expression
- Sub-assertions evaluated in response to parent assertion
- Assertion will be evaluated at rate of fastest parent

Configuration and Log files

```
config WS_Apache{
   :httpdconf: “Drivername”
       single docroot = /root/DocumentRoot, "";
    :workprop: “Drivername”
       set appservers = /root/workers, ",";
}
log Apache_logs{ "/scratch/httpd/logs/error_log";
   "/scratch/httpd/logs/modjk_log";
 }
```

- Must convert config files to XML
  - Values the results of Xpath queries
- Elements have attached configs and logs
- Usage example:

```
… ws1.config[httpdconf].docroot == …
```

Stat primitive type

- Abstracts temporally sampled data
  - E.g.: CPU load, packets through interface
- Appears as an element field
- Statistical properties:
  - Mean, median, exponential weighted, variance, linear regression (slope/intercept)
  - Each is a single real value
- Sampling properties:
  - Frequency, number of samples

Tasks

- Method to abstract human actions
- Only way to specify sequential execution
- A task is a set of assertions separated by wait statements
- Waits have:
    - a timeout
    - else clause if timeout fails
- Assertions may be scoped task only or global
  - Task only valid during task

Task Example

```
Task Add_ApplicationServer {
   name = "Add application server"; } {

   var ws_all_cfg_1_appservers_before =
   ws_all..config[workerprop].appservers;

   call balanced;   // call a named assertion

   wait("Begin Task") { timeout = 300000; freq = 1.0; }
   else{ log(“operator abandoned task”); break; };

   wait("Begin Validation!")  timeout = 300000; freq = 1.0; }
        else{ break; };

   assert appserversSuperSetOfJvmRoutes
   (ws_all..config[workerprop].appservers.superset(as_add.config[serve
   rXML].jvmroute) )
   {taskonly; } else{ };
```

Execution Model

- Assertions checked using specified timing
  - Assertion can fire at own rate
  - Also fires at rate of the parents
- Sequential execution specified in tasks
- Waits can be for a boolean expression to become true, or for an operator to click a button.

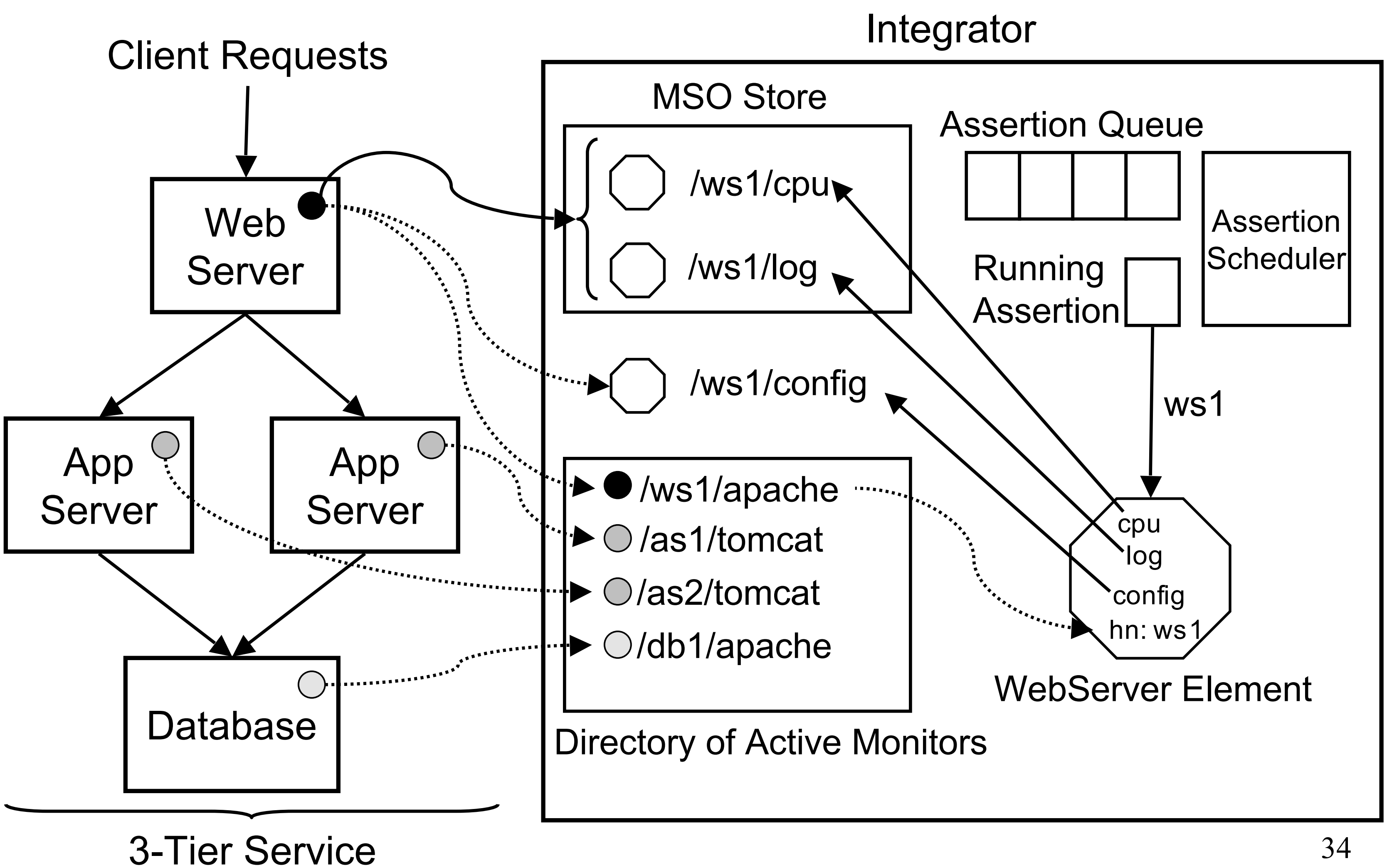
Talk Outline

- Motivation
- Human aware system design
- Our approach: Validation
- 3 Model paradigms
- A language overview
  - Language and implementation
- Using A
- Evaluation approaches
- Conclusions & future work

Runtime Architecture

- A program compiled to Java classes
    - Must use elements with a run-time definitions
    - Configs in XML
    - Logs are text files
- Adding new Elements/config/logs type means writing a new driver
    - Defined API to rest of system
    - Must be able to get access to system state
        - E.g. SNMP-like protocols

Run Time Architecture



Example compile and run

```
% /path/to/parser MainClass < source.a
% cp MainClass.java
  /path/to/vivo/source/aprograms
% cd /path/to/vivo/source/aprograms
% make
% /path/to/vivo/scripts/vivo restart
```

Talk Outline

- Motivation
- Human aware system design
- Our approach: Validation
- A language overview
  - Language and implementation
- Using A
- Evaluation approaches
- Conclusions & future work

Evaluation

- Hard to evaluate!
- Metric: How effective are A programs at signaling a faults during human interactions?
- Measure cost and benefit:
  - How effective are a collection of programs?
  - How difficult is it to write such code?
  - Evolve with the system?

Evaluation Strategy

- Create models of a service
- Write A programs for various tasks
- Create a representative set of mistakes
- Evaluate program's ability to catch mistakes on these tasks

Test Service and A program

- Service: 3 tier auction (RUBiS from Rice U.)
- A program:
  - 8 Libraries
  - 49 assertions in the libraries
  - 749 lines in the libraries
  - 4 tasks
    - 125 lines in the tasks
  - 874 lines total
- Small size of tasks encouraging result

Operations Tasks

- Add an upgraded web server
- Add an upgraded application server
- Add a load balancer
- Add a database to the DMBS

Mistake Injection Experiments

- 11 representative mistakes
  - Subtle, non-obvious, realistic
- Sources:
  - Previous human factor study of live operators
  - Survey of DMBS administrators
  - Reports in the literature
- None would have been caught with prior work on replica or trace based validation

Mistakes (I)

1. LVS ARP Problem
2. Web-server not compiled with membership protocol
3. Time-to-Live of membership heartbeat wrong
4. Wrong port numbers on webserver
5. Number of connections to DB exceeded
6. Wrong front end load balancer policy
   - Least Connections vs. Round Robin

LVS ARP problem

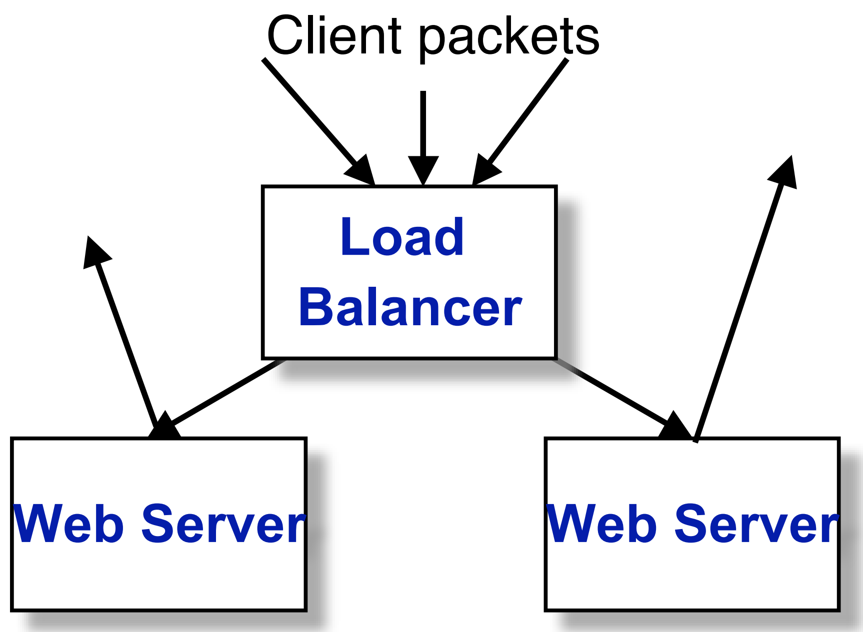
- Load balancer distributes incoming packets

- Web servers send outbound packet directly to clients, reducing traffic on balancer

- All must share 1 IP address

- Web servers must be set to ignore ARP requests for shared IP address

- Failure results intermittent loss of requests



Mistakes (II)

7. Web server load balancer misconfigured
8. DMBS performance parameters set too low
9. DB admin account has no password
10. Any machine can access the DB
11. Allowing any user to grant/revoke privs on the DB.

Results

- Caught 10 of the 11 mistakes
  - Uncaught: web server not complied with support for membership protocols
  - Assertion must check of the web-server is attached to the correct shared memory segment.
- Example points to bottom-up approach
  - Write assertions for known mistakes/faults so they do not happen again

Future Work

- More experience:
    - Production systems (or a copy)
    - Production tasks
    - Other large/complex systems
    - More mistakes
- Actuation: What to do when an assertion fails?
    - Low-level assertions may not be that important
    - When to ignore? How to prioritize?
- Monitoring: Can we tell when a fault occurred?
- Diagnosis: Can low-level assertion failures help pinpoint problems?
- Higher-level human interaction?
    - E.g., visual programming/diagramming

Conclusions

- First step to make systems more robust to human mistakes
- New programming language to increase availability
- Catches subtle, non-obvious mistakes
- Appears to be a good match
  - Needs more actual use to evaluate

Backup slides

A language definition (1)

A Program: Common Syntax

Syntax	**Meaning**	**Example**
::	binding	See here
assert <name> (<conditional>){ <assertproperties>} else{ <actions>}	assertion definition	See here
task <name> { name=<taskname>} { <vars><waits><assertions>}	task definition	See here
var <varname> = <property>	stores current value of <property> in <varname>	var utilBefore = ws_one.cpu.utilization
wait(”<somestring>”) { <waitproperties> } else{ <action in the event of timeout>}	wait on user action	See here
wait(<conditional>) { <waitproperties> } else{ <action in the event of timeout>}	wait on condtion to be true	See here

A Language definition (2)

break;	halt an operator task	break;
.	element separator	ws_one.cpu
..	group element separator	ws_all..cpu
+,-,/,*	arithmetic operators	ws_one.cpu.utilization >= 0.80 * ws_two.cpu.utilization
==,>,<,>=,⇐	logical operators	ws_one.cpu.utilization == ws_two.cpu.utilization
EQUALS()	aggregate equal to (all elements equal)	EQUALS(ws_all..cpu.utilization)
COLLECT()	used to collect all values in an aggregate	see usage below
SUM()	aggregate sum	SUM(COLLECT(ws_all..cpu.utilizatio n))
MEAN()	aggregate mean	MEAN(COLLECT(ws_all..cpu.utilizati on))
.config[”<filevar>”].<paramname>	config parameter value in the config file bound to the variable filevar	ws_one.config[httpconf].portnumber
.contains()	checks if value appears in log file	ws_one.log[httplog].contains(”httpd started”)
.without()	opposite of .contains	ws_one.log[httplog].without(”Error message”)