Stateful Detection in High Throughput Distributed Systems

Saurabh Bagchi
The Center for Education and Research in Information Assurance and Security (CERIAS)
School of Electrical and Computer Engineering
Purdue University


Joint work with:
Gunjan Khanna (alumnus), Ignacio Laguna, Fahad Arshad, Nawanol Ampornpunt, Samuel Midkiff

PURDUE UNIVERSITY

A Little Bit About Purdue University

- Land grant institution founded in 1869
  - 3,000 faculty members
  - 38,000 students
- Located about 2.5 hours by road from Chicago and about an hour's drive from Indianapolis
- Operating budget: Roughly $1.1 Billion per year
- Electrical and Computer Engineering:
  - 80 faculty members
  - 900 undergraduates
  - 650 graduate students
  - Consistently ranked in the top 10 US News and World Report rankings
- Computer Science
  - 45 faculty members
  - 150 graduate students

PURDUE UNIVERSITY

Computer Engineering (CE) at Purdue

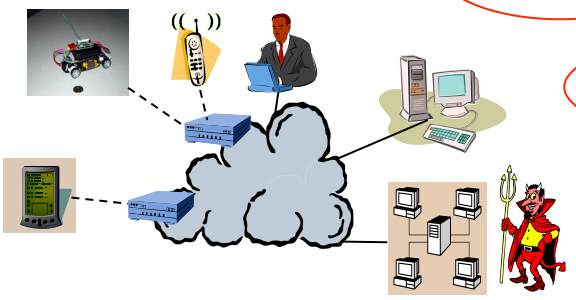
- There are 18 core faculty members in CE
- The areas of research within CE can be broadly classified as
  - Compilers: Eigenmann, Midkiff
  - Distributed Systems/Networking/Operating Systems: Hu, Rao, Bagchi
  - Computer Architecture: Vijaykumar, Pai, Thottethodi
  - Embedded Systems: Raghunathan, Lu
  - Dependability/Security: Ghafoor, Bagchi
  - Artificial Intelligence/Machine Learning/Vision: Givan, Siskind, Kak
  - Graphics/Visualization/Haptics: Ebert, Elmqvist, Tan
- We build and test systems, break systems, work together, publish at the top conferences, and have a whale of a time while we do this
- Consistently ranked in the top 10 (US News Rankings)

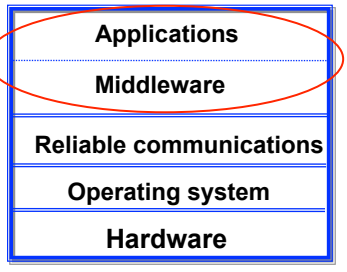
PURDUE UNIVERSITY

Dependable Computing Systems Lab (DCSL)

- URL: www.ece.purdue.edu/~dcsl
- We need computer systems that we can depend on in the face of
  - Naturally occurring faults – hardware malfunction, software bugs
  - Malicious intrusions – insider attack or external adversaries

End-to-End Dependability = Network-level issues + Node-level issues





PURDUE UNIVERSITY

Research Projects in DCSL

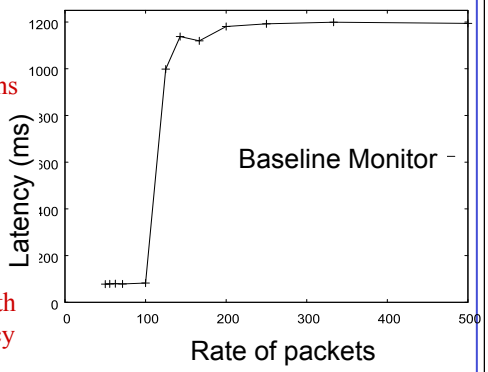
- **Framework for distributed intrusion tolerant system**
  - How to build an adaptive infrastructure for diagnosing and recovering from failures in a distributed platform?
  - Application: Distributed e-commerce application, Voice over IP system
- **Black-box diagnosis**
  - How to diagnose source of errors in high-throughput distributed apps?
  - Application: Distributed e-learning application
- **Dependable ad-hoc and sensor networks**
  - How to build dependable network out of inherently unreliable components with resource constraints?
  - Application: Monitoring environmental conditions in urban areas

Outline

- Motivation
- Monitor Based Error Detection
- Sampling Approach
- Experiments and Results.
- Related Work
- Conclusions and take-away lessons

Motivation

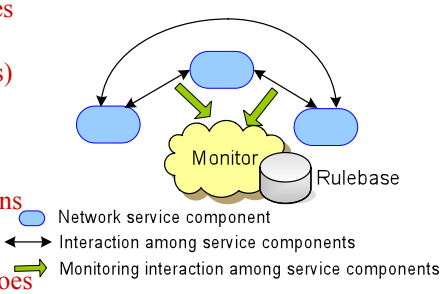
- Ever increasing bandwidth has led to proliferation of distributed applications with high throughput streams
- Error detection framework must therefore handle high throughput streams
- Our previous detection mechanism breaks beyond a particular incoming packet rate
- The goal is to push the knee to right with graceful degradation of detection latency and accuracy



PURDUE UNIVERSITY

Monitor Approach to Detection

- The approach follows an observer-observed model
- The observer (the Monitor) only observes external message interactions between components called protocol entities (PEs)
- The Monitor maintains a set of anomaly based rules to verify the PEs
- The Monitor estimates the state transitions of the PEs
- On an incoming message, the Monitor does the appropriate state transition and matches the state specific rules in the rule base
- A violation of the rule would flag an error leading to a detection of failure



PURDUE UNIVERSITY

Design Goals

- Online mechanism enforcing low latency and high accuracy
  - Graceful degradation of latency and accuracy
- Treat protocol entities as Black-box
  - Non-intrusive approach
  - Operate asynchronously with respect to application
- Monitor should be executable on off-the-shelf hardware
  - Should not have large memory footprint
  - The computation should scale slowly with the number of PEs
- Stateful approach should be followed
  - Natural errors in systems are stateful
  - Example: Failures in Windows NT
  - Example: Failure prediction in cycle-sharing systems

Detection Framework

Step1: On observing a message, perform state transition and if required, update variables corresponding to PE`s FSM

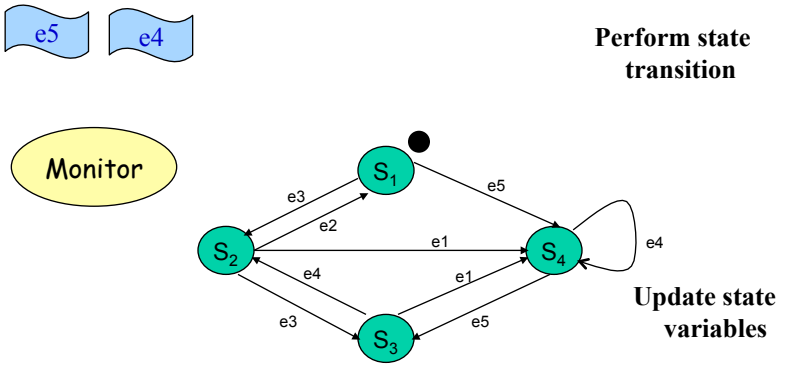
Step2: Perform rule matching for the rules associated with the particular state and message combination

Step3: Monitor flags an error if rules don`t match

For details:

Gunjan Khanna, Padma Varadharajan, and Saurabh Bagchi, "Automated Online Monitoring of Distributed Applications through External Monitors," IEEE Transactions on Dependable and Secure Computing (TDSC), vol. 3, no. 2, pp. 115-129, Apr-Jun, 2006.

Detection Framework



Example Finite State Machine (FSM)

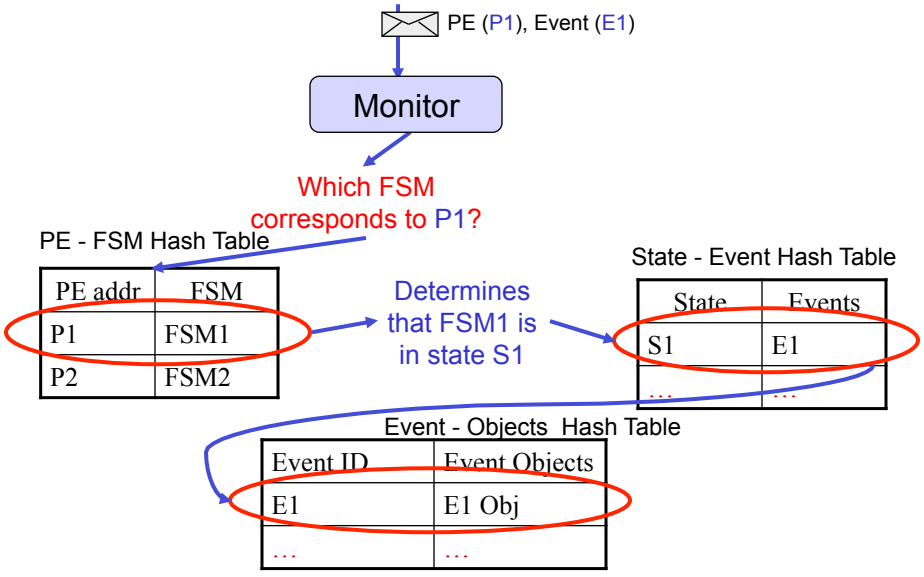
Rule Matching

- **Monitor-Baseline** has linear structures that it needs to traverse for rule matching
- Rules are defined based on protocol specifications and QoS requirements.
- Rules are anomaly based.
  - Define the correct behavior of the protocol
- Five generic temporal rule categories
  - Example:
    - The data message count should be between 10 and 30 for the next 5000 msec. (*QoS*)
    - Sender should receive an Ack after sending 32 Data packets (*protocol specification*)

Solution: Improve Per Message Processing (Monitor-HT)

- Rationale: Make processing of each incoming message more efficient
- Solution Approach:
  - Provide efficient look-up using hashtables
  - Eliminate duplicate copies of the state variables
- Finite State Machine is organized in a multi-level hashtable
  - Monitor-H has a constant order look-up while Monitor-Baseline was linear in the number of PEs being verified and number of events in each state

Lookup in multi-level Hashtable



Efficient Rule Matching – *Monitor-HT*

Data

Rule 1	[illegible]
Rule 2	[illegible]
Rule 3	[illegible]

Previous Approach

Data

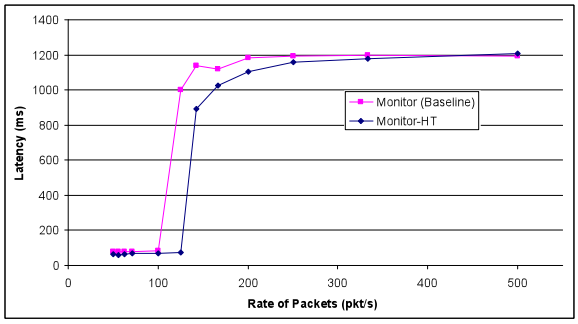
State Var	[illegible]
Rule 1	1
Rule 2	0
Rule 3	1

New Approach

- Multiple rules are matching the same message type
  - Local variables contain snapshots of the global count at instantiation and at matching instant
  - *PE* × *Event ID* tuple is only incremented once
  - Single copy update of state variables.

PURDUE UNIVERSITY

Monitor-HT versus Monitor-Baseline



- We compare the latency of detection of Monitor-Baseline and Monitor-HT. Latency is measured from instantiation of rule to the end of rule matching
- Monitor-HT achieves a 25% higher breaking point in terms of rate of incoming packets

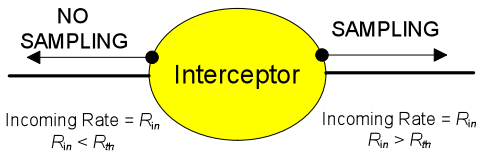
PURDUE UNIVERSITY

Solution: Sample Messages for Detection

- Rationale:
  - Monitor-HT still has to perform a minimum constant amount of work for every incoming message.
  - It gets overwhelmed when the message rate is too high
- Solution Approach
  - Modify Monitor-HT to reduce the incoming workload
  - Sample incoming messages to perform matching on a subset
- Instead of processing every message, sample the incoming messages **(Monitor-S)**
  1. When do we do sampling ?
  2. How do we sample?
  3. How do we handle state non-determinism?

When do we Sample?

- Assume Monitor-Baseline achieves a desired latency and accuracy uptil a rate of incoming messages $R_{bp.}$
  - Choose a threshold R_{th} such that $R_{th} < R_{bp}$
- When the incoming rate R_{in} is such that $R_{in} > R_{th,}$ Monitor-S switches to sampling mode of operation
- Design tradeoff:
  - R_{th} is far below R_{bp}: Inefficient use of Monior resources
  - R_{th} is very close to R_{bp}: Small spike can make the system unstable



How do we Sample?

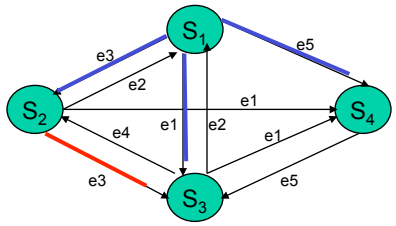
- We choose uniform random sampling: rate of sampling is dependent on the rate of incoming messages
  - Uniform random method is oblivious to the incoming message type
  - Any sampling approach based on the information of the incoming message will require some processing of the message before sampling
- Drop message at the rate of 1 in every $R_{in} /(R_{in} - R_{th})$ messages
  - Incoming rate is recalculated after a window of 30 seconds
- Scale the constants in the rules by a factor of R_{th} / R_{in}
  - Original rule — "Receive 10 Acks in 100 sec"
  - Rule modified due to sampling — "Receive 10.(R_{th} / R_{in}) Acks in 100 sec"

How do we handle Non-Determinism ?

- Dropping a message can cause Monitor-S to lose track of the current state of the PE
- Instead of keeping a single current state for each PE being verified, keep a vector of possible states the PE may be in
  - $\hat{S} = \{S_1, S_2 \ldots S_K\}$
- If r consecutive messages are dropped starting from state S_{start} then the state vector $\hat{S}$ consists of the union of states reachable in r steps from S_{start}
- Computing the state vector at runtime is expensive. So Monitor-S pre-computes state vectors offline

How do we handle Non-Determinism ?

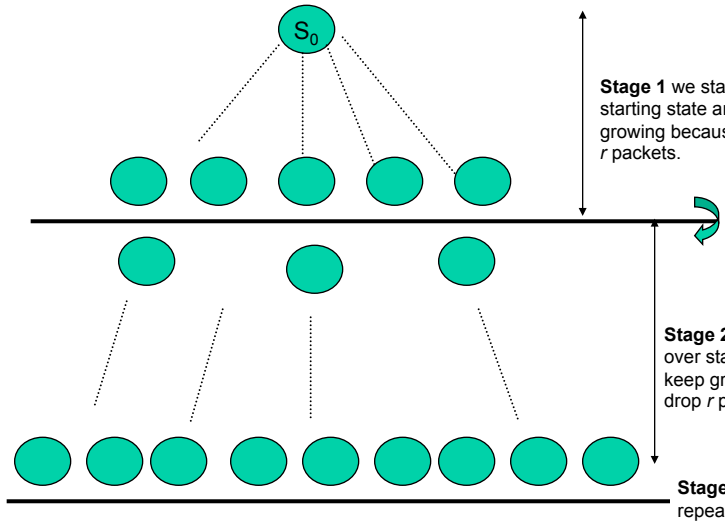
- Size of the state vector does not keep growing
  - Bounded by the total number of states
  - Sampling of a message causes state vector size to reduce since message is only possible in a few of the states of the state vector
- Example: Consider the FSM below
  - At start: $\hat{S} = \{S_1\}$
  - Drop a message: $\hat{S} = \{S_2, S_3, S_4\}$
  - Sample a message (say e3): $\hat{S} = \{S_2\}$



Example Finite State Machine (FSM)

PURDUE UNIVERSITY

Stages of Sampling Approach



Stage 1 we start with one starting state and keep growing because we drop *r* packets.

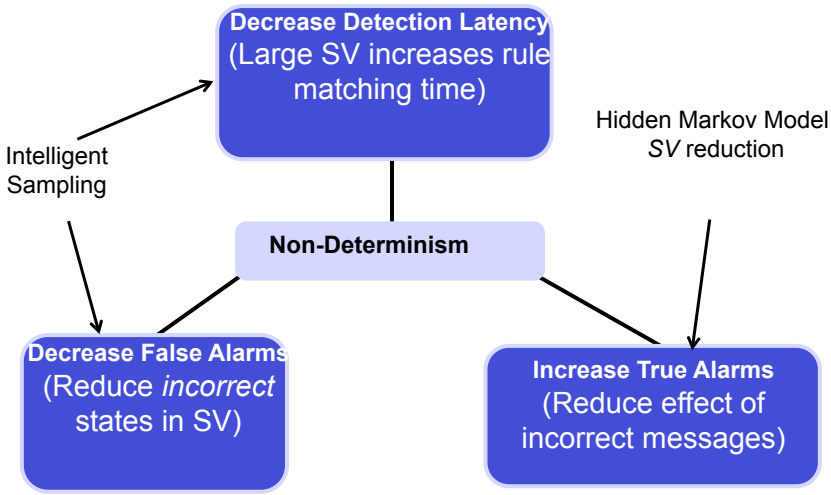
At this boundary we sample a packet and reduce the states to some number j

Stage 2 we start with left over states and again keep growing because we drop *r* packets.

Stage 3 the system repeats itself in the same manner as above…….

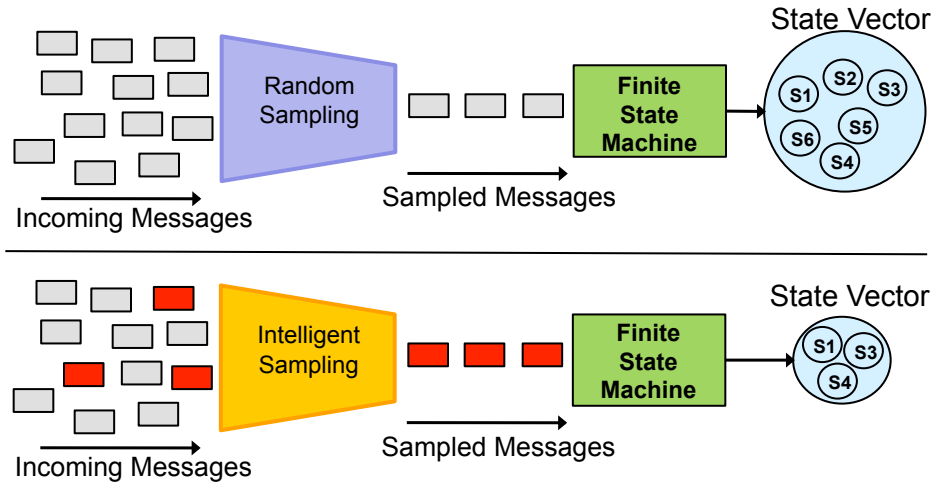
PURDUE UNIVERSITY

Challenges with Non-Determinism



PURDUE UNIVERSITY

Intelligent Sampling



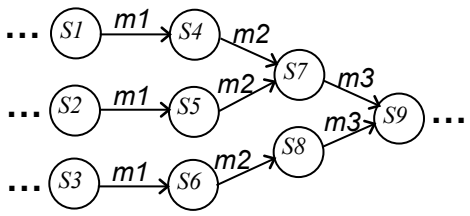
Message with *desirable* property

PURDUE UNIVERSITY

What is a Desirable Property of a Message?

Suppose, *SV* = { *S1*, *S2*, *S3* }, *Sampling Rate* = 1/3

A portion of a Finite State Machine



Sampled Message	SV	Discriminative Size
m1	{S4, S5, S6}	3
m2	{ S7, S8 }	2
m3	{ S9 }	1

Discriminative Size — Number of times a message appears in transitions to different states in the FSM

- Desirable property is a small *discriminative size*

PURDUE UNIVERSITY

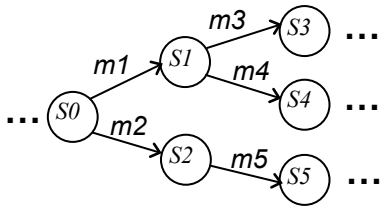
Benefits of Intelligent Sampling

Random Sampling	Intelligent Sampling
• *SV* can grow into large size	• *SV* is kept small – Detection latency reduction
• Multiple incorrect states – Increase of false alarms	• Less incorrect states in *SV* – False alarms reduction

PURDUE UNIVERSITY

The Problem of Sampling an *Incorrect Message*

- What if an *incorrect* message is sampled?
  - The message is *incorrect* in current states, e.g., a message from buggy component



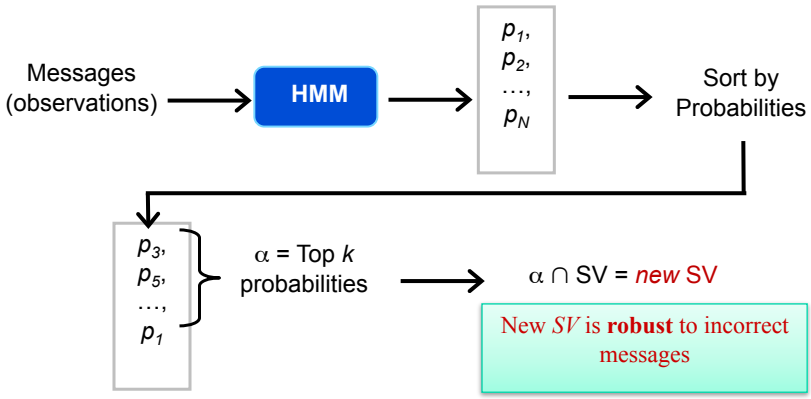
- Suppose $SV = \{ S1, S2 \}$ and *m3* is changed to *m5*
  $\Rightarrow SV = \{ S5 \}$ (incorrect *SV*!, it should be $\{ S3 \}$)

Probabilistic State Vector Reduction: *A Hidden Markov Model Approach*

- Hidden Markov Model (HMM) used to reduce SV
  - An HMM is an extended ***Markov Model*** where states are not observable
  - States are hidden as in the monitored application
- Given an HMM, we can ask:
  The probability of the application being in any state, given a sequence of messages?
  Cost is $\mathbf{O(N^3L)}$, N: number of states L: sequence length
- The HMM is trained (offline) with application traces

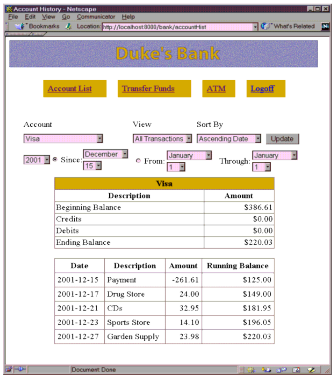
State Vector Reduction with the HMM

- Monitor asks the HMM: $\{p_1, p_2, \ldots, p_N\}$
  $p_i = P(S_i \mid O)$, S_i: application state i, O: observation sequence



PURDUE UNIVERSITY

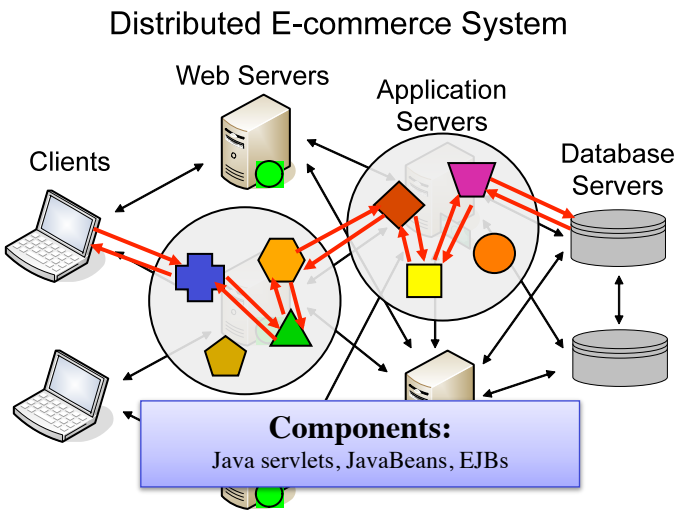
Experimental Testbed: Java Duke's Bank Application



- Simulates multi-tier online banking system
- User transactions:
  - Access account information, transfer money, etc.
- Application stressed with different workloads
  - Incoming message rate at Monitor varies with user load

PURDUE UNIVERSITY

Web Interaction: A Sequence of *calls* and *returns*



PURDUE UNIVERSITY

Error Injection Types

Error Type	Description
Response Delays	a response delay in a method call
Null calls	a call to a component that is never executed
Unhandled Exceptions	exception thrown by execution that is never caught by the program
Incorrect Message Sequences	change randomly the web interaction structure

- Errors are injected in components touched by web interactions
  - A web interaction is faulty if at least one of its components is faulty

PURDUE UNIVERSITY

Performance Metrics Used in Experiments

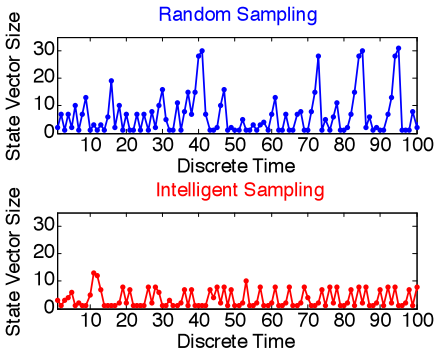
Accuracy (True Alarms)	% of *true* detections out of web interactions where errors were injected
Precision (False Alarms)	% of *true* detections out of the total number of detections
Detection Latency	time elapsed between the error injection and its detection

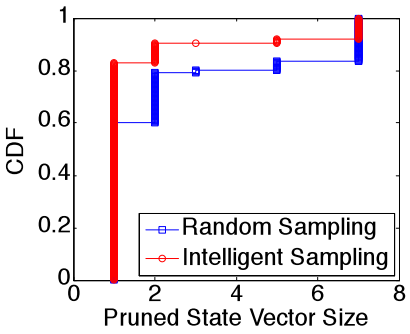
Example:

	Web Interactions				
	1	2	3	4	5
Error Injected		X	X		X
Detection (An alarm is signaled)	X	X		X	X

Accuracy = 2/3 = 0.67 **Precision** = 2/4 = 0.5

Results: State Vector Reduction

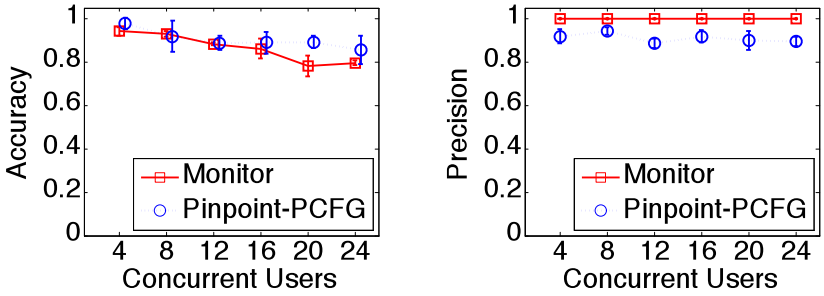




- Peaks are not observed in intelligent sampling (IS)
  - IS capability of selecting messages with small discriminative size
- SV of size 1 is more frequent in IS

Results: Monitor vs. Pinpoint (Accuracy, Precision)

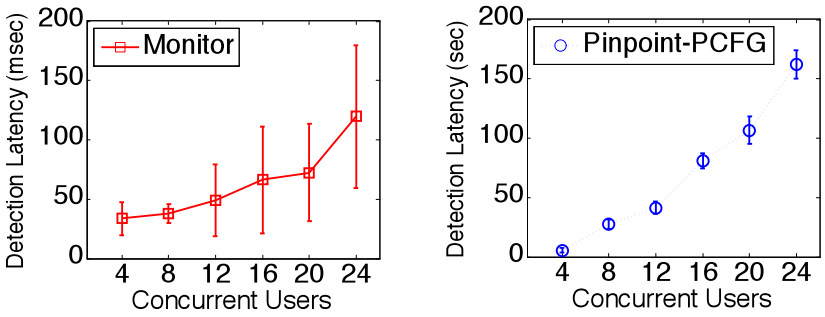
- **Pinpoint** (*NSDI* '04), traces paths though multiple components
- Use of **PCFG** to detect abnormal paths



- Monitor and Pinpoint expose similar levels of accuracy
- Precision in Monitor (1.0) is higher than in Pinpoint (0.9)

PURDUE UNIVERSITY

Results: Monitor vs. Pinpoint (Detection Latency)



- Detection latency in Monitor is in the order of **milliseconds**, while in Pinpoint is in **seconds**
- The PCFG has a high space and time complexity

PURDUE UNIVERSITY

Results: Memory Consumption (MB)

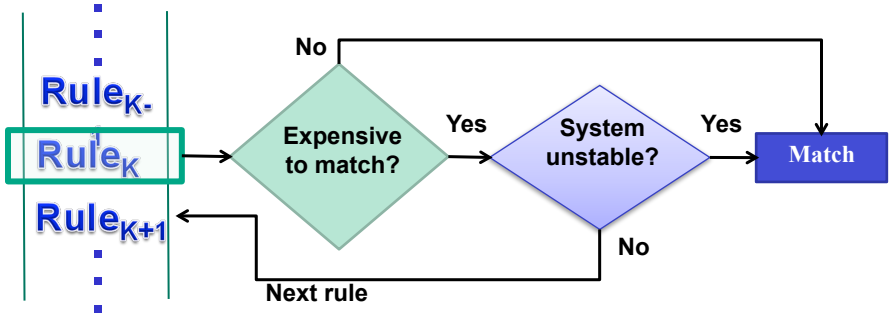
	Virtual Memory	RAM
Monitor	282.27	25.53
Pinpoint-PCFG	933.56	696.06

- Monitor does not rely on large data structures
- PCFG in Pinpoint has high **space** and **time** complexity
  - $O(RL^2)$ and $O(L^3)$

  R: number of rules in the grammar

  L: size of a web interaction
- Pinpoint thrashes due to high memory requirements

Efficient Rule Matching



- Selectively match computationally expensive rules
  - Expensive rules don't have to be matched all the time
- Rules are matched only if instability is present

Efficient Rule Matching Example: *Detecting Memory Leak*

- Efficiently detect memory leak in Apache web server
  - Memory leak injected probabilistically with web requests
- Expensive ARIMA-based rule to detect abnormal memory usage
  - Average matching latency reduced

Rule Matching Criteria	Memory Leak Detected	Average Matching Latency (msec.)
Always matched	yes	19.283
$\sigma \geq 0.5$	yes	7.115
$\sigma \geq 1.0$	no	1.25

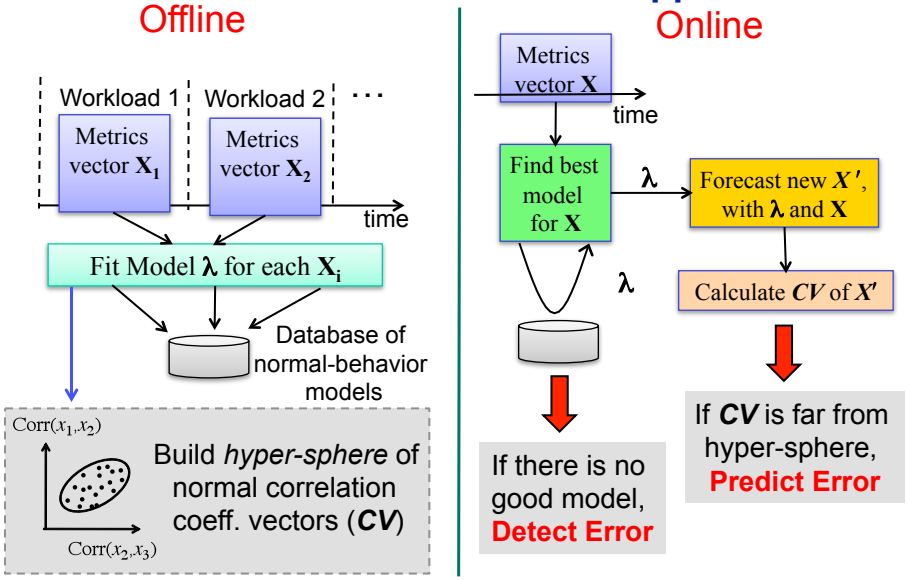
PURDUE UNIVERSITY

Error Prediction in Distributed Applications

- Goal
  - Complex distributed systems with many interacting components
  - Detect an error promptly at runtime before it is visible to the user
  - Predict an impending failure through runtime observations
- State-of-the-art
  - Threshold-based detection
  - Look at correlations of pairs of pre-defined metrics
  - Can detect an error but cannot predict impending failure
- Our Solution Approach
  - Machine learning algorithms to learn normal behavior
  - Behavior captured by trends in correlations of system and application metrics
  - Metric values predicted in the future
  - If future predicted values fall outside normal behavior, failure is predicted

PURDUE UNIVERSITY

Error Prediction in Distributed Applications



PURDUE UNIVERSITY

Conclusions

- We developed a stateful detection mechanism that can scale to a high data rate of the application protocol
- We extend an existing detection approach (Monitor-Baseline) by identifying inefficiencies in the rule matching process to reduce per packet processing overhead at the Monitor
- We use a sampling approach to reduce the number of packets being processed and Markov model to dampen the effect of incorrect messages
- We are working on predicting impending failures by modeling and predicting correlations between multiple metrics

PURDUE UNIVERSITY

The End

अंत

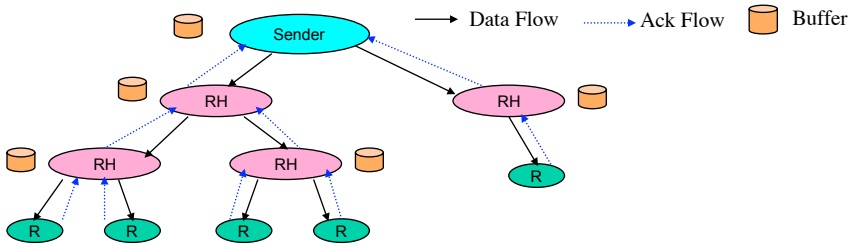
PURDUE UNIVERSITY

Backup Slides

PURDUE UNIVERSITY

Demonstration: TRAM

- We demonstrate the use of the Monitor on TRAM, *a tree-based reliable multicast protocol.*
- TRAM consists of a single sender, multiple repair heads (RH), and receivers. It ensures reliability of message transfer in case of node or link failures and message errors.
- We emulate TRAM protocol, where sender and receiver are the PEs being verified by the Monitor in all experiments.



PURDUE UNIVERSITY

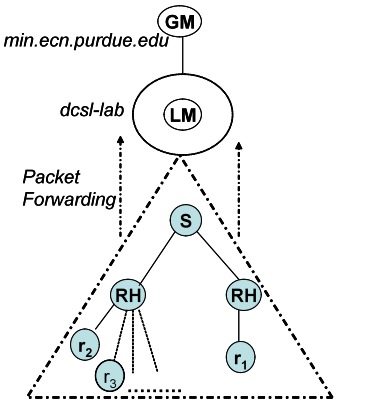
Failure Injection

- We perform *random fault injection* in the header of the emulated TRAM messages to induce failures
- We randomly choose a header field and change it to a randomly selected value, emulating protocol errors
  - Say, too many NACK messages are sent by the receiver
- To mimic faults close to reality, a burst length is chosen since TRAM is robust to isolated faults
  - A PE to inject is chosen (sender, RH or receiver) and faults are injected for a burst length of 500ms after every 5 minutes.

PURDUE UNIVERSITY

Topology and Metrics for Experiments

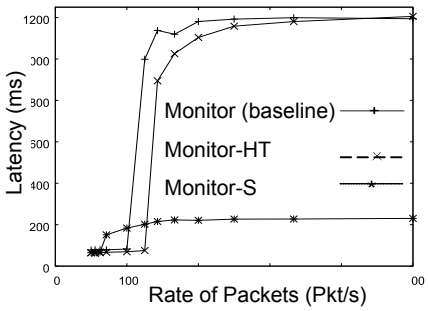
- *Accuracy* = (1- % of missed detections)
- Fault injections undetected by the Monitor are called *missed detections*
- *False detections* are errors flagged by Monitor but which do not affect TRAM entities
- *Latency* is measured as the time from the instantiation of a rule to the time when the rule matching is completed

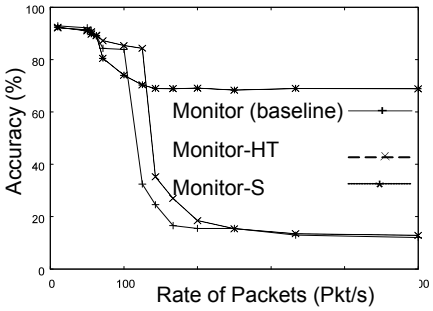


Physical Topology of the TRAM emulator and the Monitor

PURDUE UNIVERSITY

Results: Latency and Accuracy with packet rate (R_{in})

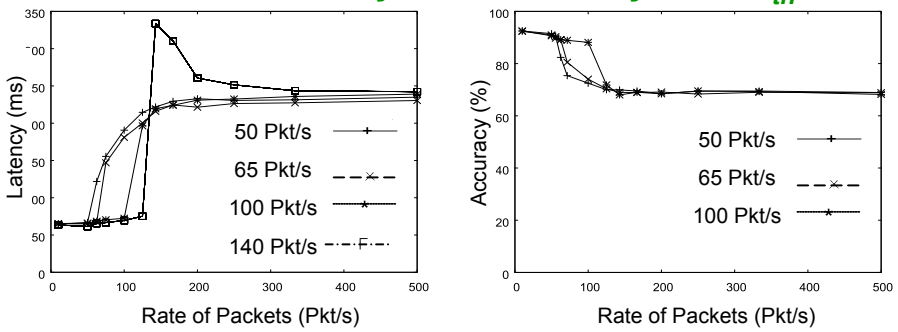




- Montior-Baseline and Monitor-HT break causing a knee. Monitor-HT`s knee is beyond Monitor-Basline`s knee
- Monitor-S shows a relatively smooth degradation in both accuracy and latency results
  - It is able to adjust the workload based on the rate
- Montior-S gives a low latency at the cost of small reduction in accuracy at high input rates
  - Accuracy suffers a little because of non determinism

PURDUE UNIVERSITY

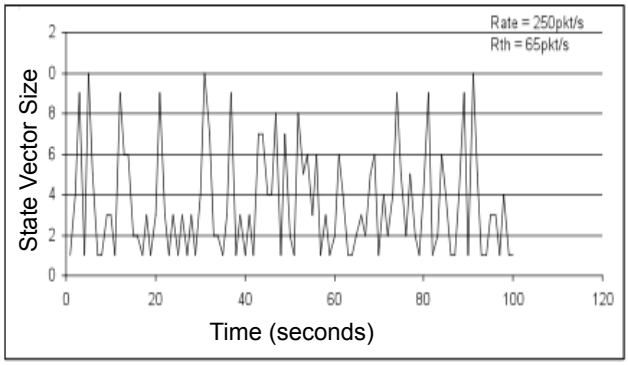
Results: Latency and Accuracy with R_{th}



- For $R_{th} < R_{bp}$ (R_{bp}=125 pkt/s), latency increases slowly before smoothing to a constant value
- For the curve of R_{th} = 140 pkt/s, the jump in latency is because the incoming rate is greater than R_{bp} and Monitor switches from sampling to non-sampling modes after the breakpoint
- The jump in latency translates to a sharp drop in accuracy in the accuracy plot
- A low R_{th} (50 pkt/s, 60 pkt/s) kicks in sampling early reducing accuracy. For R_{th} =100pkts/s the accuracy is high for a larger incoming rate

PURDUE UNIVERSITY

Results: Variation of State Vector Size



- In Region 1, |Ŝ| drops in steps from 9 to 6 and finally to 1. The drop in |Ŝ| is because of the unique possibility of the sampled event in only some of the states
- In Region 2, |Ŝ| increases from 1 to 3 because of a message drop

PURDUE UNIVERSITY

Related Research

- Change Detection in Networking
  - Sketch based approaches: Deltoids, Infocom'05, Infocom'06
  - Develop statistical models to describe the stream behavior.
  - In Monitor state of the application is closely examined and it accounts for spikes as well. Provides flexibility to switch to sampling or no-sampling
- Stateful Detection
  - Particular attention from the security community in building Intrusion Detection Systems,
  - Snort uses aggregated information from TCP packets to make decisions
  - SciDive provides stateful detection engine for VoIP, DSN`04
  - Restricted to the domain and focused on accuracy
- Detection in Distributed Systems
  - Heartbeats, watchdogs DSN `00
  - Detection of Failures using event graphs

PURDUE UNIVERSITY

Prediction using ARIMA

- **Step 1:** Out of *M* models we have from offline training, choose the one that fits the current patterns the best. This is done by dividing the observed window of measurements into two parts, and using each model to predict the latter part based on the former part. The best model is the one with lowest prediction error.
  - This step takes time O(NML(p+d+q)) where
    - N: number of metrics
    - L: length of observed window
    - M: number of models
    - p,d,q: parameters of the model
- **Step 2:** Predict new observation O_t using ARIMA(p,d,q) by evaluating

$$O_t = \mu + \sum_{i=1}^{pq} \phi_i O_{t-i} + \varepsilon_t \qquad (2)$$

  - This step takes time O(Nl(p+d+q)) where
    - N: number of metrics
    - p,d,q: parameters of the model
    - l: length of predicted window

PURDUE UNIVERSITY

Prediction using HMM

- **Step 1:** Out of *M* models we have from offline training, choose the one that fits the current patterns the best. This is done by using the forward algorithm to compute the likelihood of the observed window. The best model is the one that gives highest likelihood.
  - This step takes time $O(NMLS^2)$ where
    N: number of metrics
    L: length of observed window
    M: number of models
    S: number of hidden states
- **Step 2**: Find the distribution of current hidden state using Viterbi algorithm.
  - This step takes time $O(NLS^2)$ where
    N: number of metrics
    S: number of hidden states
    L: length of observed window
- **Step 3**: Make predictions by sampling B sequences.
  - This step takes time O(NBSl) where
    N: number of metrics
    S: number of hidden states
    B: number of sequences
    l: length of each predicted sequence

Classification

- Once we have one or more predicted sequences, we compute the pairwise correlation (e.g. Pearson correlation) for each pair of metrics, forming a correlation vector.
  - If we have more than one predicted sequence, we will have many correlation vectors
- Then, we compute the distance between the correlation vector and the "hypersphere" of correlation vectors based on normal behavior.
  - If the distance exceeds a set *threshold* then it is classified as abnormal.
- To build a hypersphere of correlation vectors, one-class SVM is used. The goal is to separate what is inside the area and what is outside.