REX: A Development Platform and Online Learning Approach for Runtime Emergent Software Systems

Barry Porter,
Matthew Grieves,
Roberto Filho
David Leslie

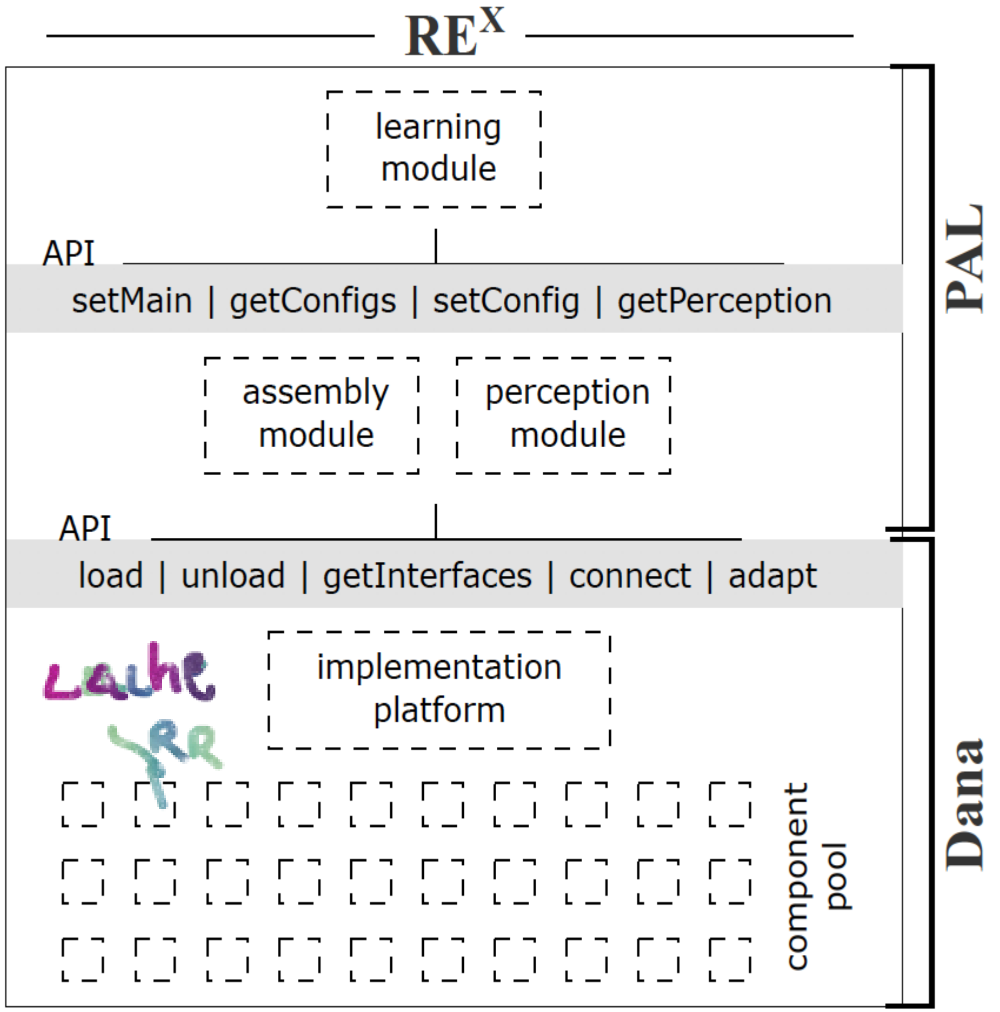
Introduction

- Designing, Analyzing and Maintaining – Millions of LOC:

  Is it sustainable?
- Software development: Models, Policies, and Processes
- Autonomous, Self-Adaptive, and Self Organized Software System
- Emergence of Software System - Autonomously from pool of available building blocks
- Responsive to actual runtime conditions.
- Can show rationale behind the choice

REX: Development Platform

- Implementation Platform-
  Dana (Dynamic Adaptive Nucleic Architecture)
- PAL Framework
  - Perceive – Internal + External conditions
  - Assemble and Re-assemble modules
  - Learn
- Online Learning – Statistical Linear Bandits, using Thompson Sampling.



Dana

- Component Based Software Paradigm
- All Components – Runtime Replaceable
- Multi-threaded imperative language (what and how)

Example: From Source code

```
component provides App requires io.Output out{
        int App:main(AppParam params[]){
                out.println("Hi! :-)")
                return 0
        }
}
```

```
interface File {
  transfer char path[]
  transfer int pos, mode

  File(char path[], int mode)
  byte[] read(int numBytes)
  int write(byte data[])
  bool eof()
  void close()
}
```

```
component provides App requires File {
  int App:main(AppParam args[]) {
     File ifd = new File(args[0].str, File.READ)
     File ofd = new File(args[1].str, File.WRITE)
     while (!ifd.eof()) ofd.write(ifd.read(128))
     ofd.close()
     ifd.close()
     return 0
  }
}
```

Figure 2 – Example interface to open, read and write files (top); and a component that uses this interface to copy a file (bottom).

Dana: Runtime Adaptation

Algorithm 1 Adaptation protocol

srcCom ▷ Comp. to rewire a required interface of
sinkCom ▷ Comp. with provided interface to wire to
intfName ▷ Interface name being adapted
pause(*srcCom.intfName*)
r_{objs} = getObjects(*srcCom.intfName*)
rewire(*srcCom.intfName*, *sinkCom*)
resume(*srcCom.intfName*)
for $i = 0$ **to** $r_{objs}.arrayLength - 1$ **do**
  if pauseObject($r_{objs[i]}$) **then**
    a = adaptConstruct(*sinkCom.intfName*, $r_{objs[i]}$)
    b = rewireObject($r_{objs[i]}$, a)
    resumeObject($r_{objs[i]}$)
    waitForObject(b)
    adaptDestroy(b)
  end if
end for

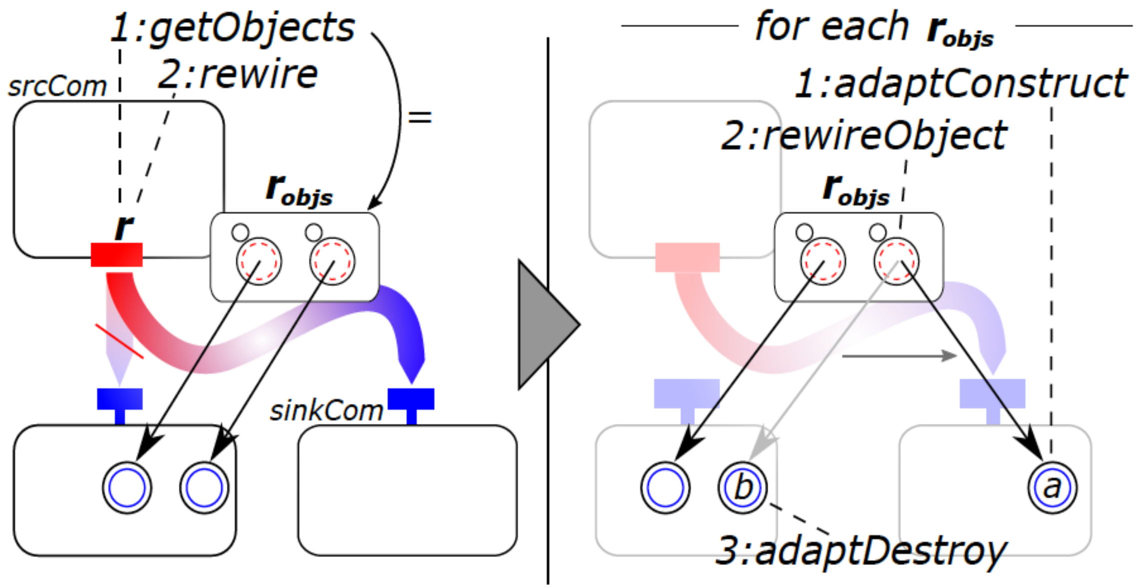


Figure 4 – Adaptation sequence overview. A selected required interface r is rewired, followed by each object in the set r_{objs}.

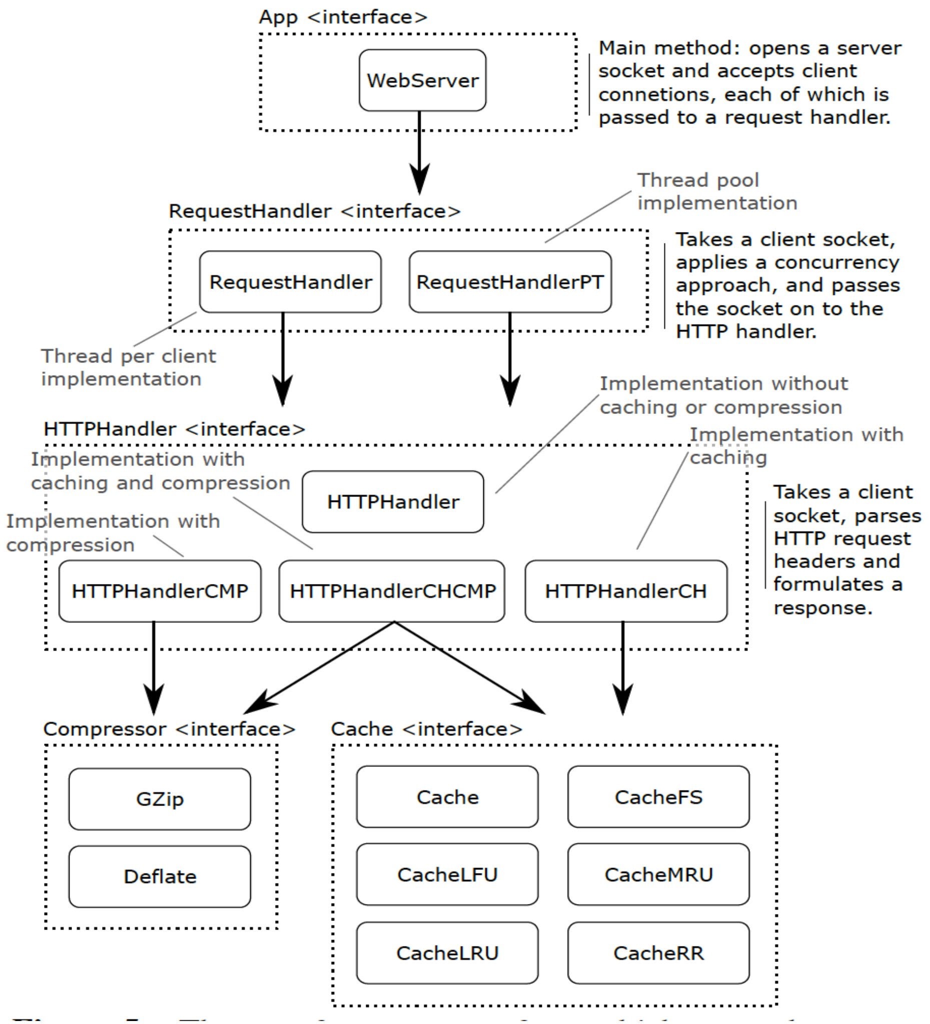
PAL Framework: Perception & Assembly

- Perception
  - Implemented using Recorder Interface
  - Data – Event and Metrics (Name, Value, Flag)

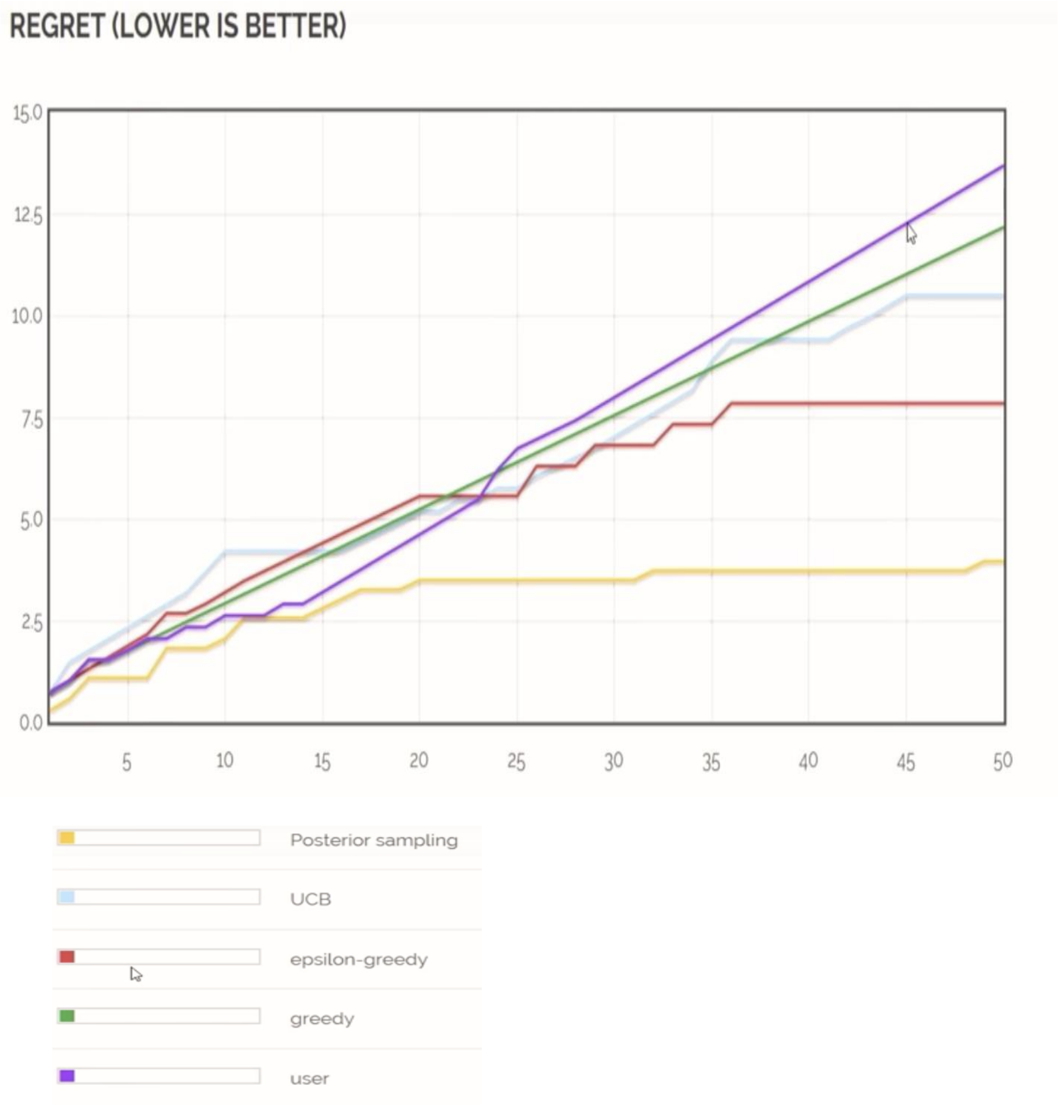
- Assembly
  - Starts with main component of target system
  - Read Required components (recursively)
  - Search interfaces in resources directory and their potential implementation
  - Example – Interface (io.File) → io (Implementation Directory)
  - Create a list of configurations
  - Use Adaptation protocol to reassemble

Sample Implementation: Webserver

- Number of components in system = 30
- File System, String Parsers,
- Number of configurations – 2*3*(2+5) = 42
- Request Handler – Avg response time
- HTTPHandler – Events for requested resource & their size



Exploration Vs Exploitation:



- ➢ Upper Confidence Bound Action Selection

$$A_t \doteq \underset{a}{\arg\max}\left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}\right]$$

- ➢ Greedy – Exploit current knowledge to maximize immediate reward
- ➢ Posterior Sampling- Thompson Sampling
  - ➢ Estimate posterior distribution using prior distribution

Multi-armed Bandit

- Arm – One configuration of webserver
- Action – choose one config and deploy

$$
\begin{aligned}
&\beta_0 + \beta_1 \mathbb{I}_{\text{RequestPT}} \\
&\quad + \beta_2 \mathbb{I}_{\text{HttpCMP}} + \beta_3 \mathbb{I}_{\text{HttpCH}} + \beta_4 \mathbb{I}_{\text{HttpCHCMP}} \\
&\quad + \beta_5 \mathbb{I}_{\text{Deflate}} \\
&\quad + \beta_6 \mathbb{I}_{\text{CacheFS}} + \beta_7 \mathbb{I}_{\text{CacheLFU}} + \beta_8 \mathbb{I}_{\text{CacheMRU}} \\
&\qquad + \beta_9 \mathbb{I}_{\text{CacheLRU}} + \beta_{10} \mathbb{I}_{\text{CacheRR}}
\end{aligned} \tag{1}
$$

$$
\begin{aligned}
x_{\text{conf}} = (1, &\mathbb{I}_{\text{RequestPT}}, \mathbb{I}_{\text{HttpCMP}}, \mathbb{I}_{\text{HttpCH}}, \mathbb{I}_{\text{HttpCHCMP}}, \\
&\mathbb{I}_{\text{Deflate}}, \mathbb{I}_{\text{CacheFS}}, \mathbb{I}_{\text{CacheLFU}}, \mathbb{I}_{\text{CacheMRU}}, \\
&\mathbb{I}_{\text{CacheLRU}}, \mathbb{I}_{\text{CacheRR}}),
\end{aligned}
$$
[3]

$$y = x_{\text{conf}} \beta + \varepsilon,$$

Algorithm 2 Learning Algorithm

```
//matrix of all available x_conf vectors (configurations)
actionMatrix = assembly.getConfigs()
X = new Matrix() //list of observed x_conf's to date
y = new Vector() //list of rewards seen for each X
n = 0
while running do
    //do linear regression & sample from posterior
    Λ = X^T X + Λ_0
    β = Λ^{-1}(Λ_0 β̃ + X^T y)
    a = a_0 + (n/2)
    b = b_0 + (y^T y + β̃^T Λ_0 β̃ − β^T Λ β) × 0.5
    σ^2 = new InverseGamma(a, b).sample()
    sample = new Normal(β, σ^2 Λ^{-1}).sample()

    //select the new configuration to use
    i = arg max(actionMatrix * sample)
    assembly.setConfig(i)

    //wait for 10 seconds, then record observations
    result = 1/perception.getAverageMetric()
    add row i of actionMatrix as new row of X
    add result as new element of y
    n++
end while
```

Handling Environment Changes

- Entropy –
  - ✓ High => Request for different resources
  - ✓ zero => Single resource requested repeatedly

- Text Volume – Highly Compressible

  Example – HTML, CSS Files

- High entropy interval – more than 50% request of high entropy
- 7 Extra Regression Coefficients
- Total number of configurations = 42*4 = 168

$$(1, \mathbb{I}_{\text{RequestPT}}, \mathbb{I}_{\text{HiEnt}}, \mathbb{I}_{\text{HiTxt}}, \mathbb{I}_{\text{HttpCMP(LowTxt)}}, \mathbb{I}_{\text{HttpCMP(HiTxt)}}, \mathbb{I}_{\text{HttpCH(LowEnt)}}, \mathbb{I}_{\text{HttpCH(HiEnt)}}, \mathbb{I}_{\text{HttpCHCMP(LowTxt,LowEnt)}}, \mathbb{I}_{\text{HttpCHCMP(HiTxt,LowEnt)}}, \mathbb{I}_{\text{HttpCHCMP(LowTxt,HiEnt)}}, \mathbb{I}_{\text{HttpCHCMP(HiTxt,HiEnt)}}, \mathbb{I}_{\text{Deflate}}, \mathbb{I}_{\text{CacheFS}}, \mathbb{I}_{\text{CacheLFU}}, \mathbb{I}_{\text{CacheMRU}}, \mathbb{I}_{\text{CacheLRU}}, \mathbb{I}_{\text{CacheRR}}).$$

Results – Runtime Adaptation

	Average	Maximum	Minimum
setConfig (idle)	509.60 ms	615.00 ms	397.00 ms
setConfig (busy)	1350.32 ms	5811.00 ms	510.00 ms
pause/resume (idle)	8.50 µs	9.94 µs	7.81 µs
pause/resume (busy)	13.22 µs	31.21 µs	8.51 µs
pauseObject/resumeObject (idle)	4.51 µs	5.34 µs	3.84 µs
pauseObject/resumeObject (busy)	28.54 µs	387.17 µs	4.35 µs
components adapted in setConfig()	1.22	3.00	1.00

Table 1 – Adaptation speed measured in different ways, from full configuration changes to individual component adaptations.

- Webserver is actually paused
- pauseObject – busy waiting for new function call
- pause – prevent new objects from being instantiated

Results: Divergent Systems

Entropy\Text	Low	High
Low	Cache	Cache & Compress
High	Default (Due to hash collision)	Cache & compression

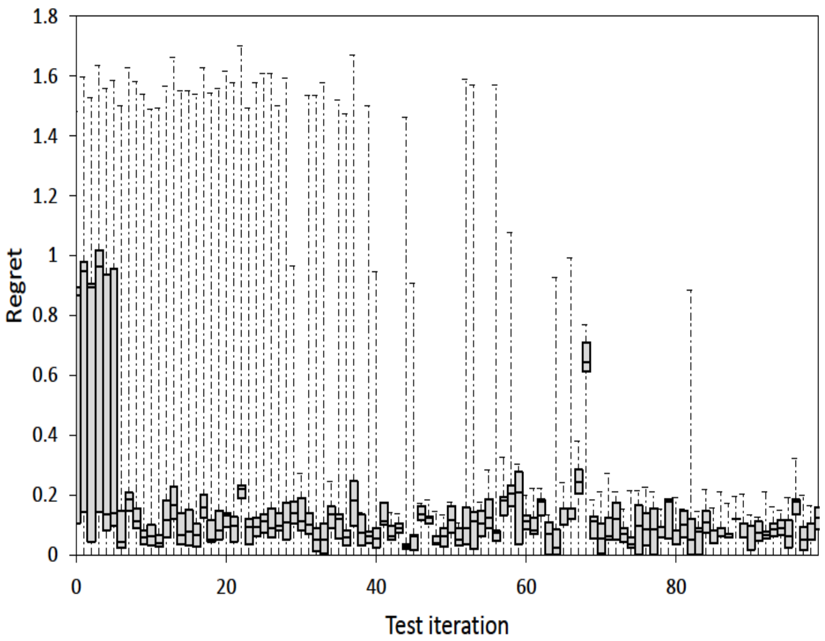
Request pattern	File size (b) [GZ]	Default	Caching	Caching & compression
Text *low entropy*	156,983 [12,757]	11.94 ms	9.56 ms	**0.70 ms**
Text *low entropy*	82,628 [11,949]	4.05 ms	**0.60 ms**	0.66 ms
Text *low entropy*	3,869 [1,930]	1.18 ms	**0.59 ms**	0.63 ms
Image *low entropy*	1,671,167 [1,667,464]	160.81 ms	**150.72 ms**	154.42 ms
Image *low entropy*	84,760 [66,914]	4.02 ms	**0.66 ms**	0.74 ms
Image *low entropy*	4,001 [3,895]	1.22 ms	**0.55 ms**	0.62 ms
Text *high entropy*	156,983 [12,757]	19.27 ms	19.66 ms	**3.04 ms**
Text *high entropy*	82,628 [11,949]	4.61 ms	3.27 ms	**3.07 ms**
Text *high entropy*	3,869 [1,930]	**1.25 ms**	2.93 ms	2.52 ms
Image *high entropy*	1,671,167 [1,667,464]	**156.50 ms**	156.64 ms	157.66 ms
Image *high entropy*	84,760 [66,914]	4.48 ms	3.19 ms	**2.94 ms**
Image *high entropy*	4,001 [3,895]	**1.30 ms**	2.90 ms	2.67 ms

Table 2 – Results of different configurations under different request patterns, showing average response times. The standard deviation throughout these results is low, at around 0.2.

Results:

- 1 test iteration = 10 second (1000 experiments)
- Large File => Less training samples

Small HTML Files with low latency



Large HTML Files with low latency

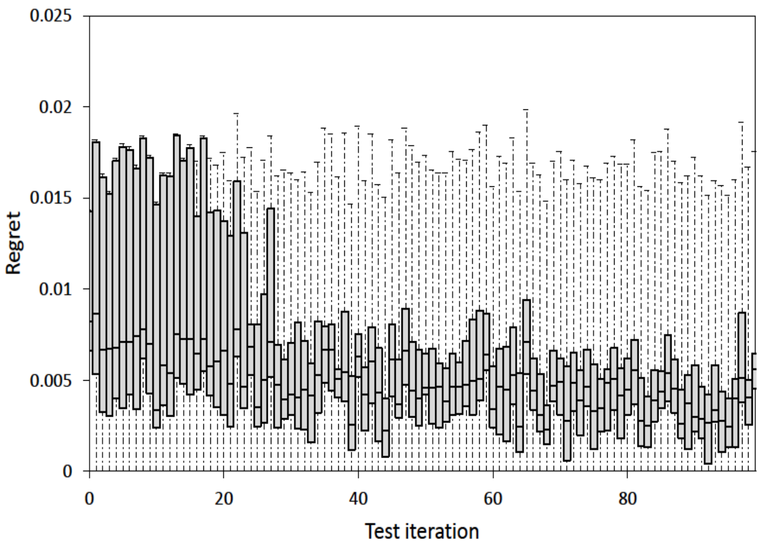


Figure 7 – Learning using response times to large text files, with adjusted prior values for $\tilde{\beta}_0$ and b_0.

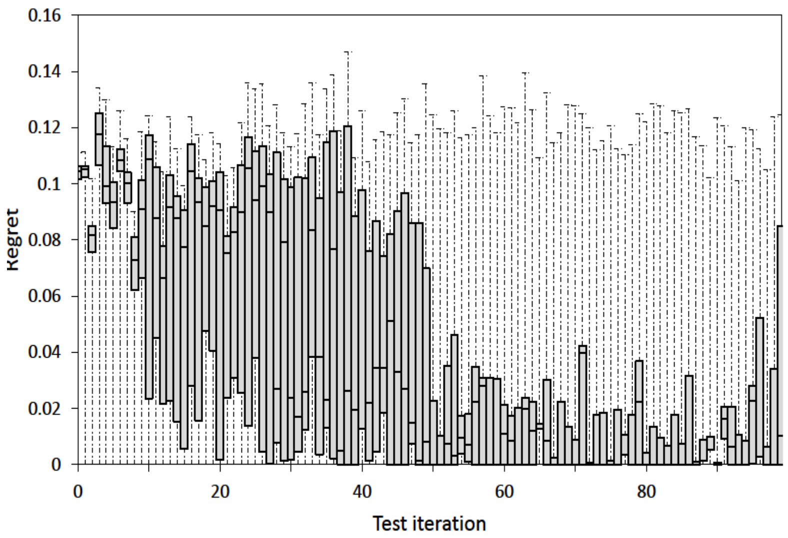


Figure 9 – Learning using response times to a realistic (and ighly varying) request pattern, using the NASA server trace [2].

Results: Alternating Request Pattern

- Left - Constantly forget and re-learn

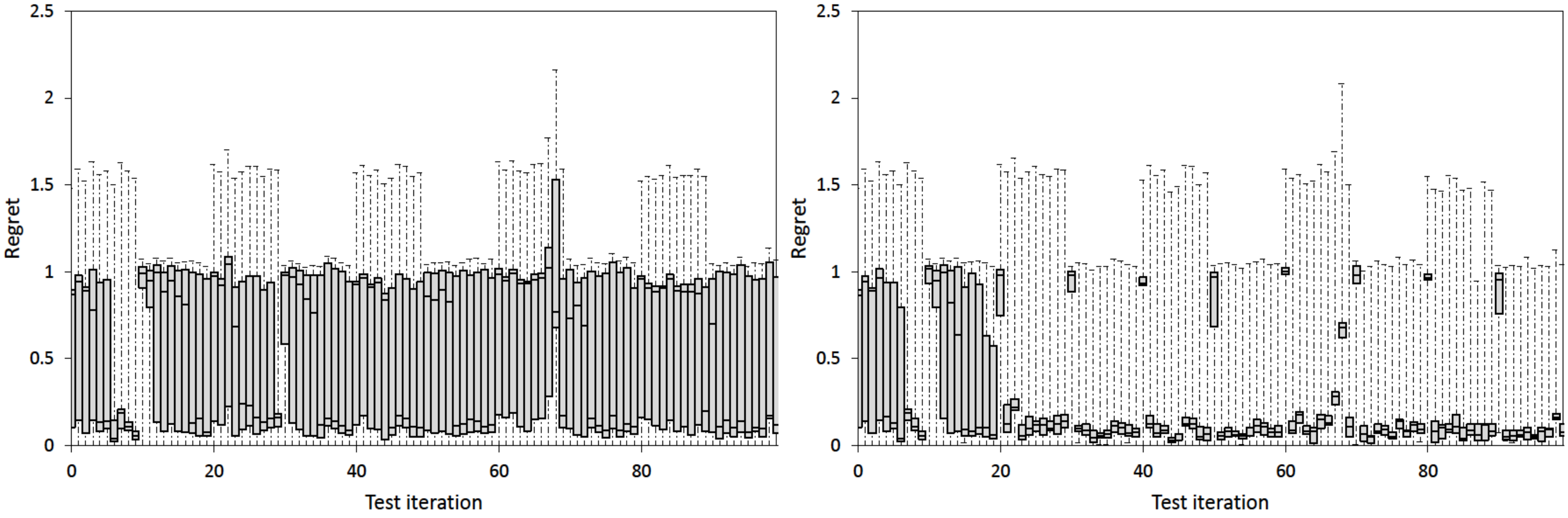


Figure 8 – Learning without (left) and with (right) categorization on a request pattern that changes every ten iterations.

Thoughts

- Will the adaptation be computationally expensive as number of components and metrics increase ? – Scalable?
- Impact on QoS during transition
- Ease of adding access patterns in model
- Overhead of providing various implementation for a component Vs Simple Knob Tuning
- Extra overhead of module loading for a large system

Questions ?