An Actor-Based Runtime Monitoring System for Web and Desktop Applications

Paul Lavery* & Takuo Watanabe
Department of Computer Science, Tokyo Institute of Technology
(*Currently with ContentSquare)

Jun. 27, 2017
IEEE/ACIS SNPD 2017, Kanazawa, Japan

About This Talk

- Introduces a library-based approach to runtime-monitoring for actor-based systems
- Two case studies to evaluate the proposal

- Talk outline
  - Runtime Verification/Monitoring
  - Motivation: GPL/Library-based Approach
  - Monitoring Module for Scala/Akka
  - Case Study
  - Conclusion

Runtime Monitoring/Verification

- "A computing system analysis and execution approach based on extracting information from a running system and using it to detect and possibly react to observed behaviors satisfying or violating certain properties"
    - From http://fsl.cs.illinois.edu/index.php/Runtime_Verification

- A kind of 'light-weight' formal methods
  - Bridging the gap of (static) verification and testing
    - RM/RV can deal only with finite execution traces
  - Properties are usually specified in a formal notation/DSL
    - ex. RE, Büchi Automata, LTL, PT-LTL, PT-DTL

PT-LTL

- Past-Time Linear Temporal Logic [Manna et al '92]
- Formula

$$F ::= true \mid false \mid p \mid \neg F \mid F \wedge F \mid F \vee F \mid F \rightarrow F \mid$$
$$\odot F \mid \diamond\!\!\cdot F \mid \boxdot F \mid F \,\mathcal{S}\, F$$

  - temporal operators: "previously", "sometime in the past", "always in the past", "since"
- Example [Sen et al '04]

$$\boxdot((action \wedge \odot\neg action) \rightarrow (\neg stop \,\mathcal{S}\, start))$$

  - "Whenever *action* starts to be true, it is the case that *start* was true at some point in the past and since then *stop* was never true"

PT-DTL

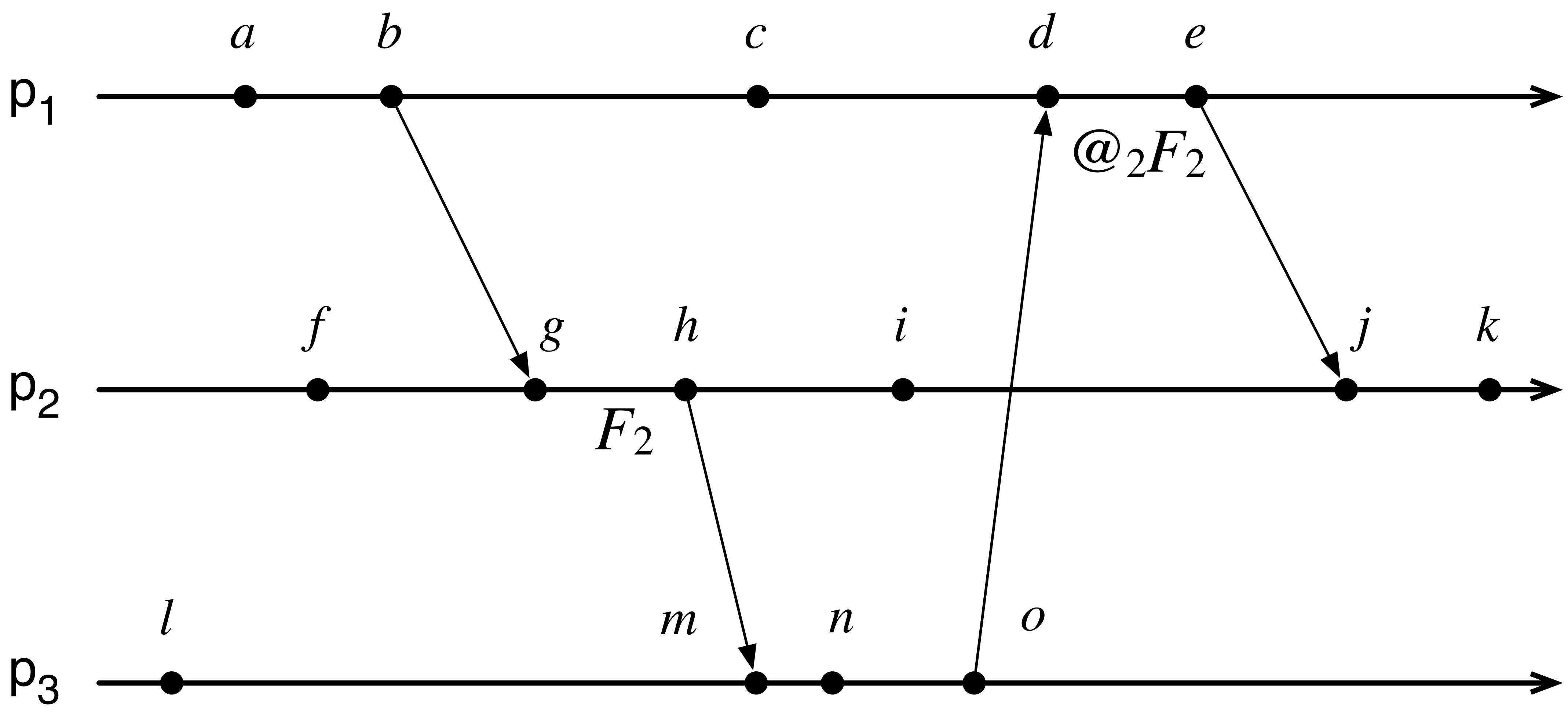
- Past-Time Distributed Temporal Logic [Sen et al '04]

$$
\begin{aligned}
F_i &::= \mathit{true} \mid \mathit{false} \mid P(\overrightarrow{\xi_i}) \mid \neg F_i \mid F_i \wedge F_i \mid F_i \vee F_i \mid F_i \rightarrow F_i \mid \\
&\qquad \odot F_i \mid \Diamond\!\!\!\cdot\, F_i \mid \boxdot F_i \mid F_i \,\mathcal{S}\, F_i \mid @_j F_j \\
\xi_i &::= c \mid v_i \mid f(\overrightarrow{\xi_i}) \mid @_j \xi_j \\
\overrightarrow{\xi_i} &::= (\xi_i, \ldots, \xi_i)
\end{aligned}
$$

- i-formulae / i-expressions
  - F_i, ξ_i : formula / expression local to process p_i
    - Subscript indicates that they refers to the local names
- Epistemic formulae / expressions
  - $@_j F_j, @_j \xi_j$: refers to the latest local knowledge of p_i about p_j

K. Sen, A. Vardhan, G. Agha and G. Roşu, "Efficient Decentralized Monitoring of Safety in Distributed Systems", ICSE 2004.

Epistemic Formulae/Expressions



- Examples
  - $@_2F_2$ in p_1 at d equals to F_2 in p_2 at h
  - $@_2\xi_2$ in p_1 at d equals to ξ_2 in p_2 at h

Classification of RV/RM

- Property Specification
  - General Purpose Languages vs. DSL/Formal Notations
    - DSL: External or Embedded
  - Imperative vs. Declarative
    - Imperative : GPL, Automata
    - Declarative: Temporal Logic
- Monitoring & Enforcement (Mitigation)
  - Modified Runtime vs. Unmodified Runtime
    - Modified : Kernel, VM, Language Runtime, Libraries
    - Unmodified : Code Modification/Instrumentation, Reflection
  - Synchronous vs. Asynchronous
  - Centralized vs. Decentralized

Our Previous Works on RV/RM

- Runtime Security Monitor for JVM
  - Property Specification
    - DSL based on Büchi Automata [Watanabe et al, 2003]
  - Enforcement Mechanism: Code Instrumentation
  - Application to Secure E-Mail System [Shibayama et al, 2003]

- Runtime Monitoring of Information-Flow Properties
  - Theoretical Foundation of Information-Flow Property Monitoring [Nagatou et al, 2005]
  - Application to Detecting Covert Channels [Nagatou et al, 2006]

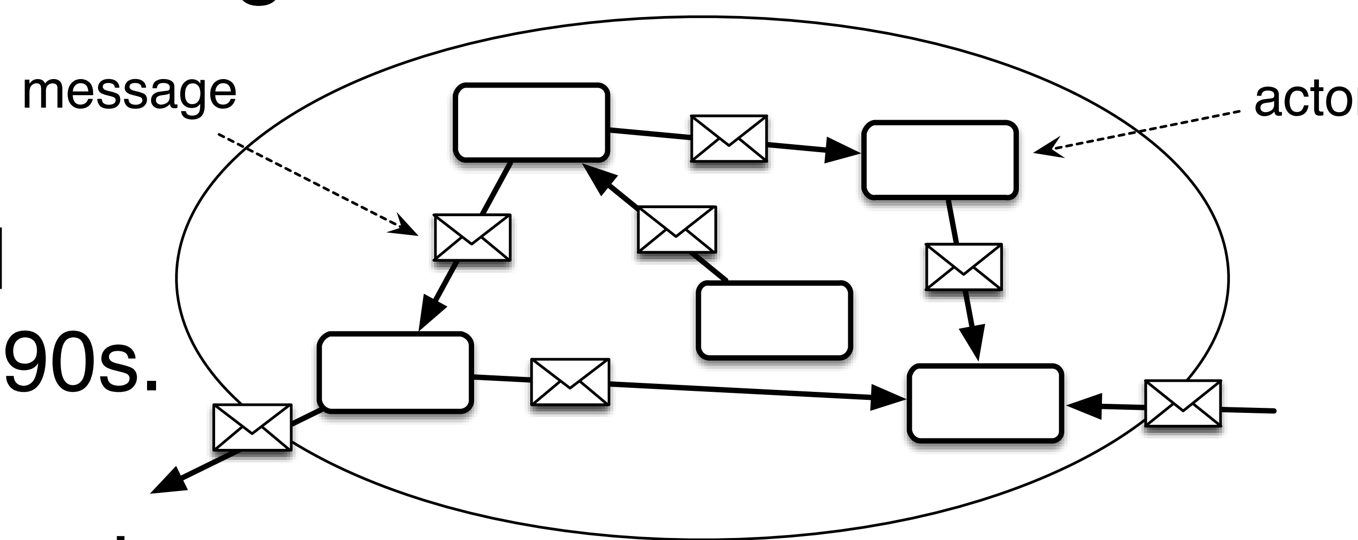
Watanabe, Yamada, Nagatou, "Towards a Specification Scheme for Context-Aware Security Policies for Networked Appliances", IEEE STFES 2003.
Shibayama et al, "AnZenMail: A Secure and Certified E-mail System", Software Security: Theories and Systems, LNCS 2690, 2003.
Nagatou, Watanabe, "Execution Monitoring and Information Flow Properties", IEEE DSN 2005.
Nagatou, Watanabe, "Run Time Detection of Covert Channels", IEEE ARES 2006.

About this Work

- Goal
  - Provide an easier access to runtime monitoring by presenting an easy-to-use, scalable monitoring framework that provides developers with a way to dynamically verify some important specifications and, in case they are violated, with a mitigation mechanism
- Proposed Solution
  - Target: Actor-based Applications written in Scala/Akka
  - Property Specification: Scala
    - Monitor/Worker/Listener Classes
    - Checking code embedded in the target applications source
  - Monitoring and Enforcement
    - Scala library that receives monitoring information as asynchronous messages

The Actor Model

- A concurrent computation model based on asynchronous message passing
  - Originally invented by C. Hewitt in 1970s and developed by G. Agha and other researchers in 1980-90s.
  - Basis of many languages: Erlang, Scala (Akka), Pony, etc.



- A system is modeled as a collection of actors that communicate with each other only via messages.
  - "Shared Nothing": no shared states, no global clock
  - No channels (mail address based)
  - Dynamic Topology (mail addresses are 1st class)

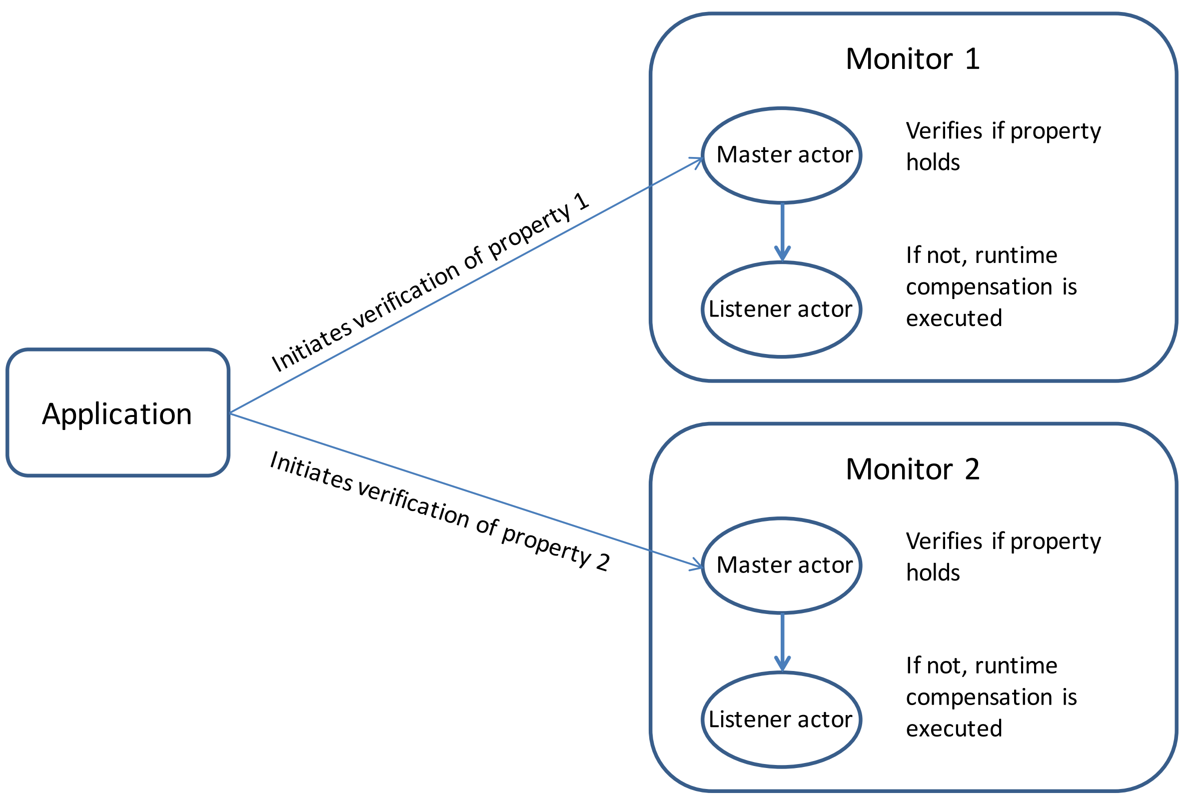
Monitoring Modules

- The desired properties of an application are specified as a collection of monitoring modules.

- At the runtime, modules check those properties and executes some compensation (mitigation) tasks if they are violated
  - one monitor per property
  - asynchronous monitoring (non-blocking)

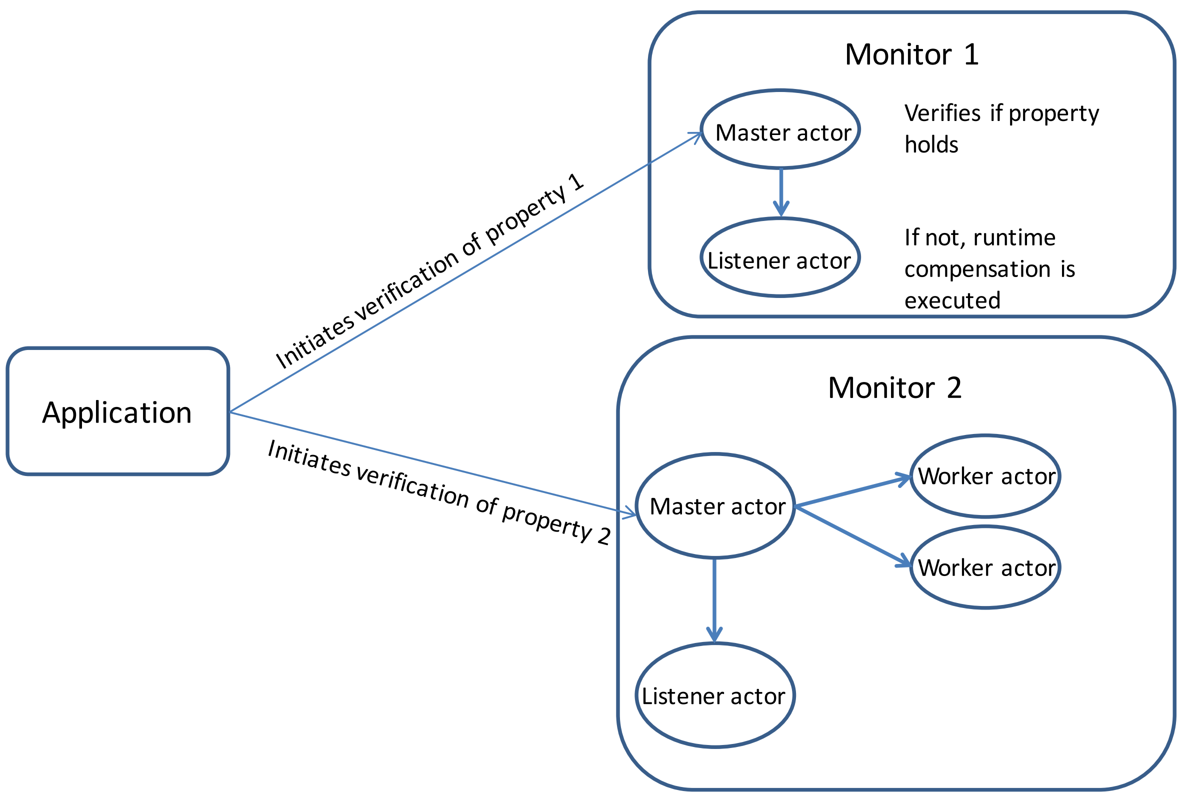
- Modules are written as Akka actors

Monitoring Architecture (1)



- Master actor
  - Checks whether the specified property holds in current system state by executing the property method.
  - If it does not, sends the arguments for mitigation to the listener actor.
- Listener Actor
  - Performs compensation (mitigation) tasks

Monitoring Architecture (2)



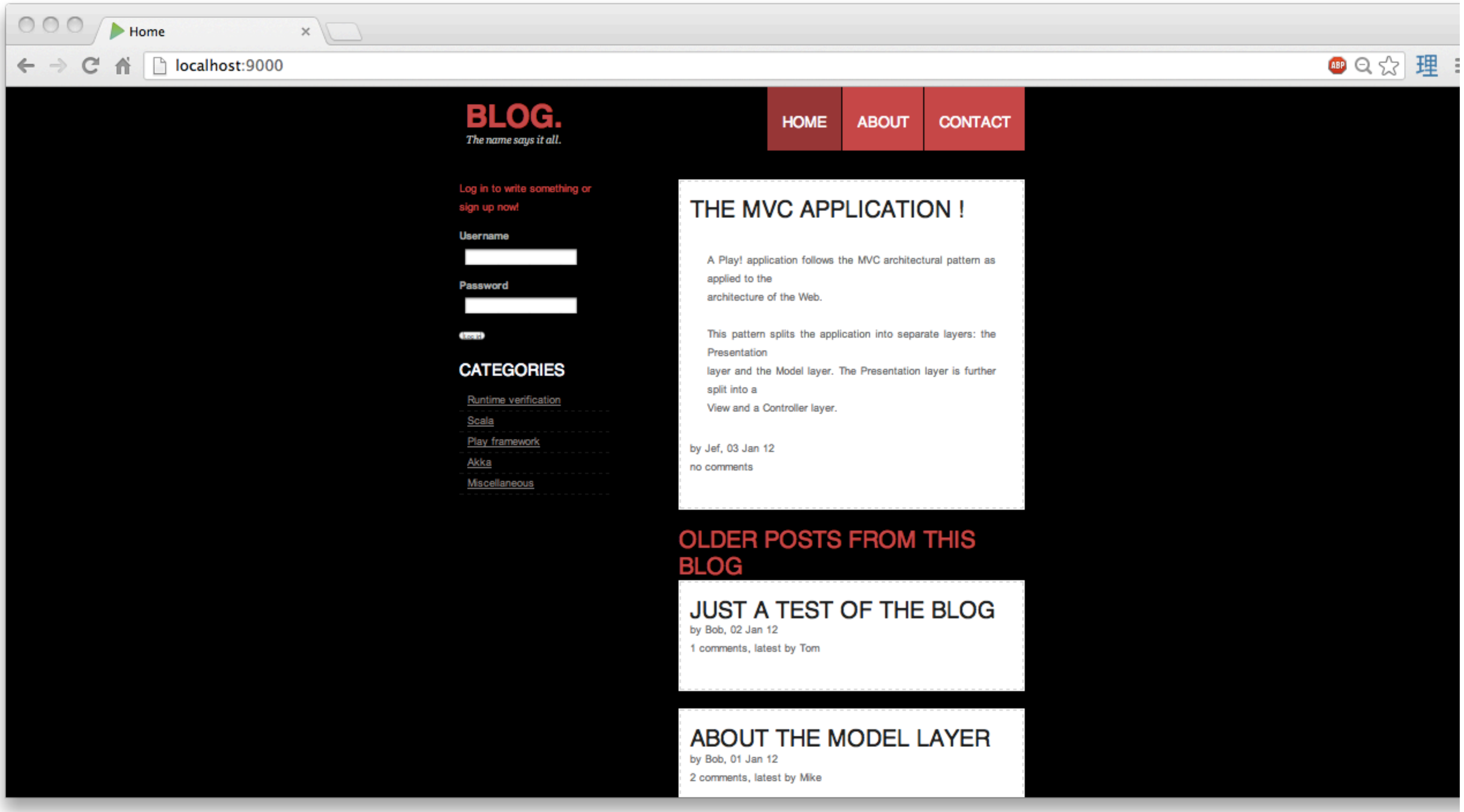
- (Extended) Master Actors
  - Creates a pool of workers
  - Distributes the work to the workers in a round robin fashion through a router
  - Receives the result message from the workers and forwards the content to the listener
- Worker Actors
  - Executes property method and if the property does not hold sends the arguments for mitigation to the master actor
  - Able to check properties in parallel independently

How to Integrate Monitors

- Define Monitors/Workers/Listeners
  - Monitors
    - property: S => (Boolean, T)
      - If false the property does not hold and mitigate is executed with T-typed element as argument
      - If true the property holds and nothing more is done
    - mitigate: T => Unit

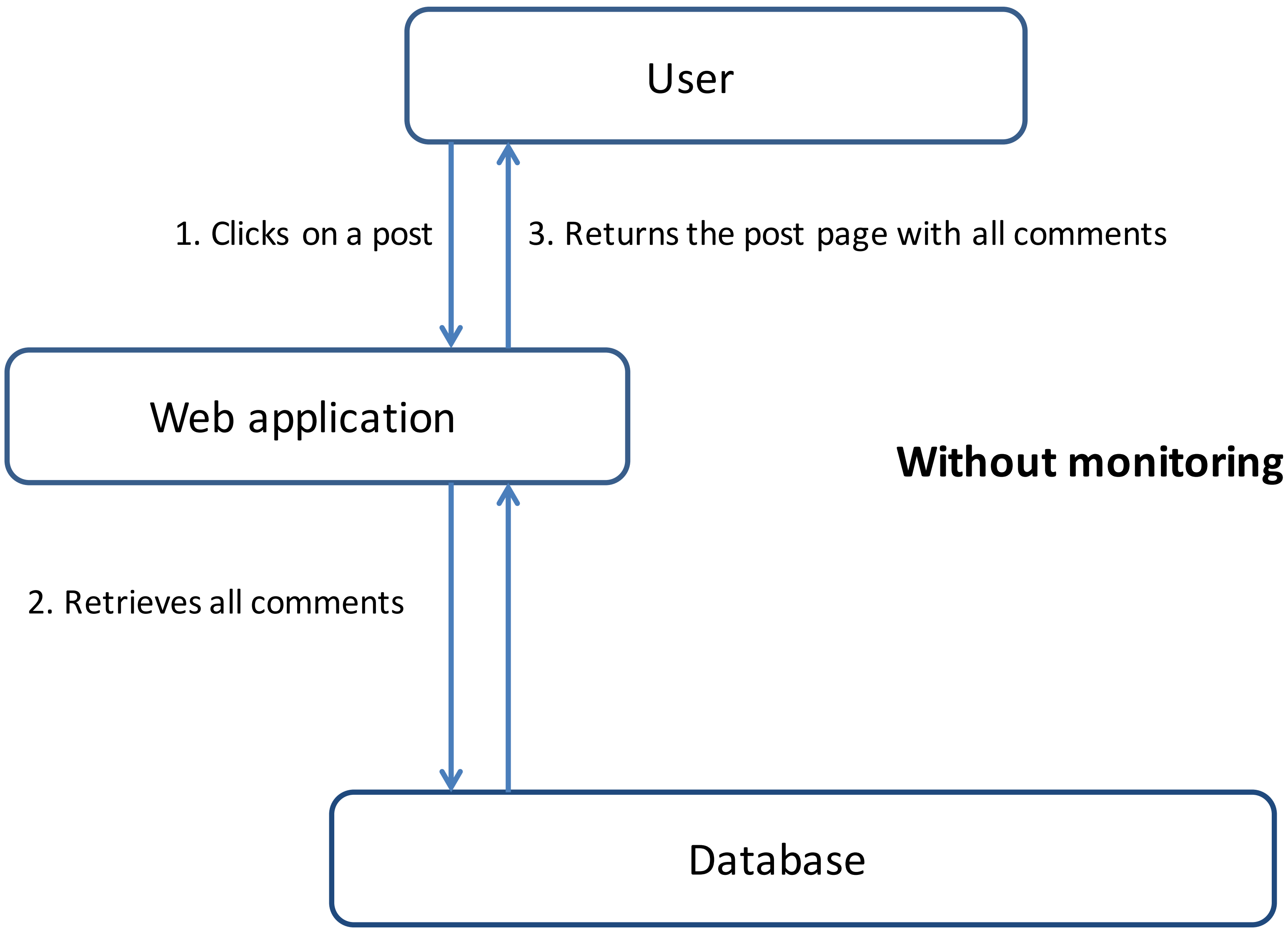
- Modify the Target Application Code
  - Insert transmission sentences of Check messages to application actors where the properties should be verified

Case Study (1): Blog Application



- Blog application written using Play! framework
- Properties to be monitored
  - No Spam Comments/Posts
  - No Inactive Users

Simplified Blog Architecture

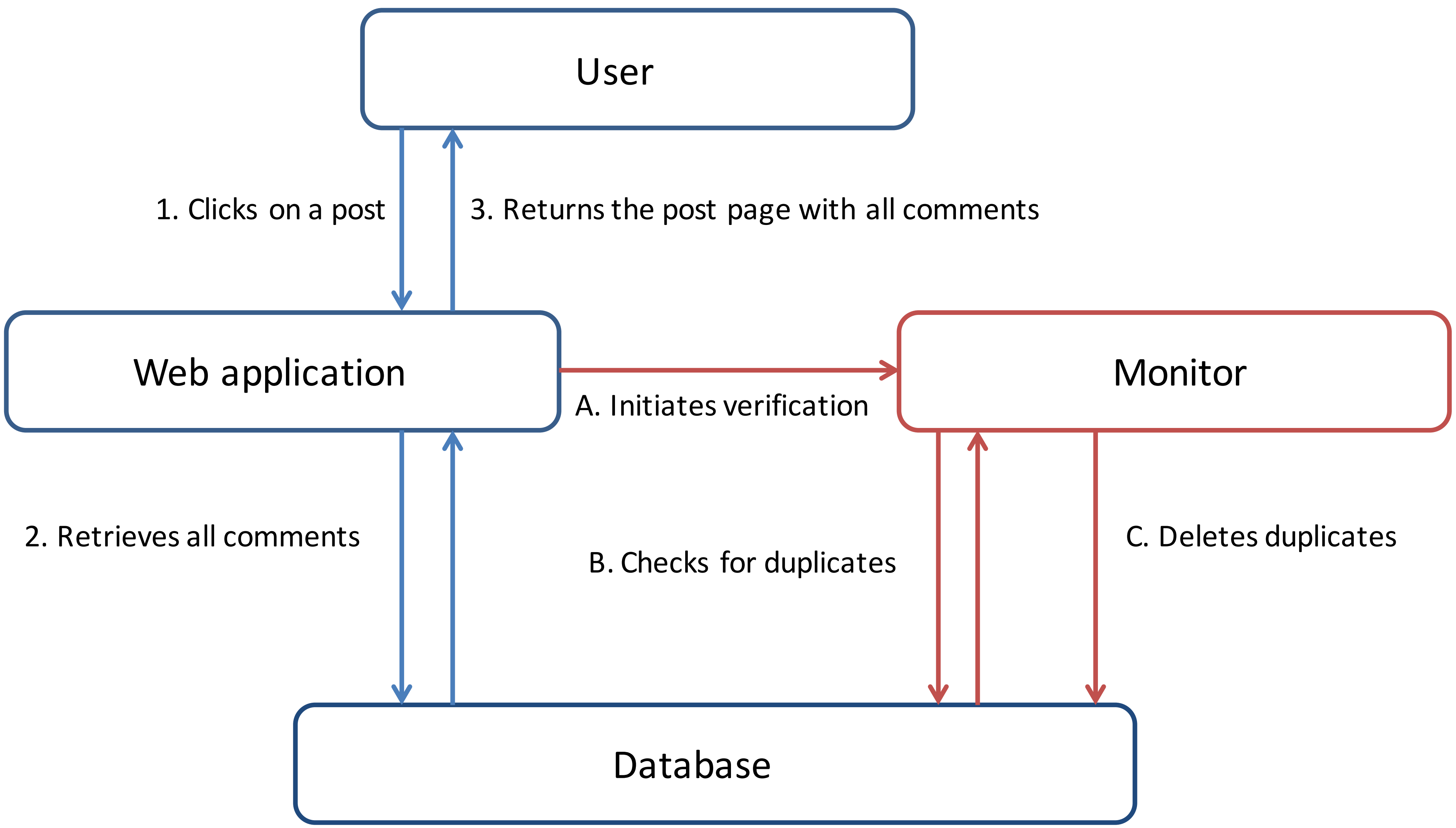


Without monitoring

Properties to be monitored

- No Spam Comments/Posts
  - Repetition of same comments/posts should be deleted
  - Short-term property checked asynchronously by a monitoring actor in the same host of blog engine
- No Inactive Users
  - Users who have not posted any blog articles for long time (> 1 year) should be deleted
  - Long-term property checked asynchronously by a monitoring actor in a separate host

Checking "No Spam Comments" Property



Monitor for "No Spam Comments"

```
def checkPropertyComment(args: Unit): (Boolean, List[Int]) = {
        val result =
            Comment.mostRecent().groupBy(_.content).toList.map(_._2.drop(1)).flatten.map
            { commentt => commentt.id.get.toInt }
        if (!result.isEmpty)
                return (false, result)
        else
                return (true, List(0))
}
def mitigationComment(res: List[Int]) = {
        for (commentId <- res) Comment.delete(commentId)
}
val system = ActorSystem("mySystem")
val monitorComment = new Monitor[Unit, List[Int]](system, "monitorComment",
    checkPropertyComment, mitigationComment, 4)
```

Case Study (2): Shooting Game

- 2D side scrolling game in the style of old arcade games like R-Type
- Spaceship shoots missiles to destroy all the enemies coming from the right of the screen
- Written using Scala Swing GUI library
- Properties to be monitored
  - No cheats!

Case Study (2): Shooting Game

- The application runs on top of usual JVM. So the player can cheat by modifying game-state variables (e.g., by using JDI)

- The monitor checks that (a) the number of spaceship lives and (b) the damage of shields are consistent with game execution

Monitor for "Shield Consistency"

```scala
def checkShieldProperty(args: Unit): (Boolean, Unit) = {
        var result = false
        if (!Item.itemsFound.filter(i => i._1.isInstanceOf[Shield]).isEmpty) {
                var w = Item.itemsFound.filter(i => i._1.isInstanceOf[Shield]).last
                result = w._2 > ( System.nanoTime() -
                    (w._1.activeDuration+30)*30*scala.math.pow(10, 6) ) }
        return (result, Unit)
}
def shieldMitigation(arg: Unit): Unit = {
        //Logging
        println("a cheat shield was being used")
        //Active mitigation
        Spaceship.shield = false
        Spaceship.numberOfLives -= 1
}
val shieldMonitor = new Monitor[Unit, Unit](system, "shieldMonitor",
    checkShieldProperty, shieldMitigation)
```

Pros/Cons of Proposed System

- Pros
  - Simple, Easy to Use
    - No need to learn dedicated DSL or logical notations
    - Can be integrated as a normal Scala library
      - No JVM modification, No code transformation
  - Can cover most of internal/distributed properties
- Cons
  - No correctness guarantee
    - The monitoring module should be programmed to represent the properties to be monitored
  - No support for complex properties

Microbenchmarking

- Calculated over 1200 computations of factorials of all numbers between 1 and 100 and monitoring the correctness of results (no mitigation)
- Results
  - Average runtime overhead of 15%
  - Only 8% of the cases causing more than a 5% overhead
  - Higher variance with the monitored program

Data considered	Mean	Variance
Test program outputs	623.3 ms	1.4
Monitored program outputs	715.3 ms	2.7

Future Work

- The mechanism to support building correct properties
  - type based property description
    - The type-safety enforced by the module could be improved upon for further convenience for the programmer
  - library of common properties
- Avoid bypassing
  - Inserting checking sentences just before runtime
    - e.g., Dynamic AOP, Reflection
  - Using "Software Diversity" mechanism
- Justification
  - Theoretical side
  - Practical side

Conclusion

- We developed an efficient and simple runtime monitoring module for Scala/Akka applications based on the Actor model
  - Properties are written as Scala code
  - Asynchronous/Distributed Monitoring and Mitigation

- Case Study
  - Blog using Play! Framework
    - no spam (no duplicate posts + comments)
    - no inactive users
  - A Desktop Shooting Game
    - no cheats

- Good results on the micro-benchmark