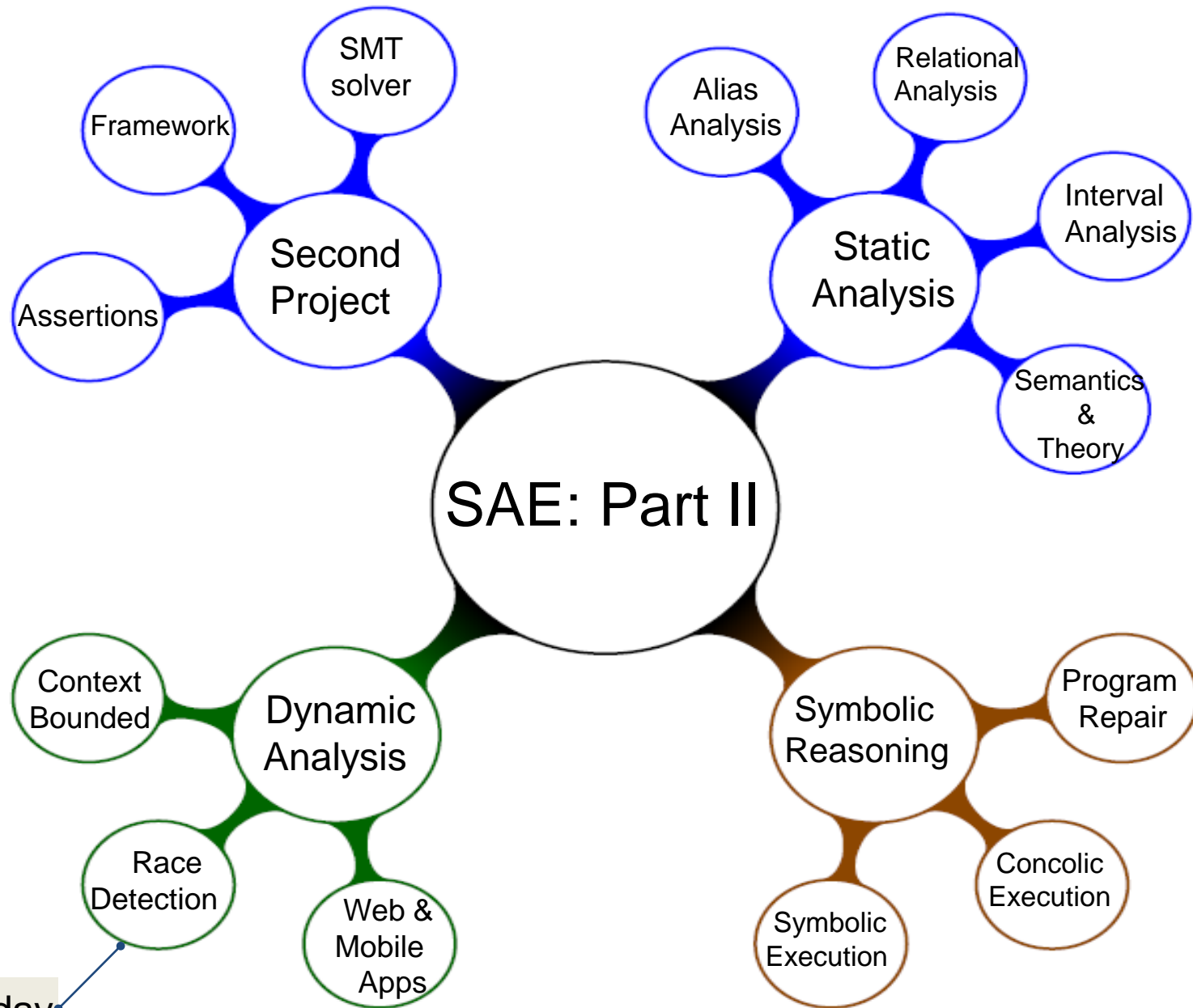


Software Architecture and Engineering: Part II

ETH Zurich, Spring 2016

Prof. Martin Vechev

<http://www.srl.inf.ethz.ch/>



Today

Dynamic Race Detection

- A popular kind of **dynamic analysis**
 - The analysis is an under-approximation: it considers a subset of the program behaviors
- **Highly effective** for finding concurrency bugs
- Many **different variants**
 - Interesting trade-off between **asymptotic complexity** and **precision** of the analysis

Today

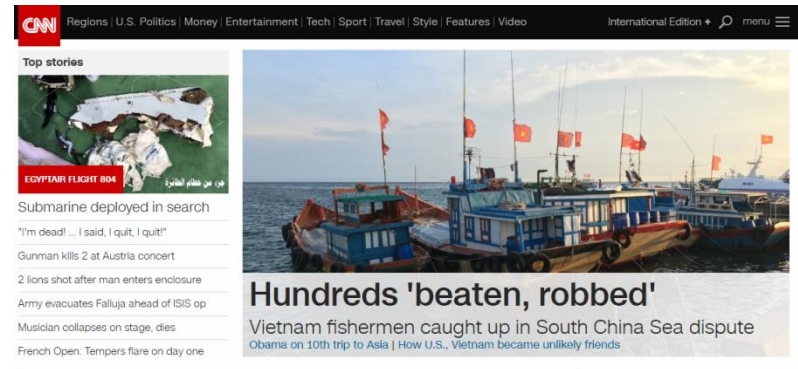
We will illustrate the **key concepts** of race detection on a rich application domain that is quite prevalent today, namely **event-driven** applications such as Web pages and Android

All concepts we study today apply to other settings: e.g. regular concurrent Java programs.

Motivation: Event-Driven Applications



~ 1 billion smartphones



~ 640 million web pages


Reacts to events: user clicks, arrival of network requests

Event-Driven Applications


Wanted: fast response time

Highly Asynchronous,
Complex control flow

Looks Like This

Regions | U.S. Politics | Money | Entertainment | Tech | Sport | Travel | Style | Features | VideoInternational Edition + 🔍 menu ☰

Top stories



EGYPTAIR FLIGHT 804 جزء من حطام الطائرة

Submarine deployed in search

"I'm dead! ... I said, I quit, I quit!"


Gunman kills 2 at Austria concert

2 lions shot after man enters enclosure

Army evacuates Falluja ahead of ISIS op

Musician collapses on stage, dies

French Open: Tempers flare on day one



Hundreds 'beaten, robbed'

Vietnam fishermen caught up in South China Sea dispute

[Obama on 10th trip to Asia](#) | [How U.S., Vietnam became unlikely friends](#)

This is what Runs

[illegible]

Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



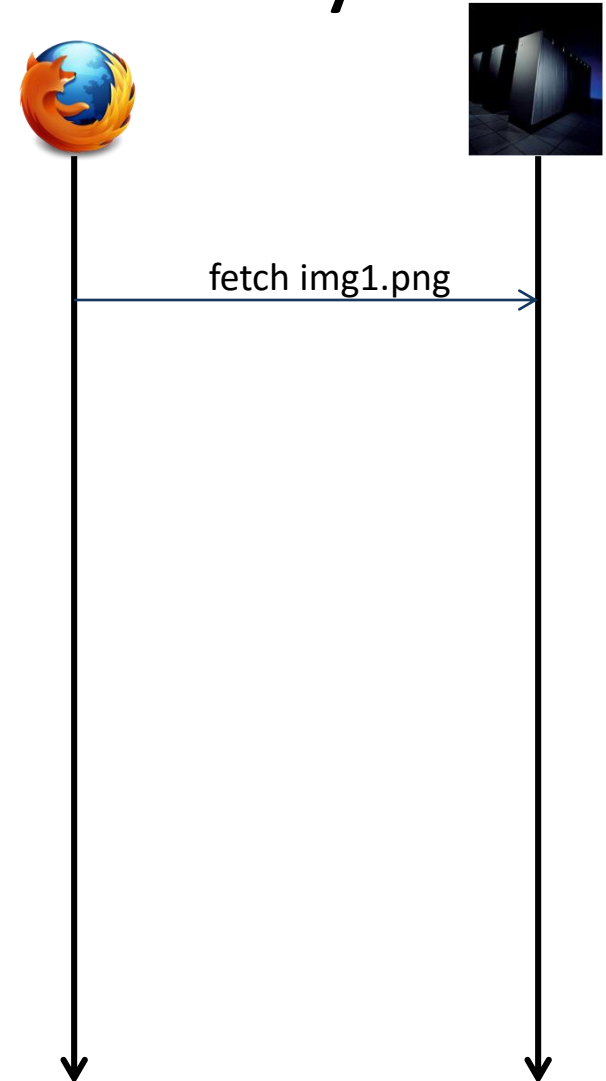
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



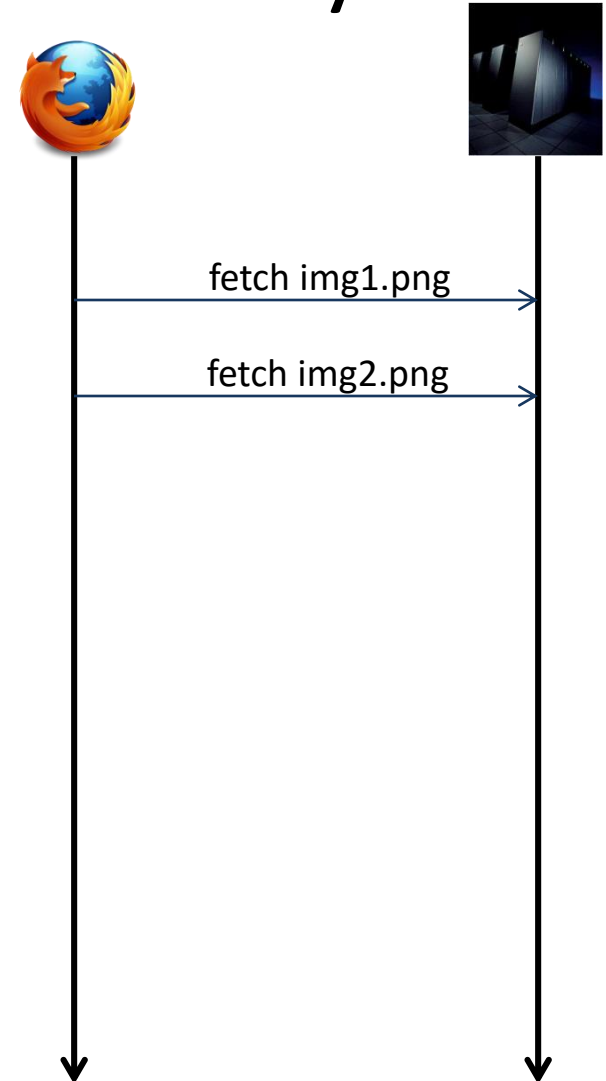
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



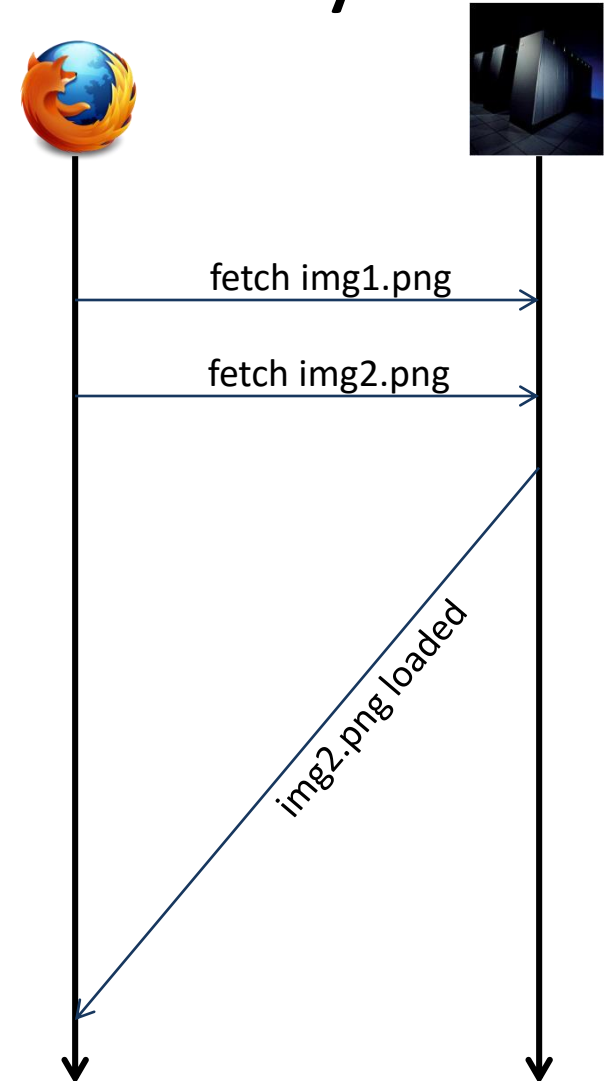
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



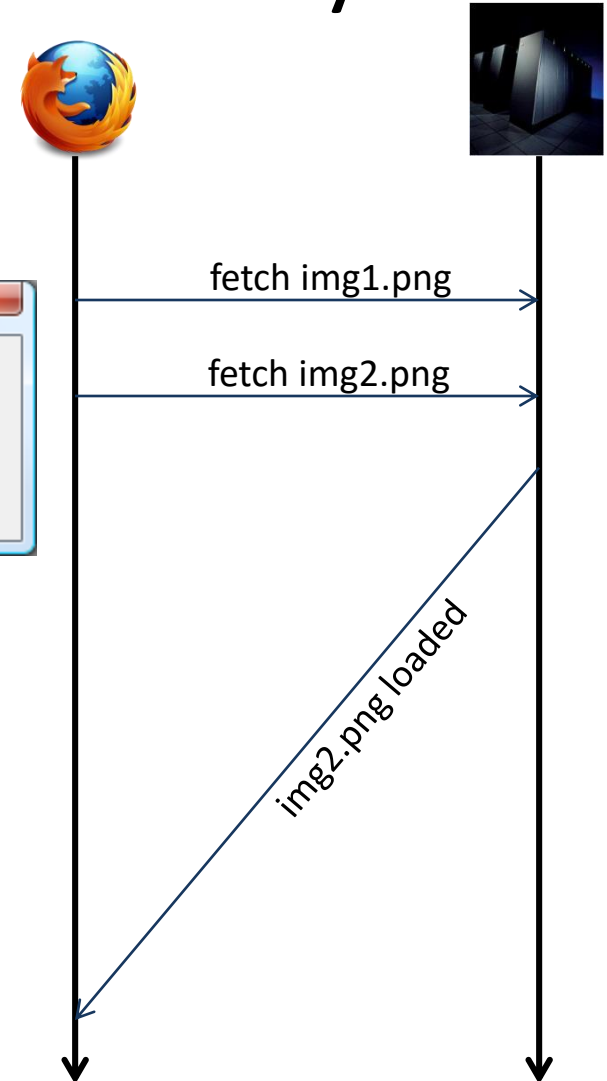
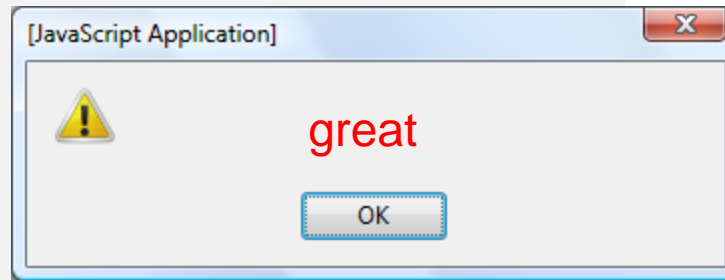
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



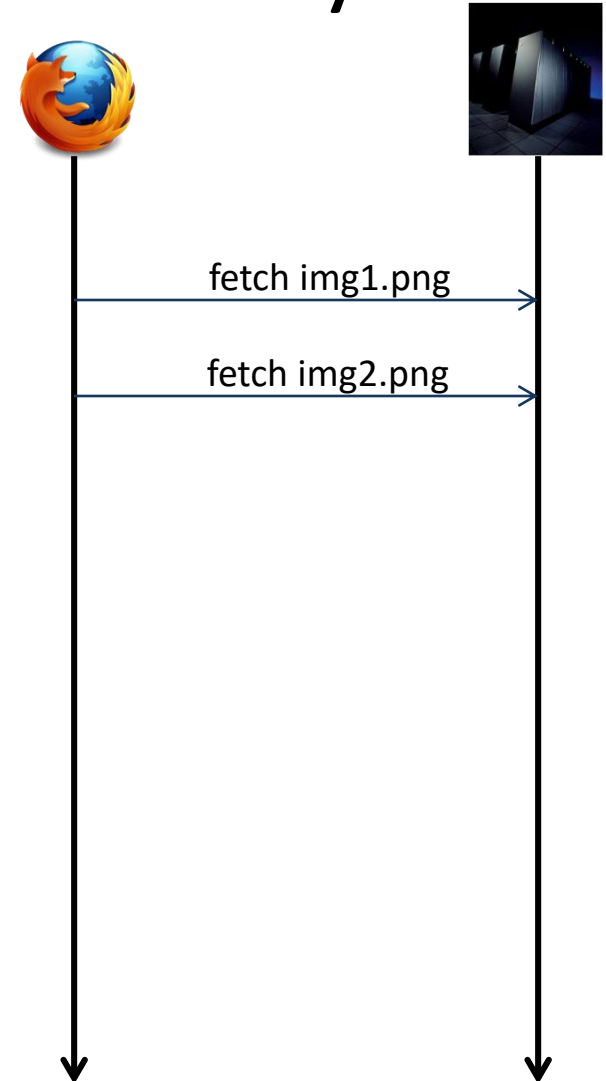
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



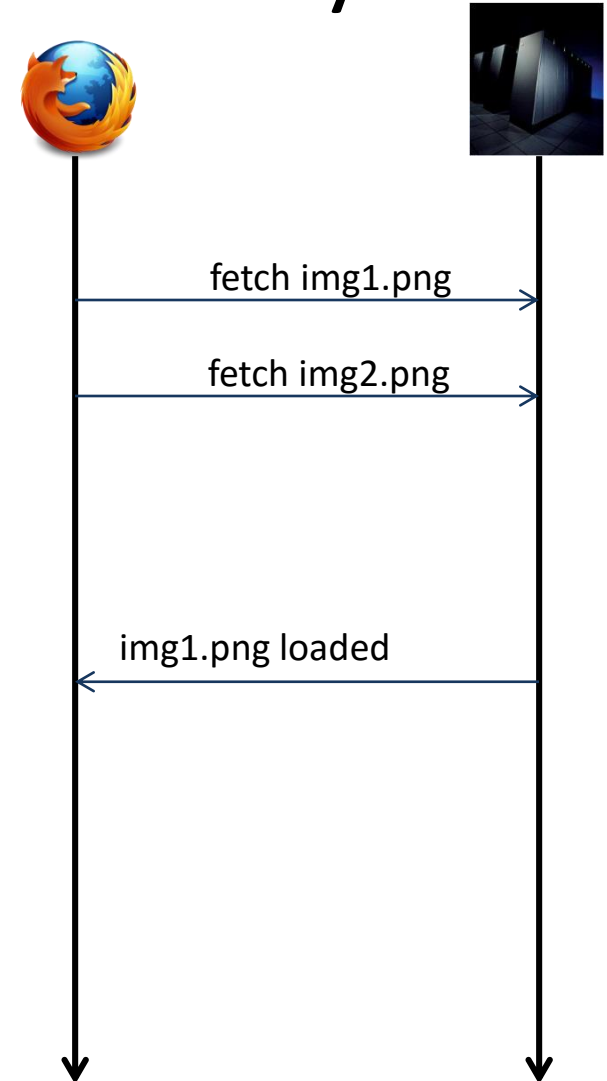
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



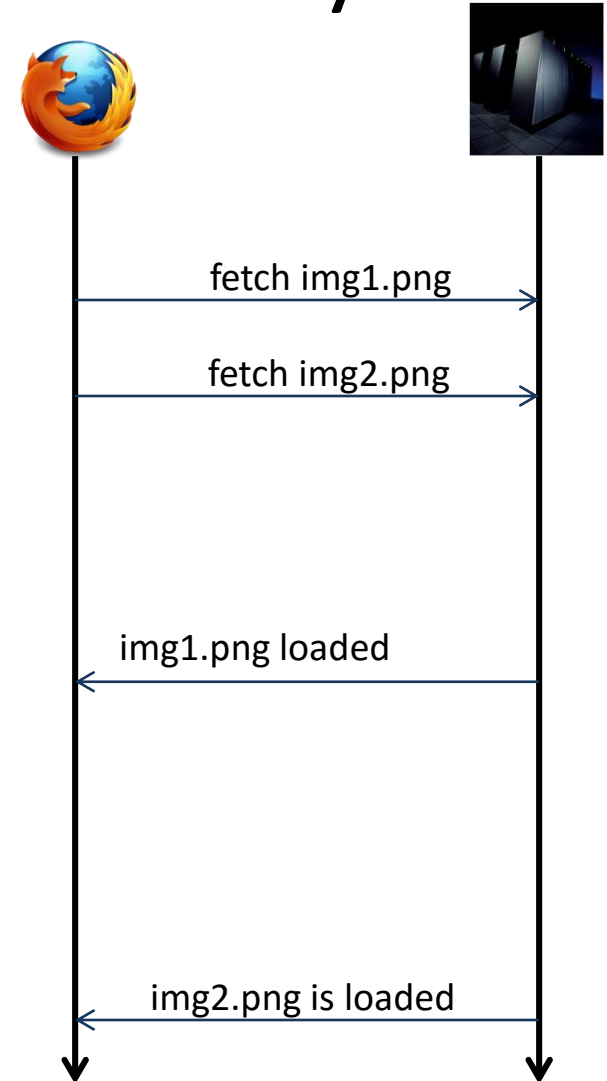
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



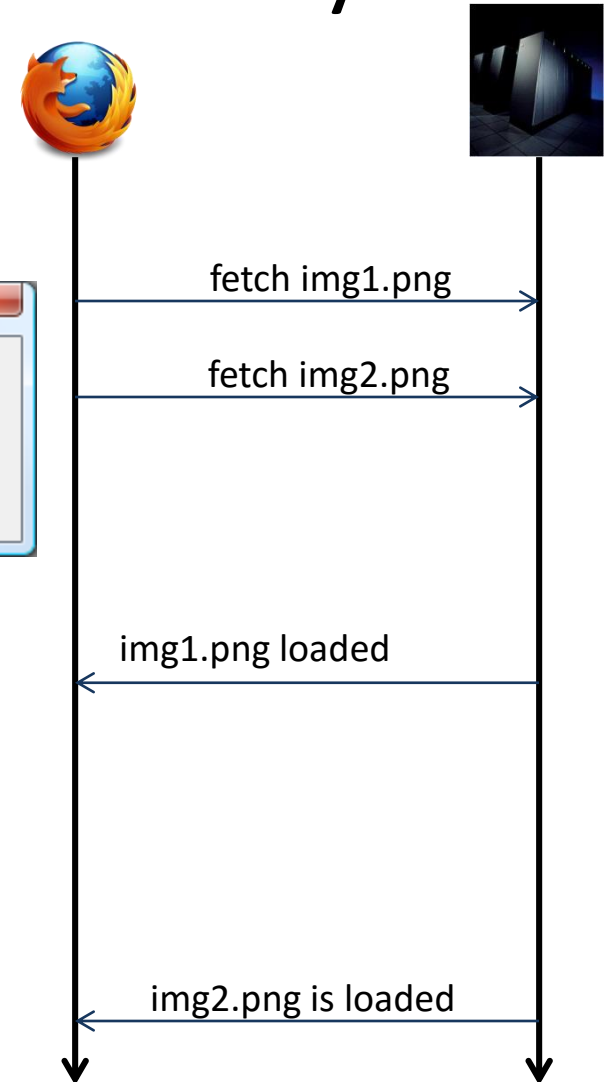
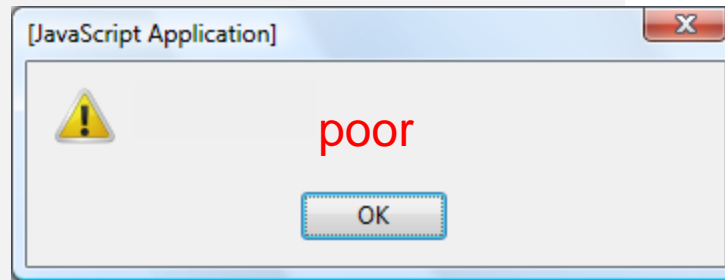
Non-determinism: network latency

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>




</body>
</html>
```



What do we learn from these?

Asynchrony + Shared Memory



Non-Determinism



Unwanted Behavior

What do we learn from these?

Asynchrony + Shared Memory



Non-Determinism



Unwanted Behavior

Lets phrase the problem as **data race detection**

What is a Data Race ?

What is a Data Race ?

Semantically, a data race occurs when we have a reachable program state where:

- we have two outgoing transitions by two different threads
- the two threads access the **same memory location**
- one of the accesses is a **write**

Examples

Data Race on X

Thread T_1 :	Thread T_2 :
fork T_2 $X = 1$	$X = 2$

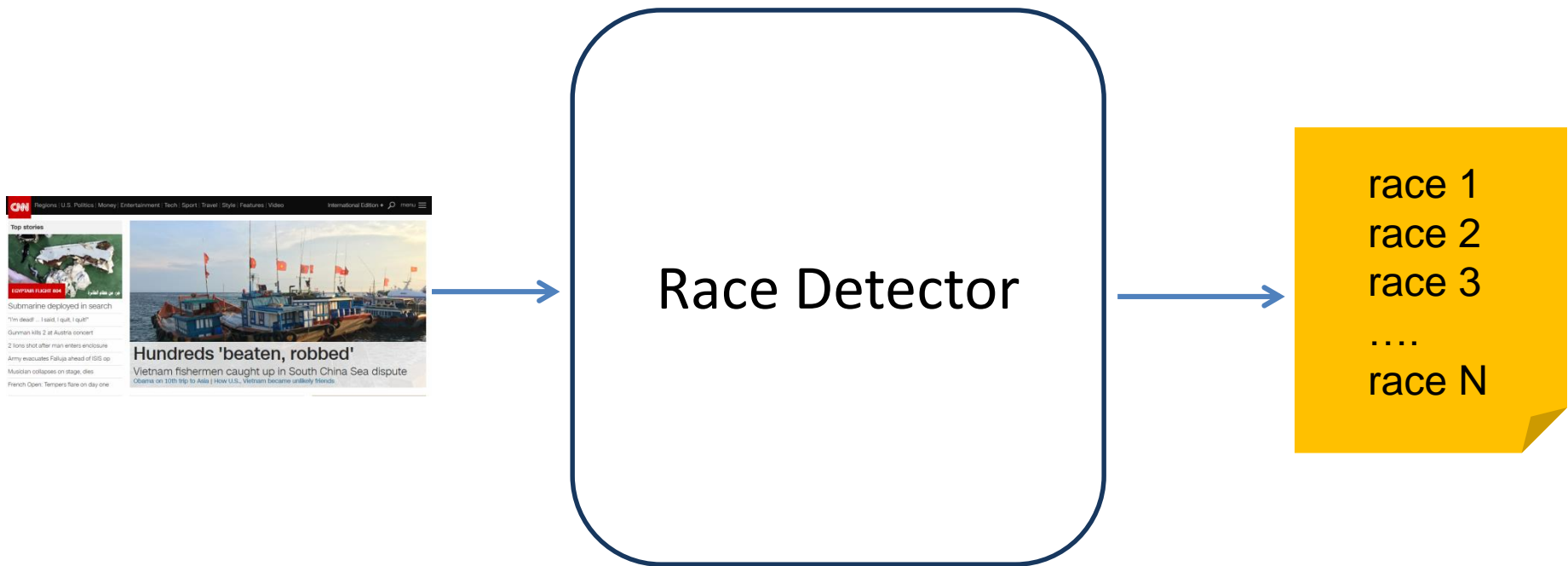
The program **has a reachable state** where both $X = 1$ and $X = 2$ are enabled

Program has No Data Races

Thread T_1 :	Thread T_2 :
$X = 1$ fork T_2	$X = 2$

The program **does not have a reachable** state where both $X = 1$ and $X = 2$ are enabled

Wanted



Naïve Algorithm

The definition of a data race suggests a naïve algorithm which finds **all races of a program** given some input states. The algorithm simply enumerates all reachable states of the concurrent program from the initial input states and checks the definition on each such reachable state.

Naïve Algorithm

The definition of a data race suggests a naïve algorithm which finds **all races of a program** given some input states. The algorithm simply enumerates all reachable states of the concurrent program from the initial input states and checks the definition on each such reachable state.

Does Not Scale to Real-World Programs

In Practice

In practice, algorithms aim to scale to large programs by being more efficient and not keeping program states around. To accomplish that, they **weaken their guarantees**.

We will see the guarantees they provide a little later, but at this point it is sufficient to mention that a typical guarantee is that the **first race the algorithm reports is a real race**, but any subsequent reported races after the first race are not guaranteed to exist, that is, they may be **false positives**, a major issue to deal with for any modern analyzer.

False positives exist because of **user-defined synchronization**.

Example of a False Positive Race (on variable X)

Initially: $X = Y = 0$

Thread T_1 :		Thread T_2 :
<code>while(Y == 0);</code>		<code>X = 0</code>
<code>X = 1</code>		<code>Y = 1</code>

A state of the art race detector may report a **race on X and Y**

Modern Dynamic Race Detection: 5 Steps

Step 1: Define Memory locations (on which races can happen)

Usually easy but there can be issues (framework vs. user-code)

Step 2: Define Happens-Before Model (how operations are ordered)

Can be tricky to get right due to subtleties of concurrency

Step 3: Come up with an Algorithm to detect races

Hard to get good asymptotic complexity + correctness

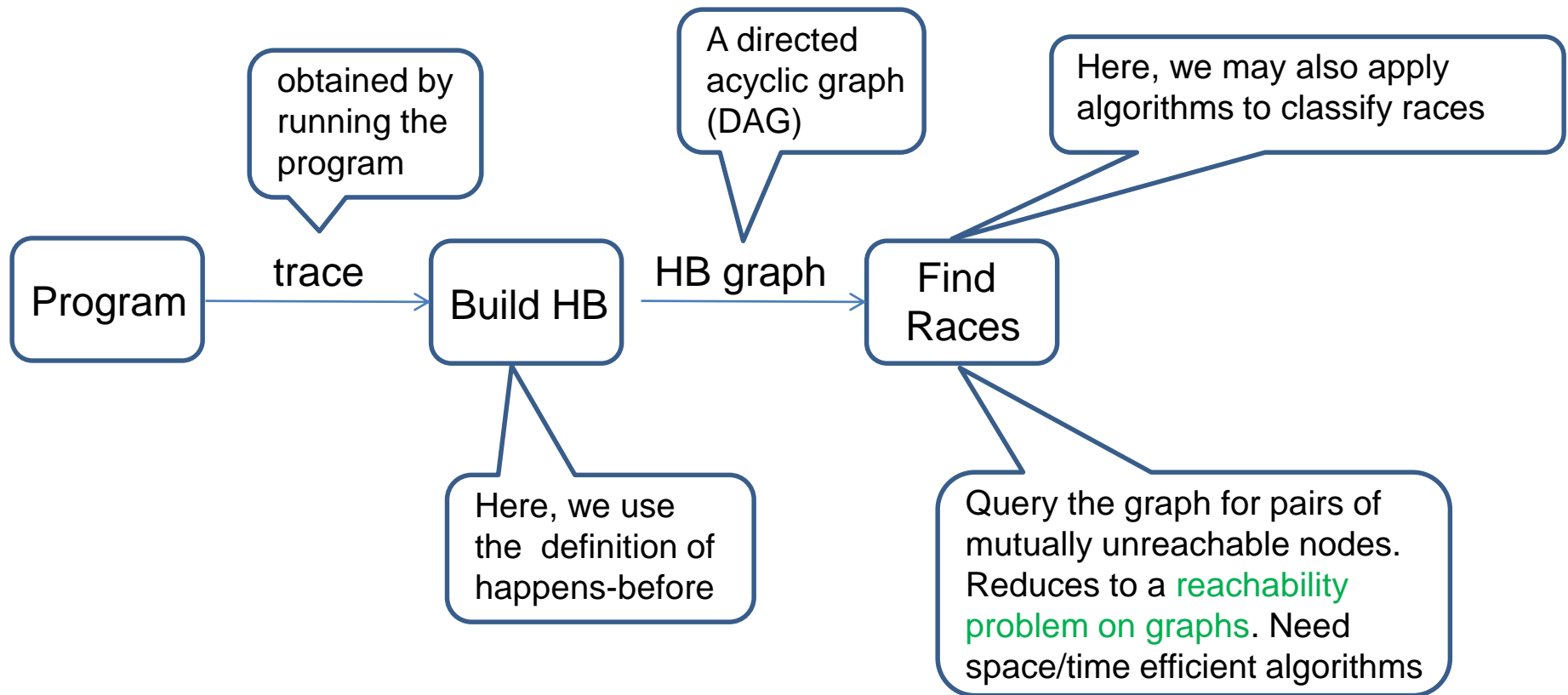
Step 4: Come up with techniques (algorithm, filters) to remove harmless races

Needs to answer what harmless means

Step 5: Implement Algorithm and Evaluate

Important to have low instrumentation overhead

Dynamic Race Detection: Flow



(some of these boxes will become clear later in the slides)

Let us now discuss these 5 steps in our example domain: event-driven applications

These 5 steps need to be taken for **any other domain**

Step 1:

Memory
Locations

- "Normal", C-like, memory locations for JavaScript variables
- Functions are treated like "normal" locations
- HTML DOM elements
- Event, event-target and event-handler tuple

Memory Locations: Example

```
<html>
<head></head>
<body>

<script>
var SAE = "great";
</script>





</body>
</html>
```

Step 2:

Happens-
Before
Model

... is a partial order (A, \preceq)

Step 2:

Happens-
Before
Model

... is a partial order (A, \preceq)

First, define the contents of A , i.e. atomic **action**

- E.g.: parsing a single HTML element, executing a script, processing an event handler

Step 2:

Happens-
Before
Model

... is a partial order (A, \preceq)

First, define the contents of A , i.e. atomic **action**

- E.g.: parsing a single HTML element, executing a script, processing an event handler

Then, define \preceq , i.e. how to **order** actions

- E.g.: parsing of HTML elements of the web page is ordered

Happens-Before: Example

```
<html>  
<head></head>  
<body>
```

```
<script>  
var SAE = "great";  
</script>
```

```

```

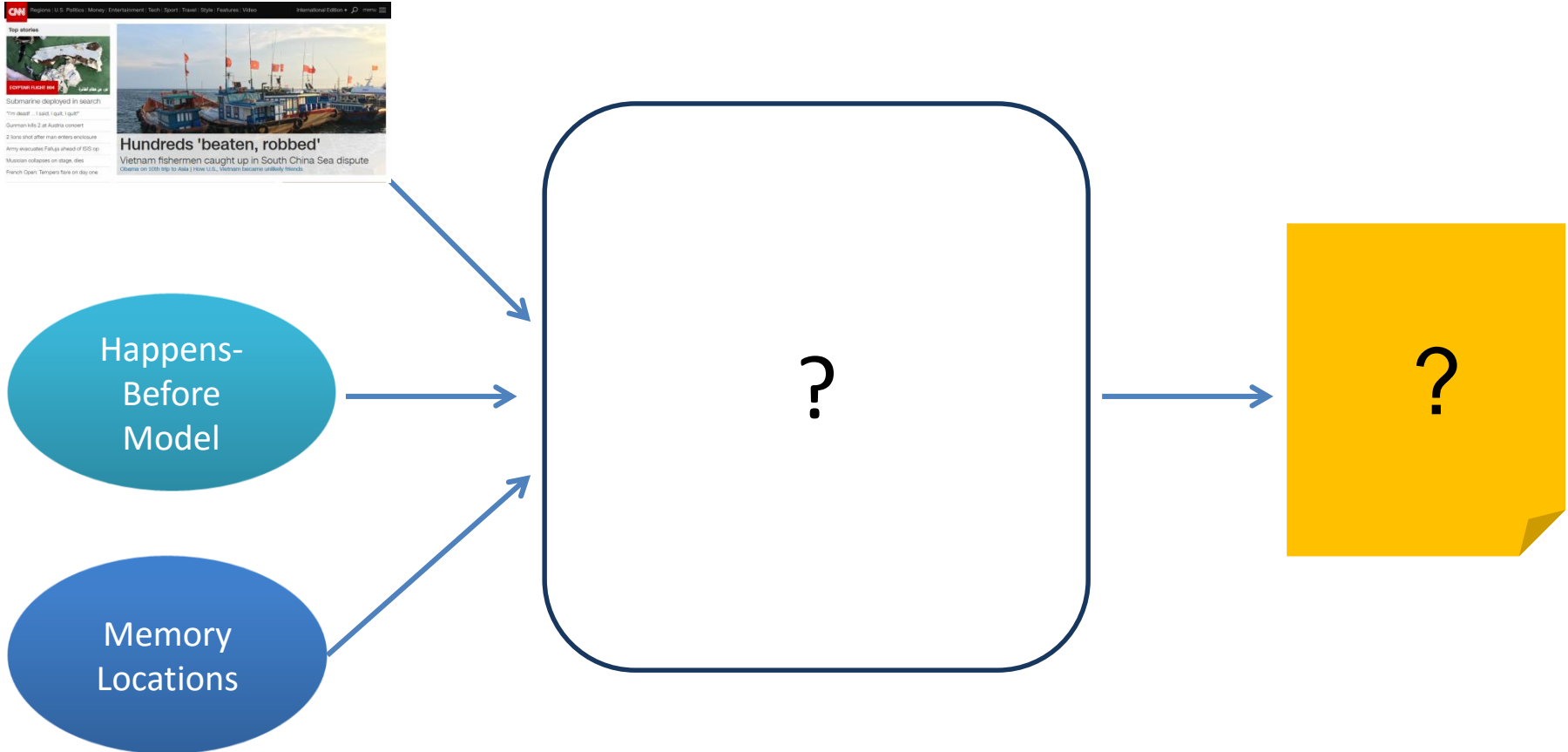
```

```

a data race on **SAE**

```
</body>  
</html>
```


Steps 3 and 4 : Define Race Detection Algorithm



Dynamic Race Detection: Theorems (that an analyzer should ensure)

No false negatives: if the Analysis reports **no races** on an execution, then the execution **must not contain a race**

No false positives: if the Analysis reports **a race** for a given execution then the execution **for sure contains a race**

Two Challenges Affecting Steps 3 and 4

Synchronization done with read/writes

quickly leads to **thousands of false races**



Massive number of event handlers

quickly causes **space blow-up** in analysis data structures



False Positives: Example

```
<html><body>

<script>
  var init = false, y = null;
  function f() {
    if (init)
      alert(y.g);
    else
      alert("not ready");
  }
</script>

<input type="button" id="b1"
  onclick="javascript:f()" ">

<script>
  y = { g:42 };
  init = true;
</script>

</body></html>
```

- 3 variables with races:
init
y
y.g
- some races are synchronization:
init
- reports **false races** on variables:
y
y.g

Wanted: “guaranteed” races

```
<html><body>

<script>
  var init = false, y = null;
  function f() {
    if (init)
      alert(y.g);
    else
      alert("not ready");
  }
</script>

<input type="button" id="b1"
  onclick="javascript:f()">

<script>
  y = { g:42 };
  init = true;
</script>

</body></html>
```

Intuition: identify races that are
guaranteed to exist.

We report races on variable
init

But not on:

y
y.g

Because races on **y** and **y.g** are
covered by the race on **init**

Two Challenges Affecting Steps 3 and 4

Synchronization with read/writes

race coverage **eliminates false races**



Massive number of event handlers

quickly causes **space blow-up** in analysis data structures



Computing Races

A race detector should compute races. The basic query is whether two operations a and b are **ordered**:

$$a \preceq b$$

Observation: represent \preceq (the happens-before of an execution trace) as a **directed acyclic graph** and perform graph connectivity queries to answer $a \preceq b$

Report a race if a and b are not reachable from one another, they teach the same memory location and one is a write.

Example \preceq built from a trace

Lets take the trace: ABCDE.

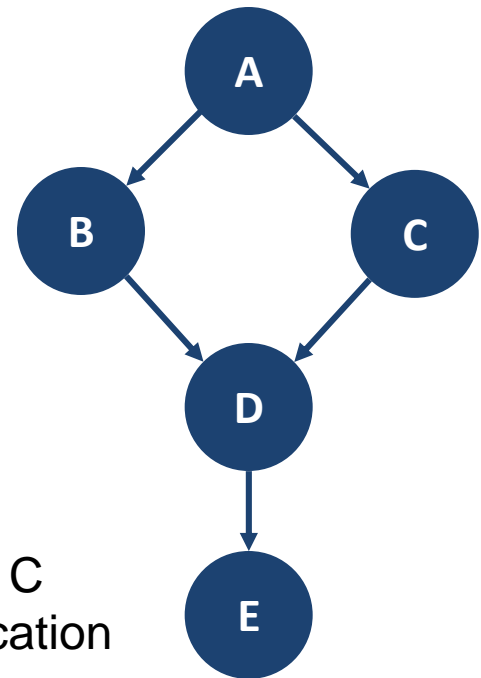
If the happens-before tells us that B and C need not be ordered, but all others are ordered, then we obtain the following graph on the right, also written in text as:

$$\preceq = \{ (A, B), (A, C), (B, D), (C, D), (D, E) \}$$

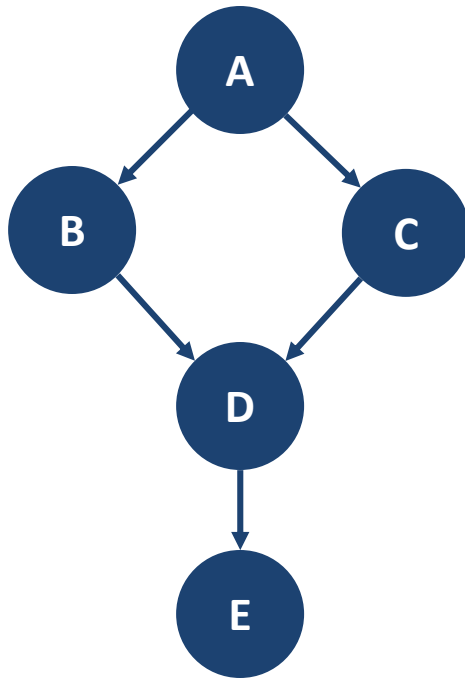
This graph captures that we not only have ABCDE as a trace but we also have ACBDE as a trace

In this example, we would **have a race** between B and C if actions B and C were touching the same memory location and one of them was writing to that location.

The DAG representing \preceq
(Hasse diagram)



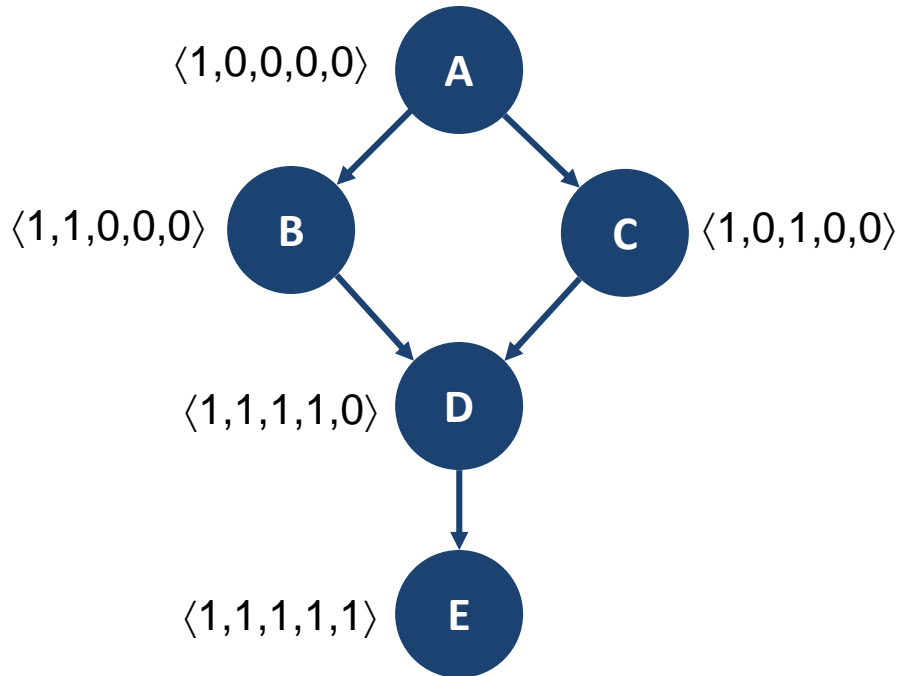
$a \stackrel{?}{\preceq} b$ via BFS



M - number of edges
N - number of nodes

Query Time: $O(M)$
Space : $O(N)$

? $a \preceq b$ via vector clocks



A vector clock vc is a map:

$$vc \in \text{Nodes} \rightarrow \text{Nat}$$

associate a vector clock
with each node

$$\langle 1,0,0,0,0 \rangle \sqsubseteq \langle 1,1,1,1,0 \rangle$$

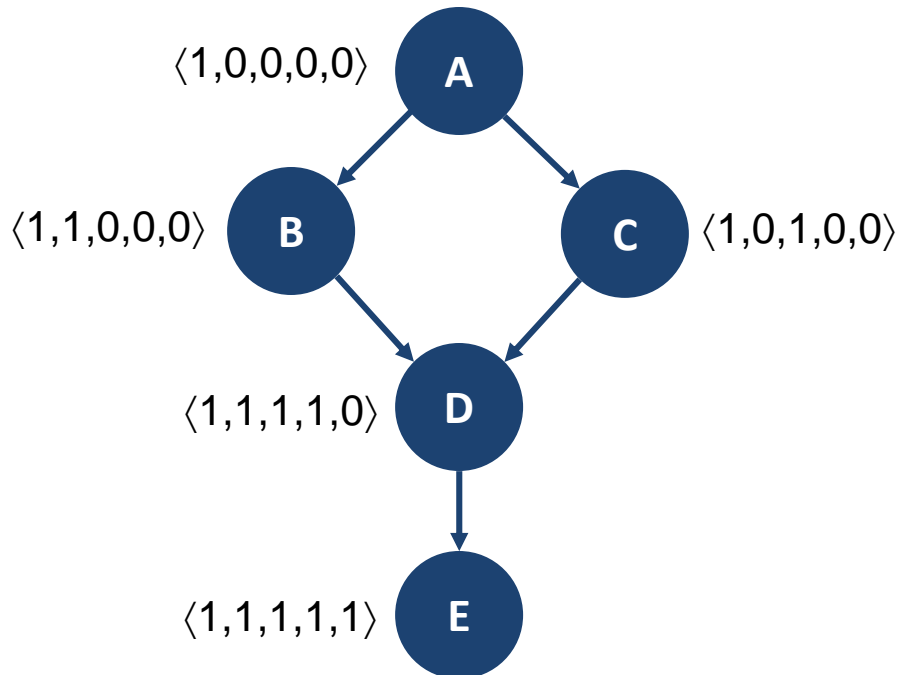
it follows that $A \preceq D$

$$\langle 1,1,0,0,0 \rangle \not\sqsubseteq \langle 1,0,1,0,0 \rangle$$

it follows that $B \not\preceq C$

In this example graph, $\text{Nodes} = \{A, B, C, D, E\}$

? $a \preceq b$ via vector clocks



At a given node, its vector clock captures who can reach that node.

For example, for node C, its vector clock $vc-C \langle 1, 0, 1, 0, 0 \rangle$ denotes that:

A can reach C: because $vc-C(A) = 1$

B cannot reach C: because $vc-C(B) = 0$

C can reach C: because $vc-C(C) = 1$

D cannot reach C: because $vc-C(D) = 0$

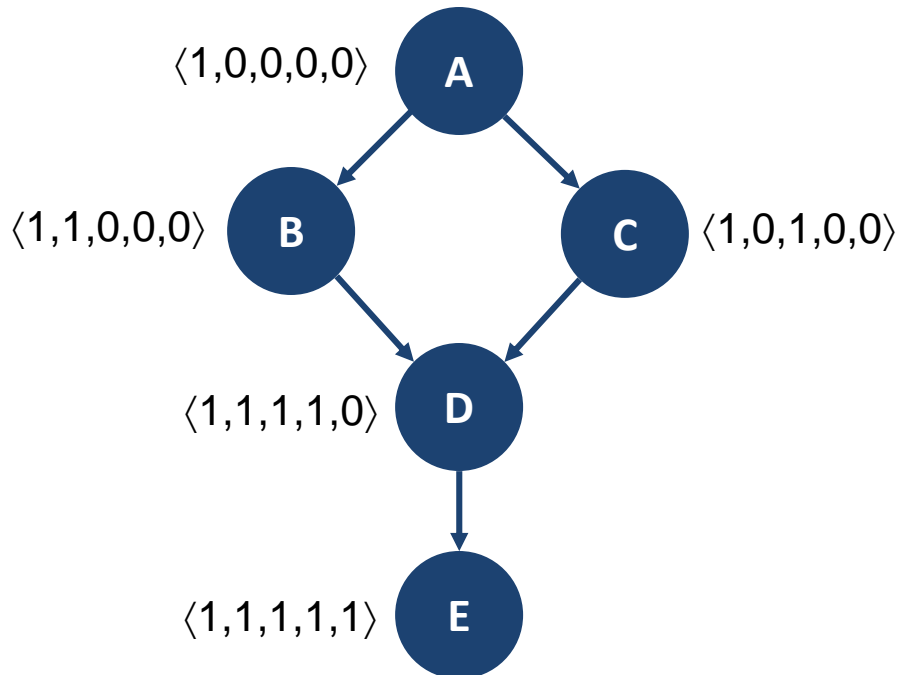
E cannot reach C: because $vc-C(E) = 0$

Given two nodes, say B and C, we can determine whether they are mutually unreachable by just checking:

whether $vc-C(B) = 0$ and $vc-B(C) = 0$

This is constant-time work.

? $a \preceq b$ via vector clocks



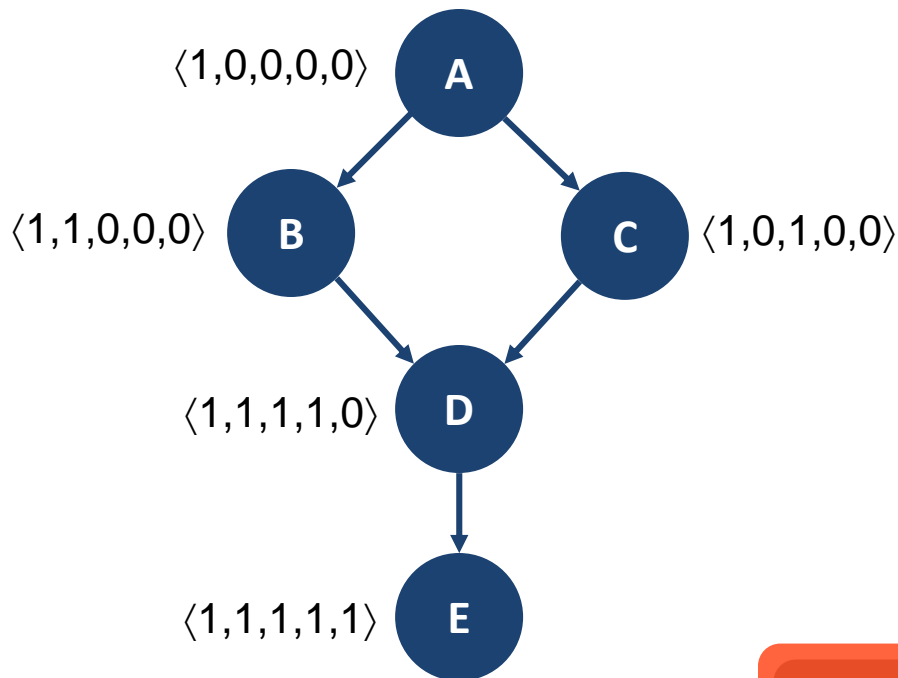
To compute the vector clocks, simply process each edge of the graph and join the vector clocks.

For instance, to compute the vector clock for node D, we may first process the edge from B to D, thereby copying the vector clock $\langle 1,1,0,0,0 \rangle$ from B to D.

Then, when we process the edge C to D, we will join (take the max) of the current vector clock at D ($\langle 1,1,0,0,0 \rangle$) and the vector clock coming from C ($\langle 1,0,1,0,0 \rangle$).

That is, for each edge we process, we do $O(N)$ work (as we need to iterate over each entry in the vector clock and the number of such entries is N).

? $a \preceq b$ via vector clocks



Pre-computation Time: $O(M \cdot N)$
(to obtain all vector clocks)

Query Time: $O(1)$
(for a pair of nodes)

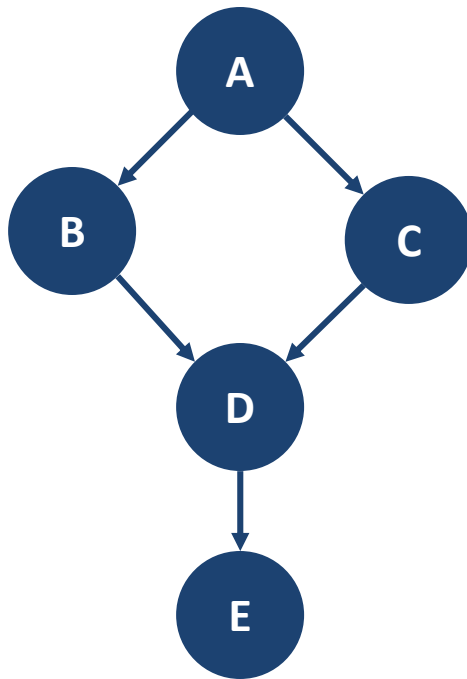
Space: $O(N^2)$



Space Explosion

? $a \preceq b$ via combining chain decomposition with vector clocks

Key idea: Re-discover threads by partitioning the nodes into chains.



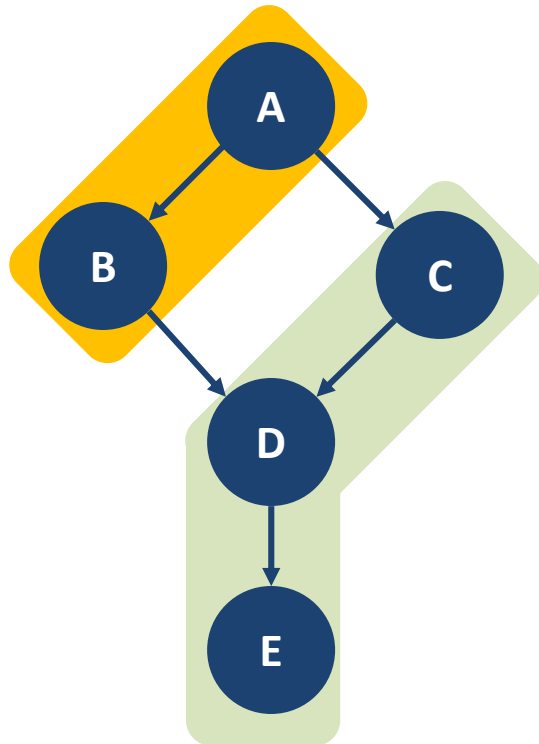
computes a map:

$$c \in \text{Nodes} \rightarrow \text{ChainIDs}$$

associate a chain with each node

? a \preceq b via combining chain decomposition with vector clocks

Key idea: Re-discover threads by partitioning the nodes into chains.

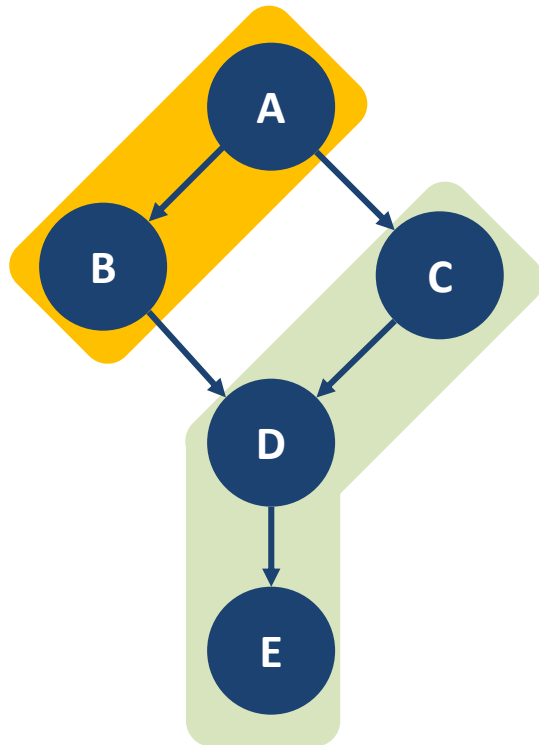


computes a map:

$$c \in \text{Nodes} \rightarrow \text{ChainIDs}$$

associate a chain with each node

$a \preceq b$ via combining chain decomposition with vector clocks (optimal version)



C = number of chains

Chain Computation Time: $O(N^3 + C \cdot M)$

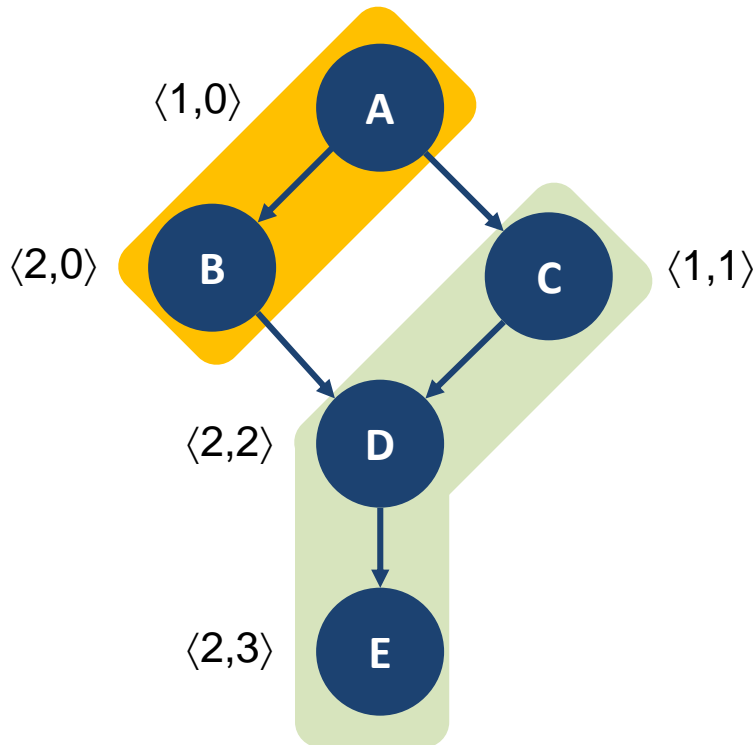
Vector clock computation: $O(C \cdot M)$

Query Time: $O(1)$

Space: $O(C \cdot N)$

Improved

[?] $a \preceq b$ via combining chain decomposition with vector clocks (optimal version)



C = number of chains

Chain Computation Time: $O(N^3 + C \cdot M)$

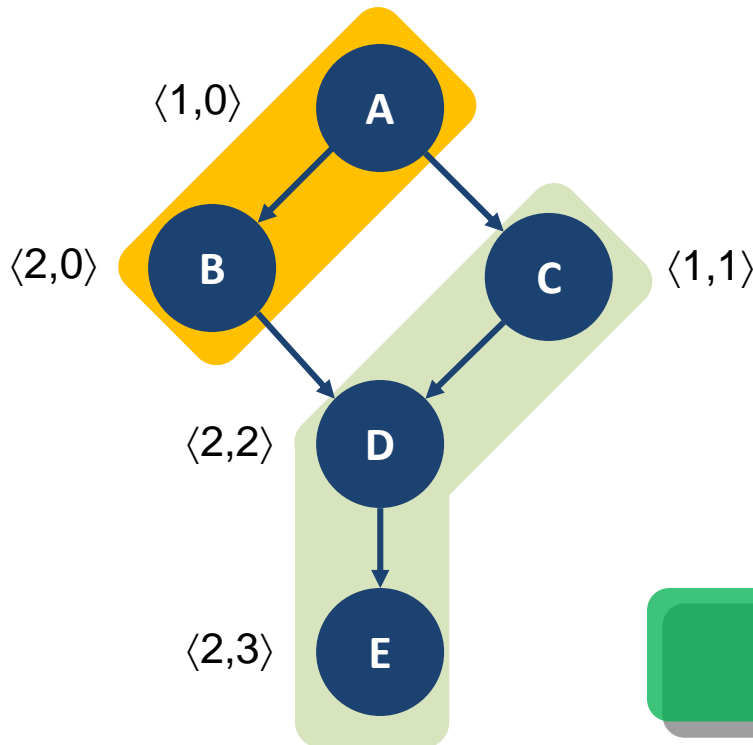
Vector clock computation: $O(C \cdot M)$

Query Time: $O(1)$

Space: $O(C \cdot N)$

Improved

[?] $a \preceq b$ via combining chain decomposition with vector clocks (greedy version)



C = number of chains

Chain Computation Time: $O(C \cdot M)$

Vector clock computation: $O(C \cdot M)$

Query Time: $O(1)$

Space: $O(C \cdot N)$

Improved

Improved

Two Challenges Affecting Steps 3 and 4

Synchronization with read/writes

race coverage eliminates all false races



Massive number of event handlers

greedy chain decomposition + vector clocks

space: $O(C \cdot N)$ where $C \ll N$



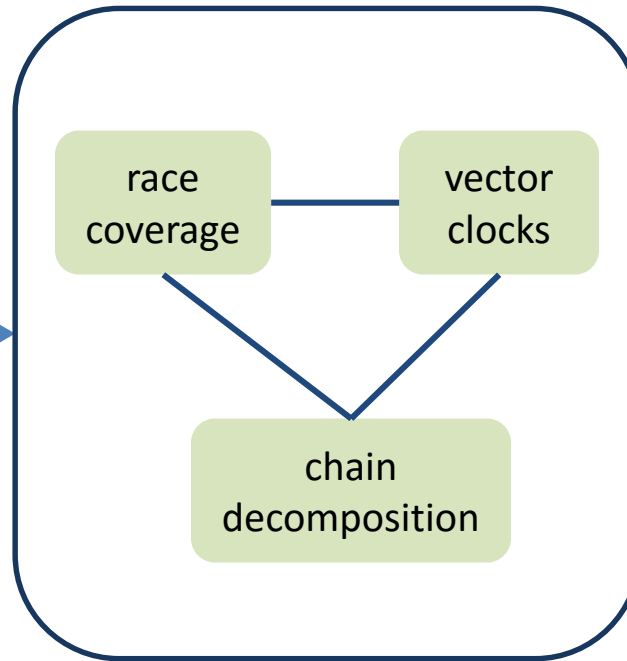
Race Detection: Web



Happens-
Before
Model

Memory
Locations

Race Detector



Pre-computation Time: $O(C \cdot M)$
Query Time: $O(1)$
Space: $O(C \cdot N)$

Step 5: Implement and Evaluate

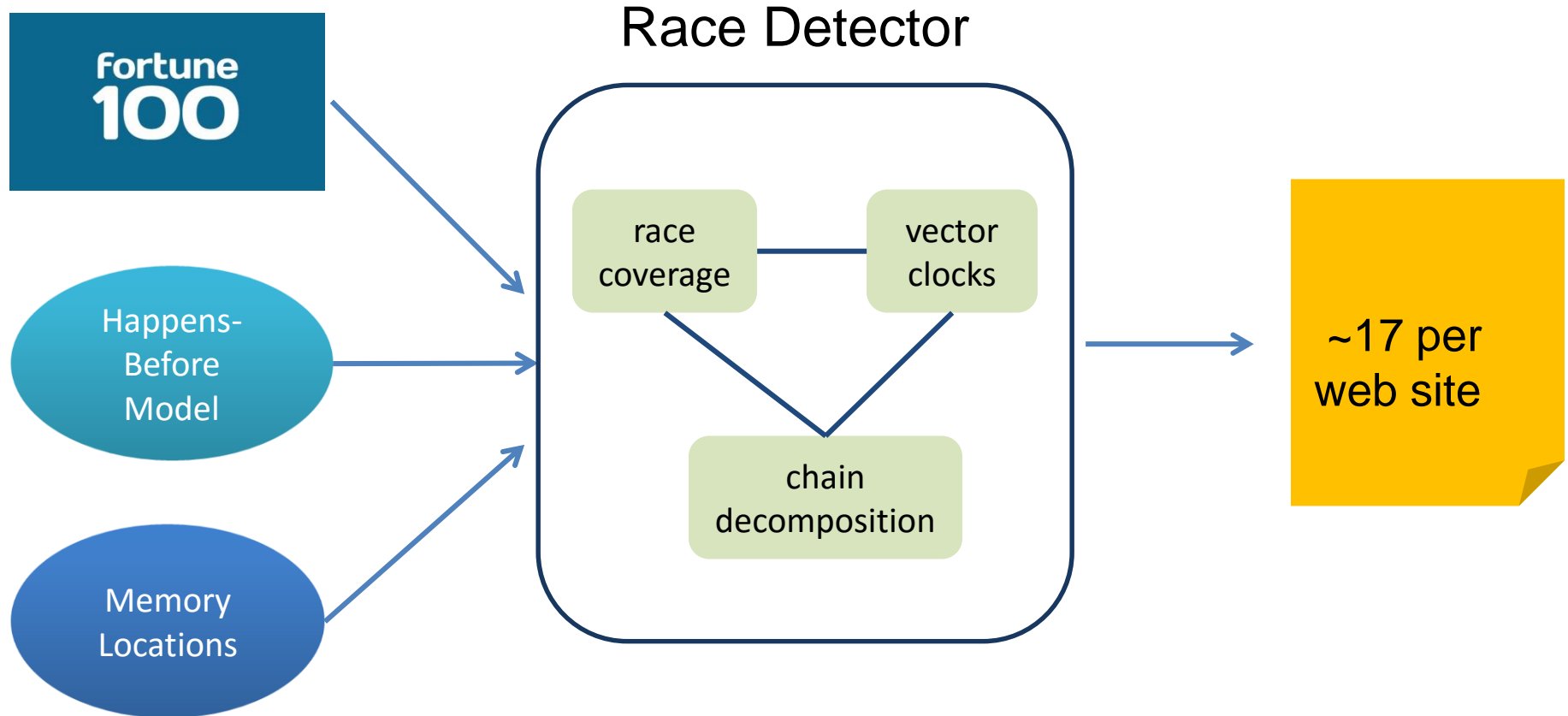
- Based on WebKit Browser
 - Used by Apple's Safari and Google's Chrome
- Check it out: <http://www.eventracer.org>

Step 5: Implement and Evaluate

We evaluate algorithm performance and precision

Hopefully algorithm is fast and does not report too many false positives on a wide range of applications

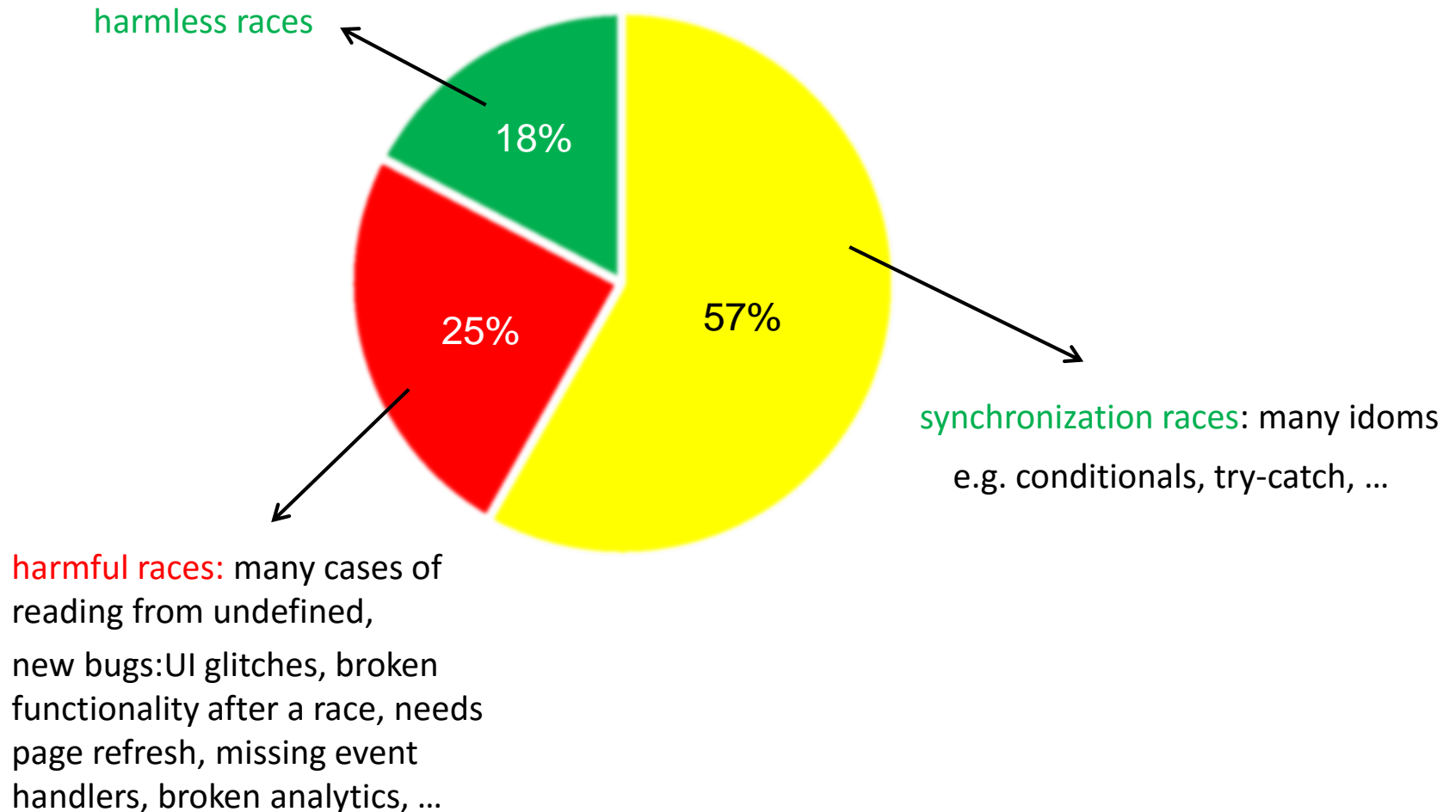
Experiments: Fortune 100 web sites



Race coverage: benefit

Metric	Mean # race vars	Max # race vars
All	634.6	3460
Only uncovered races	45.3	331
Filtering methods		
Writing same value	0.75	12
Only local reads	3.42	43
Late attachment of event handler	16.7	117
Lazy initialization	4.3	61
Commuting operations - className, cookie	4.0	80
Race with unload	1.1	33
Remaining after all filters	17.8	261

314 uncovered races: manual inspection



Algorithm: Space

Metric	Mean	Max
Number of event actions	5868	114900
Number of chains	175	792
Graph connectivity algorithm		
Vector clocks w/o chain decomposition	544MB	25181MB
Vector clocks + chain decomposition	5MB	171MB

Algorithm: Time

Metric	Mean	Max
Number of event actions	5868	114900
Number of chains	175	792
Graph connectivity algorithm		
Vector clocks w/o chain decomposition	>0.1sec	OOM
Vector clocks + chain decomposition	0.04sec	2.4sec
Breadth-first search	>22sec	TIMEOUT

Modern Dynamic Race Detection: 5 Steps

Step 1: Define Memory locations (on which races can happen)

Usually easy but there can be issues (framework vs. user-code)

Step 2: Define Happens-Before Model (how operations are ordered)

Can be tricky to get right due to subtleties of concurrency

Step 3: Come up with an Algorithm to detect races

Hard to get good asymptotic complexity + correctness

Step 4: Come up with techniques (algorithm, filters) to remove harmless races

Needs to answer what harmless means

Step 5: Implement Algorithm and Evaluate

Important to have low instrumentation overhead

Check it Out

Web: <http://www.eventracer.org>

Android: <http://www.eventracer.org/android>

All Open Source: <https://github.com/eth-srl/>