A Decentralized Approach for Monitoring Timing Constraints of Event Flows

Hyoseung Kim[*] Shinyoung Yi[*] Wonwoo Jung[*] Hojung Cha[†]

[*] LIG Nex1 Co., Ltd.
[†] Yonsei University, Korea

2010. 12. 3.

Hyoseung Kim
hyoseung@gmail.com

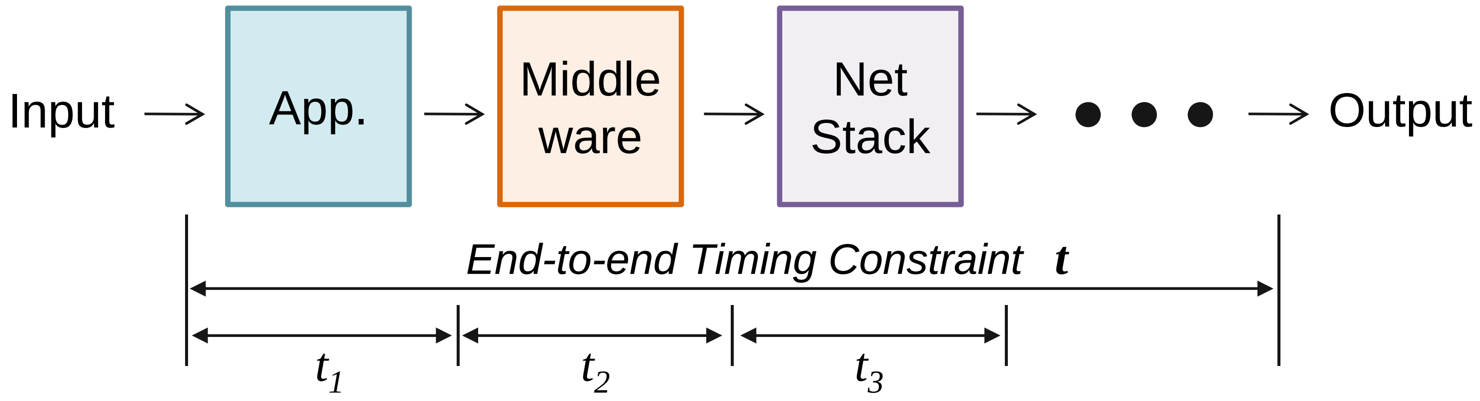
Run-time monitoring

- **Difficulties in guaranteeing timing constraints**
  - High S/W abstraction level
  - Diverse H/W techniques
    - Multi-core, caches, pipelines…
  - External environmental factors
    - Workload surges, weather condition, network status, …

- **Run-time monitoring is needed**
  - Identify the origin of timing faults
  - Clarify time consumption of each module

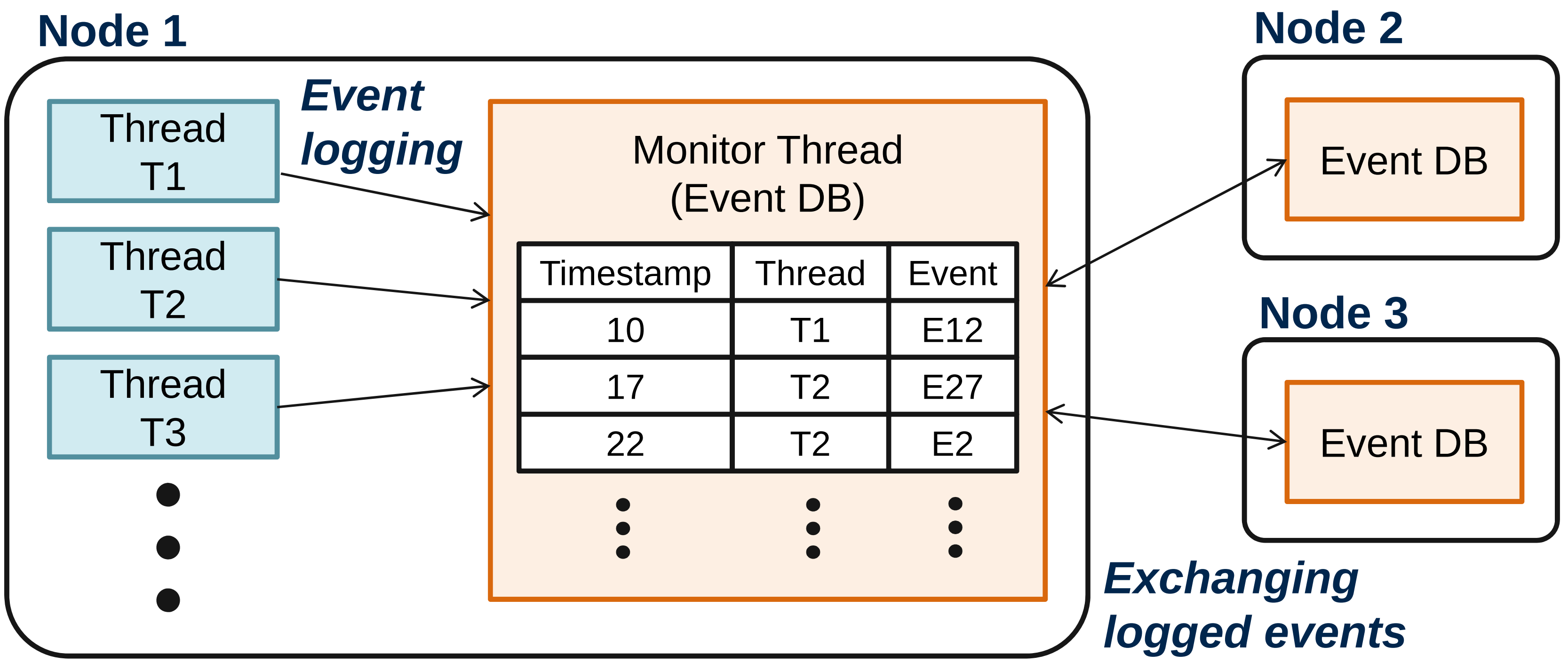
End-to-End Timing Constraints

- **Monitoring end-to-end timing constraints of event flows in distributed real-time systems**
  - Applications, device drivers, network stacks, …
  - Can be adapted to meet timing requirements of systems
  - Monitoring would provide developers with timing knowledge of various factors in a complex timing model



Challenges in Run-time Monitoring (1)

- **Typical architecture of existing run-time monitors**



Timestamp	Thread	Event
10	T1	E12
17	T2	E27
22	T2	E2

  – IPC for event logging: CPU overhead & unpredictable delay
  – Inter-node timing constraint: Additional network packets

Challenges in Run-time Monitoring (2)

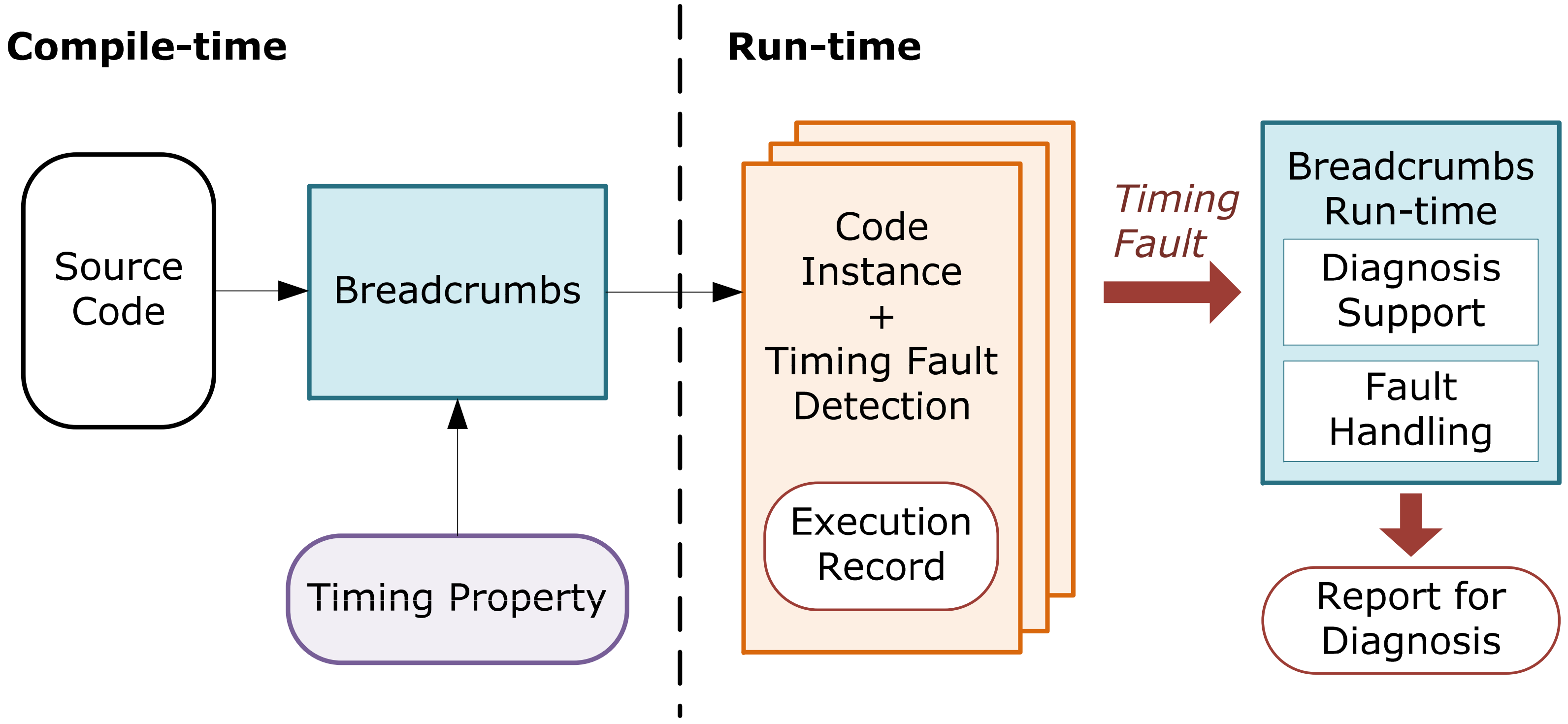
- **End-to-end timing constraints of event flows**
  - Existing monitors
    - Distinguish event order only with timestamps of events
    - Do not understand causal relationship of end-to-end events
  - Multiple intermediate paths
  - Out-of-order execution

$\rightarrow$ Hard to detect timing fault

Our Goal

- **Breadcrumbs: Run-time monitoring to detect end-to-end timing constraint violations**
  - Detect timing fault of end-to-end event flows
  - Provide run-time path of event flows
  - Acquire time consumption of each module on the path
  - Low run-time CPU overhead
  - No user intervention is needed beyond the event flow specification

Breadcrumbs: System Overview

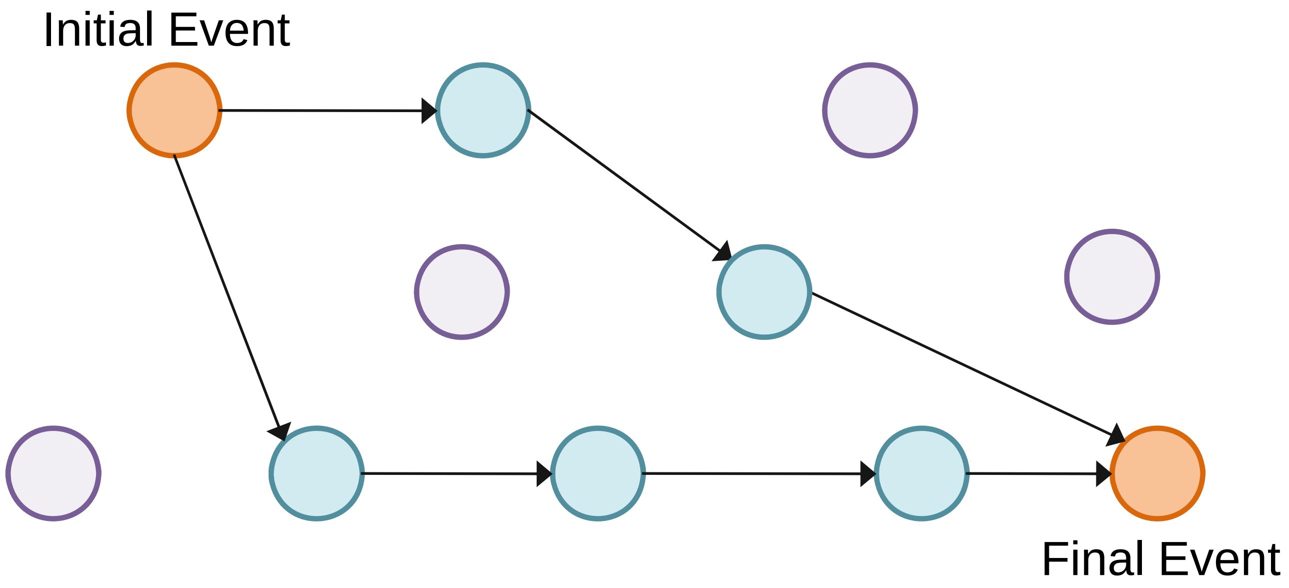


Definition of Event Flow (1)

- **Event**
  - State changes in the real-time system
    - Interrupts, IPC, system-calls, specific routines, …
  - We assume that functions are the basic units causing the state changes in the program model

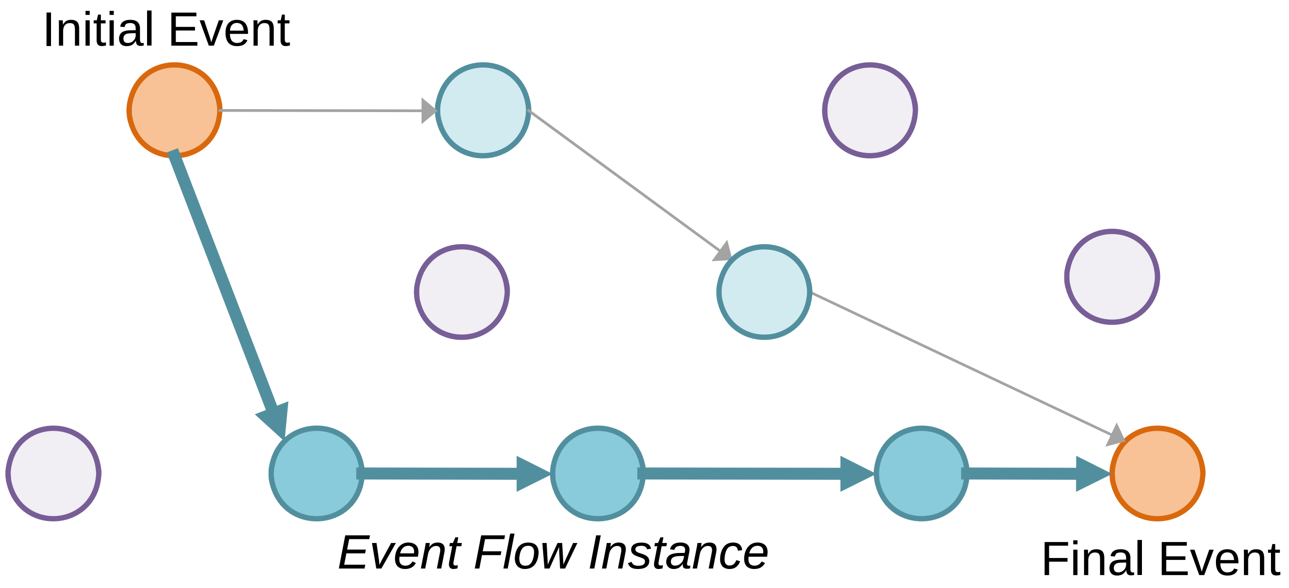
Definition of Event Flow (2)

- **Event Flow**
  - Set of event occurrences in causal relationship
  - Bounds monitoring scope of events in the system
  - Includes all possible events on the path from the initial event to the final event



Definition of Event Flow (3)

- **Instance of Event Flow**
  - Single causal chain of events
  - Every instance has the same initial & final event
  - Each instance can have different intermediate events



Timing Constraint Specification

Initial Event	***Function***	Start of *Function Name*
	Execution Context ID	Numeric | String
	Node ID	N/A | Numeric | Set of Numeric
Final Event	***Function***	(Start | End) of *Function Name*
	Execution Context ID	Numeric | String
	Node ID	N/A | Numeric | Set of Numeric
Deadline		Numeric
Fault Handling Method		Halt | Reboot | User Handler | …
Periodic Event (Option)	***Periodicity***	Yes | No
	Period	Numeric
	Error Bound	Numeric

Breadcrumbs

Event Flow Path Analysis

Timing Fault Detection

Fault Diagnosis Support

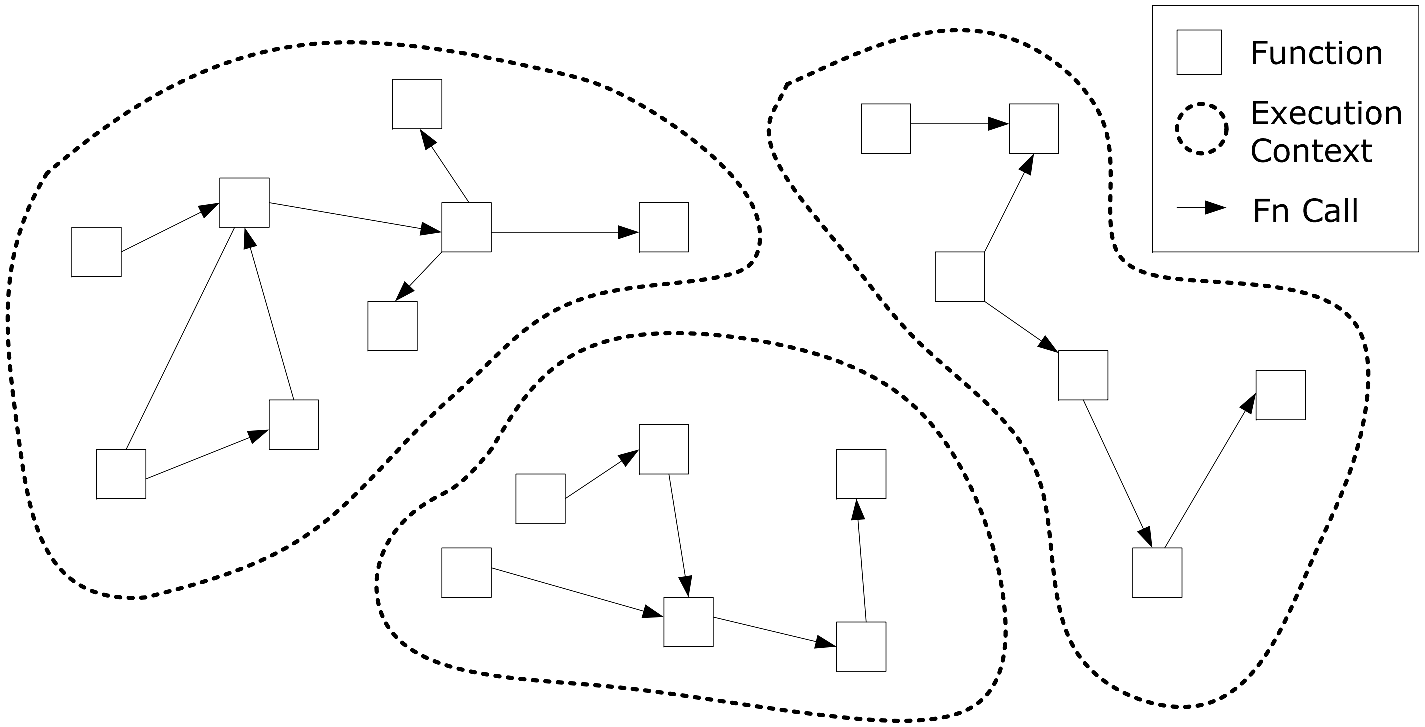
Event Flow Path Analysis (1)

- **Objective**
  - Insert timing fault detection routines for event flows

- **Steps**
  1. Identify execution contexts and construct a function call graph in each execution context
  2. Find out event passing between execution contexts
  3. Discover every node on possible paths from the initial event to the final event

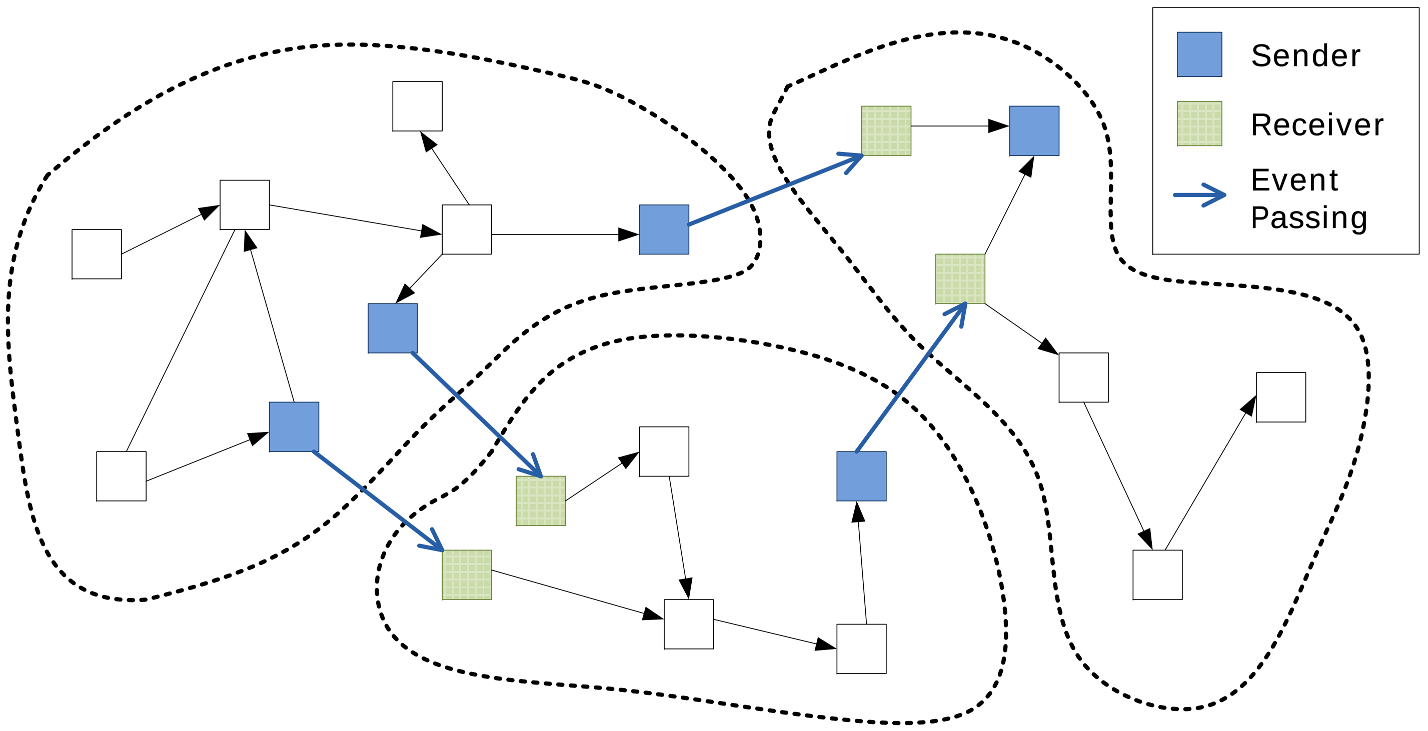
Event Flow Path Analysis (2)

- **Execution context and function call graph**
  - Identify execution contexts from source code
  - Construct directed edge graph (node = function)



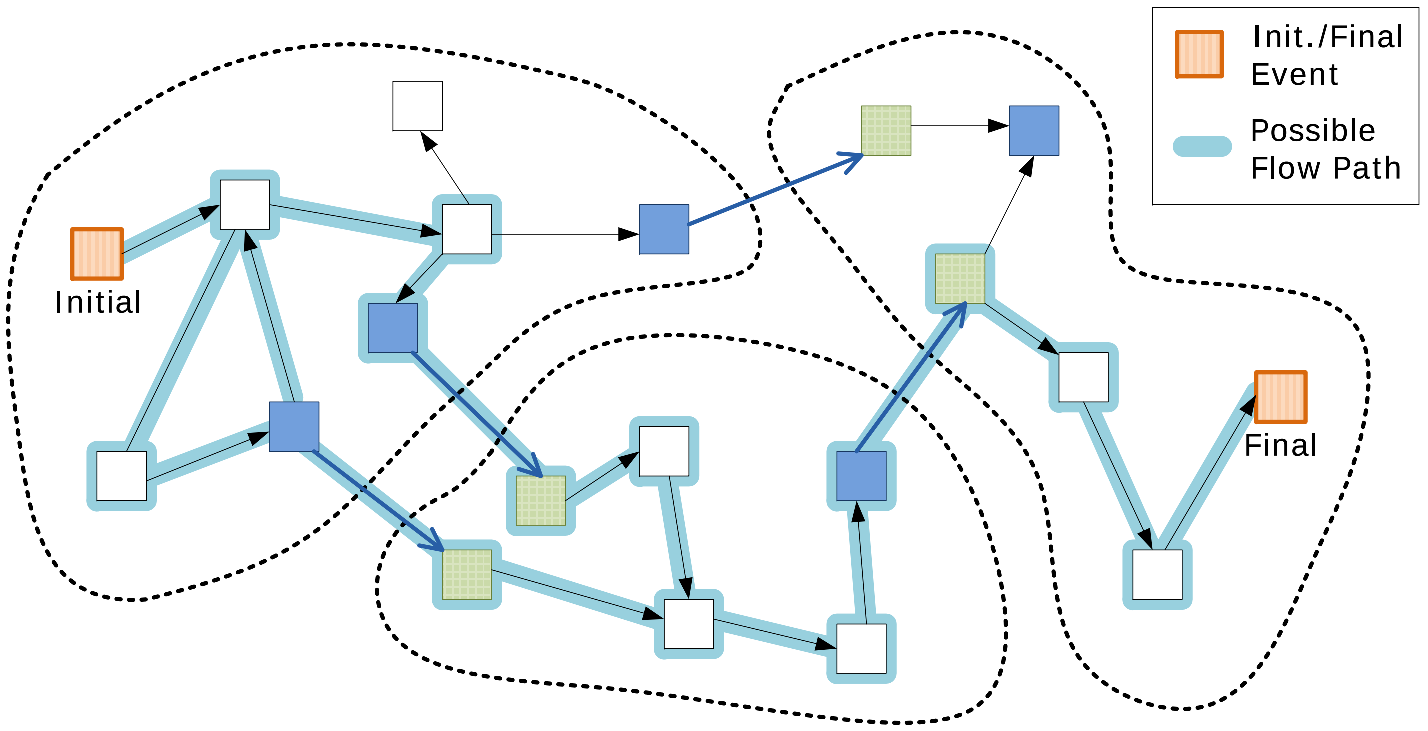
Event Flow Path Analysis (3)

- **Event passing between execution contexts**
  - In each execution context, find sender and receiver functions for passing messages to other execution contexts



Event Flow Path Analysis (4)

- **Discover every node on possible event flow paths**
  - Use a simple Breath-First Search
    - Initial event function to the final event function



Breadcrumbs

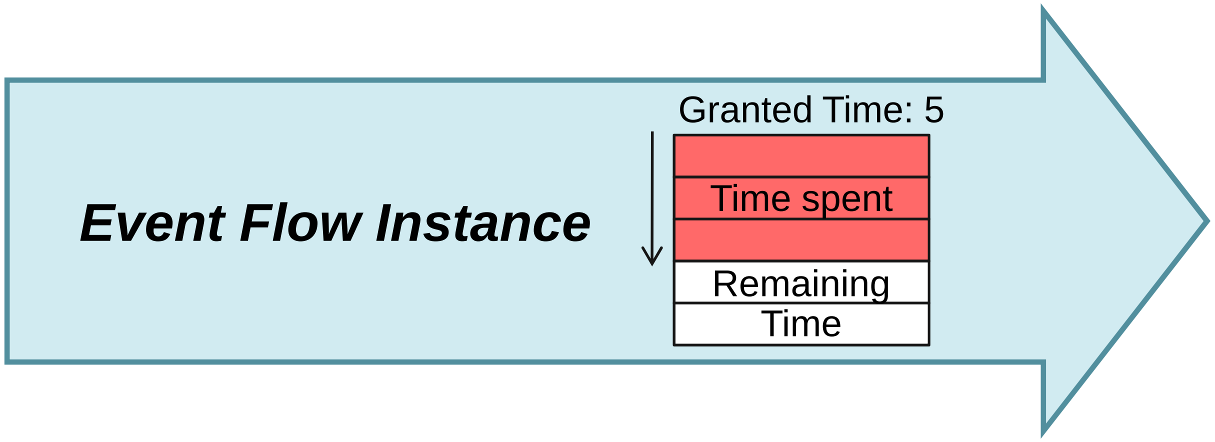
Event Flow Path Analysis

Timing Fault Detection

Fault Diagnosis Support

Timing Fault Detection

- **Transparently embedding timing information into event flow instances**
  - Deadline is granted to the event flow instance
  - Time consumption during the execution of the event flow instance is deducted from the granted time
  - The event flow instance can detect the deadline expiration by itself.

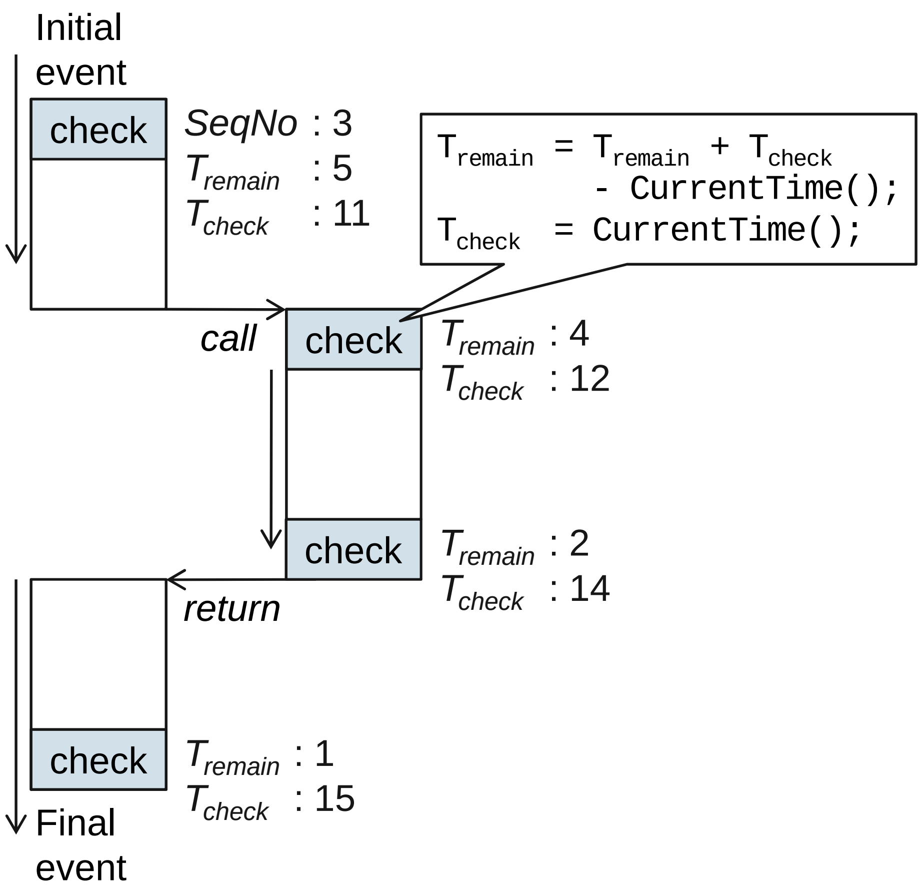


Detection in Single Execution Context (1)

- **Vars. to be declared in each execution context**
  - Save the timing information of the currently running event flow instance
  - $\boldsymbol{T_{remain}}$: saves the remaining time of the instance
  - $\boldsymbol{T_{check}}$: saves the system time when T_{remain} is updated
  - ***SeqNo*** : Event flow instance's unique identifier. Used to distinguish the instance from other instances

Detection in Single Execution Context (2)

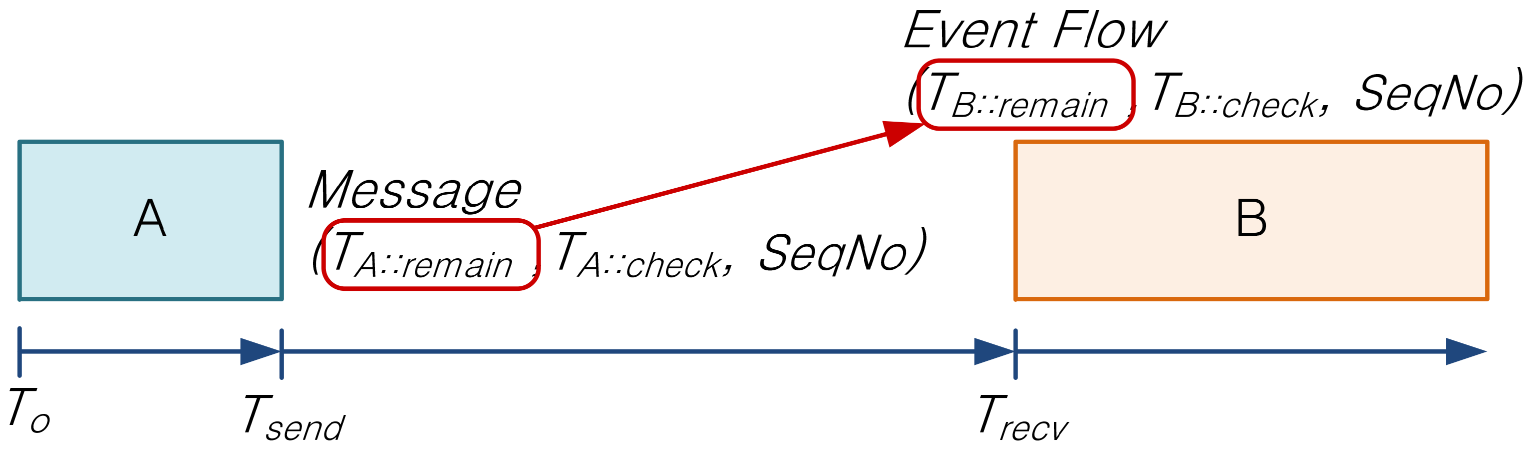
- **Timing fault checks**
  - Inserted at
    - Initial and final event functions
    - Intermediate functions on event flow paths
    - Senders/receivers for message passing btw. execution contexts
  - Initialize & update T_{remain}, T_{check}, *SeqNo*



```
Tremain = Tremain + Tcheck
          - CurrentTime();
Tcheck  = CurrentTime();
```

Detection of Multiple Execution Contexts

- **Event passing between execution contexts**



Event Flow
$(T_{B::remain}, T_{B::check}, SeqNo)$
A
Message
$(T_{A::remain}, T_{A::check}, SeqNo)$
B
T_o
T_{send}
T_{recv}

  – Transmit timing information in message itself *(Encapsulation)*
  – Only introduce small increase in the message size
    - Less monitoring interference, compared to generating extra messages
  – Time synchronization between adjacent nodes is required

Breadcrumbs

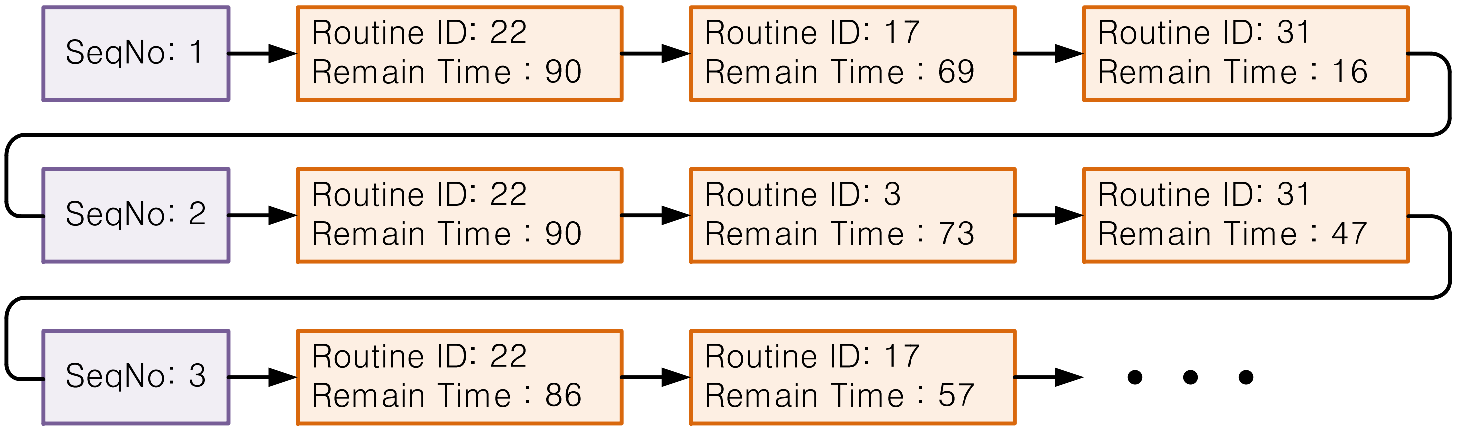
Event Flow Path Analysis

Timing Fault Detection

Fault Diagnosis Support

Fault Diagnosis Support

- **Repository for execution time history**
  - *{SeqNo*, (*Check Routine's ID*, T_{remain})*}* are saved
  - Execution time histories can be referenced by the sequence number when the timing fault occurs
  - Implementation of examining repositories can be differ
    - Dumping memory
    - Sending a query to every execution context

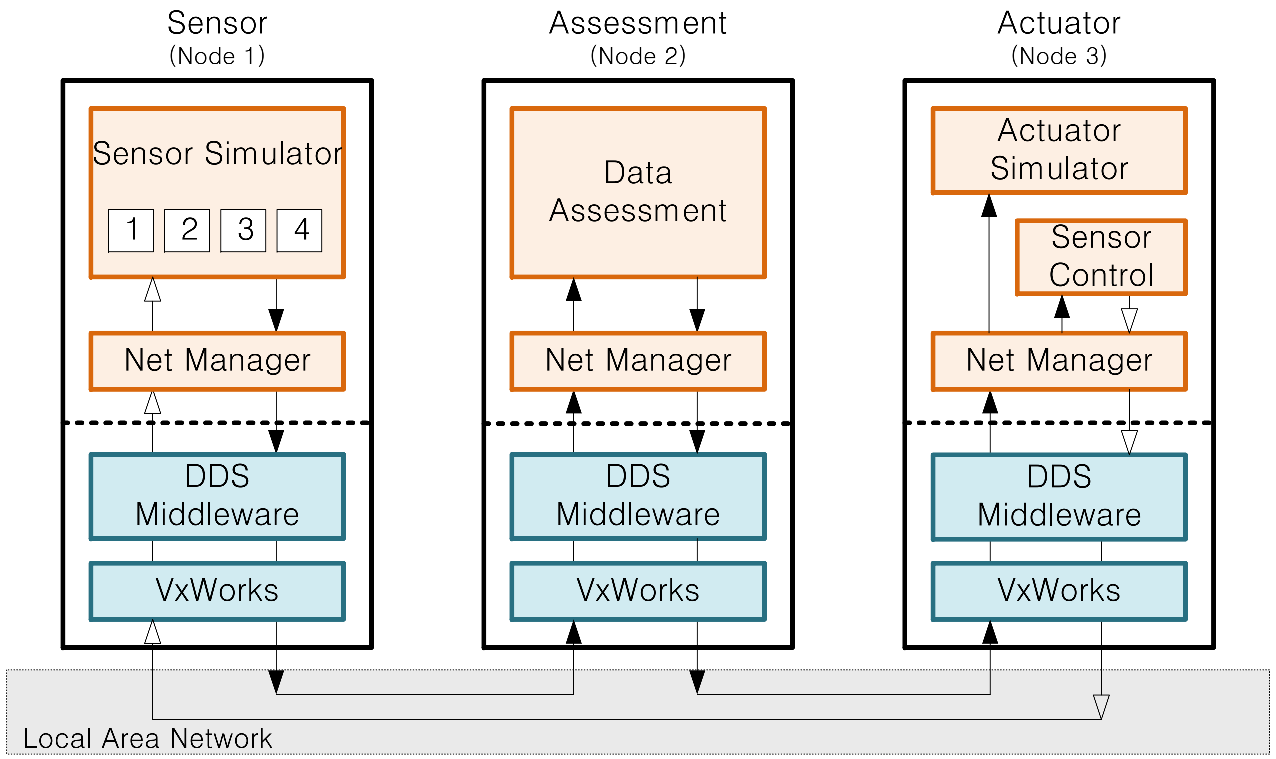


Timing Fault Handling

- **Fault handling**
  - Executed when a timing violation is detected
- **Example 1: Halt/Reboot**
  - System perform safe termination or reboot after the fault diagnosis support
- **Example 2: User defined function**
  - In a system where the timing faults only affect the quality of service, user function could compensate or disregard a delayed event flow instance.

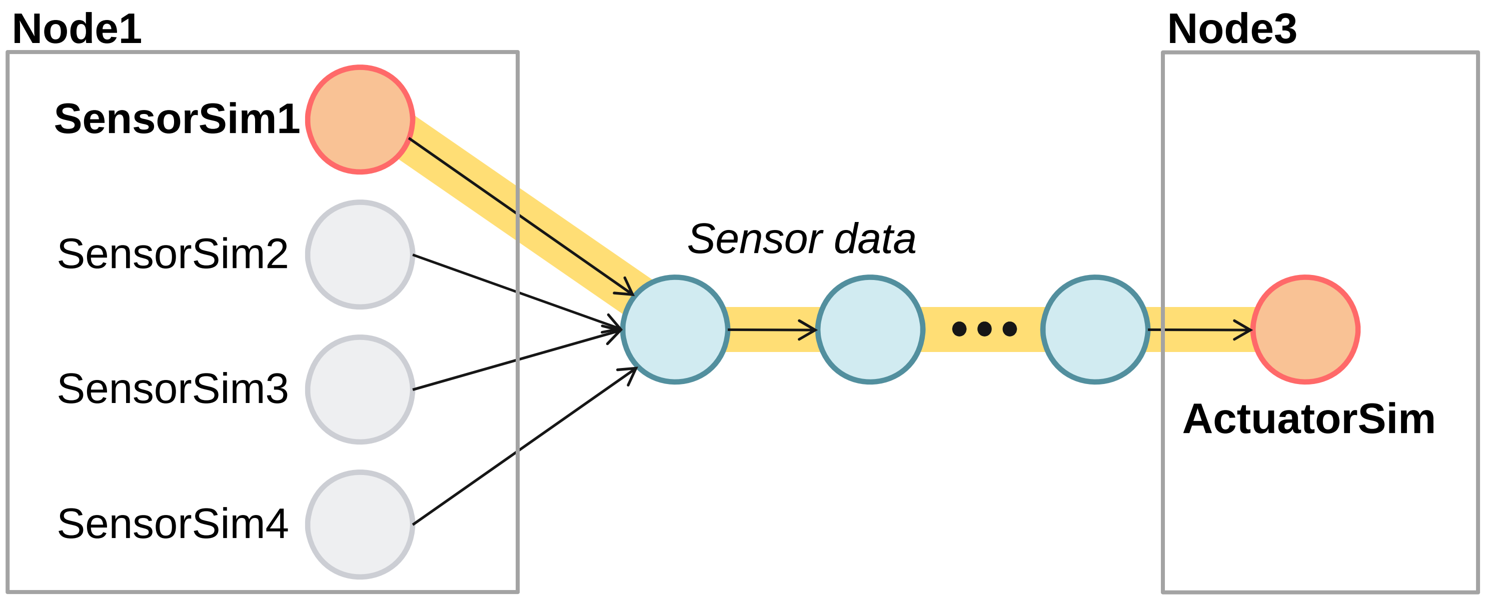
Implementation

- **Target System**
  - Rational RoseRT (C Version) + RTI DDS + VxWorks
  - Each node: PowerPC 7447A 1GHz, 1GBytes RAM



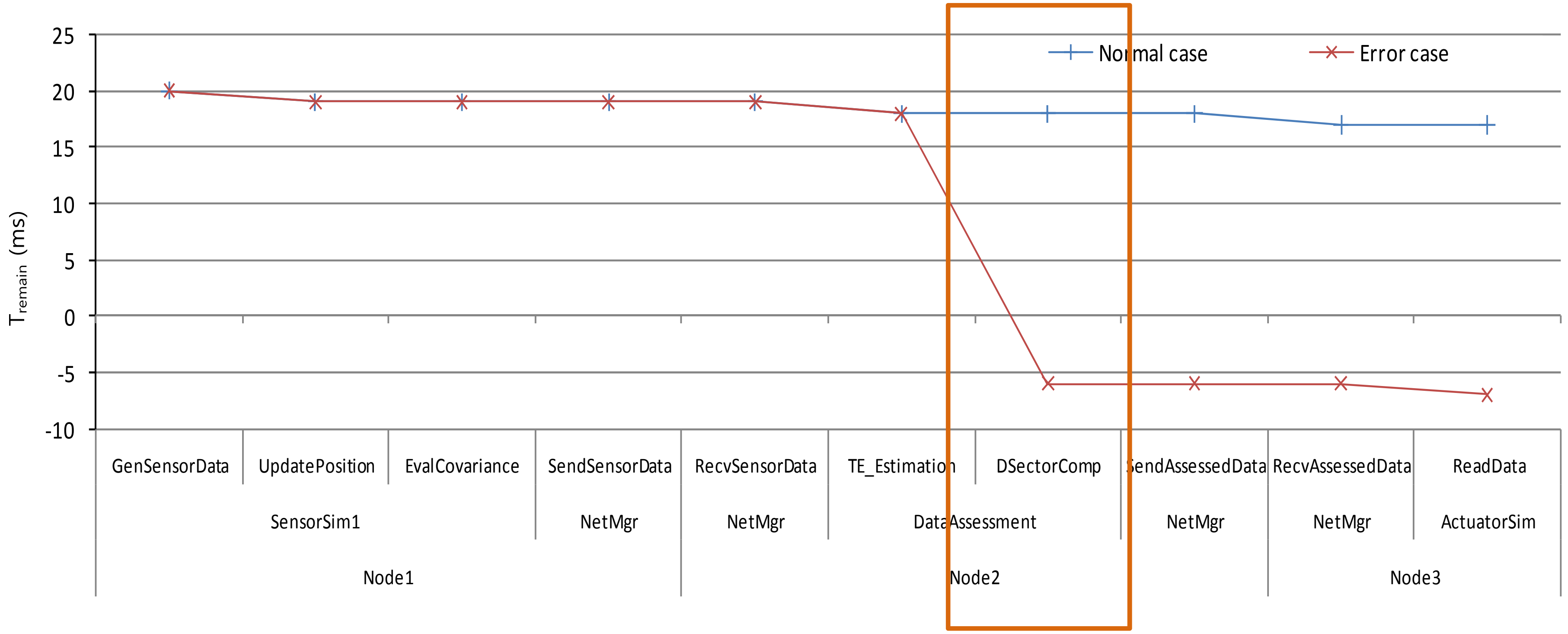
Experiment (1)

- **Timing Constraint 1**
  - Event flow: SensorSim1(Node1) ~ ActuatorSim(Node3)
  - Same intermediate path with other event flows
  - Injected buggy code for time delay & out-of-order execution



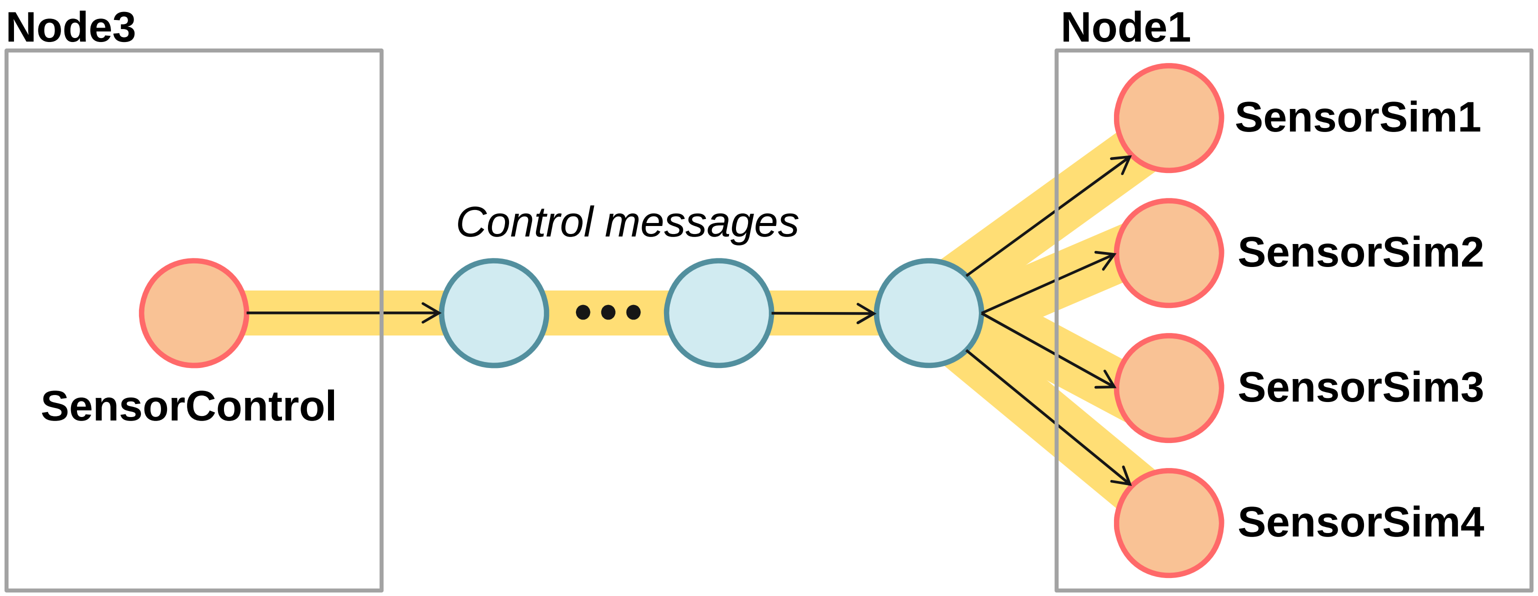
Experiment (2)

- **Timing Constraint 1: Fault Diagnosis**
  - Remaining time on normal and error case



Experiment (3)

- **Timing Constraint 2**
  - Event flow: SensorControl(Node3) ~ SensorSim1-4(Node1)
  - Single initial event is diverged into multiple final events



Experiment (4)

- **Increased message size**
  - 14 bytes are appended to messages
  - In our test application, increased message size < 10%
- **CPU Overhead**

Samples	Original Program			Breadcrumbs		
	Node1	*Node2*	*Node3*	*Node1*	*Node2*	*Node3*
# 100	4.2%	7.3%	3.2%	4.3%	7.3%	3.3%
# 200	6.1%	12.5%	5.2%	6.1%	12.6%	5.2%
# 400	14.2%	28.7%	10.7%	14.3%	29.0%	10.7%
# 800	32.4%	67.2%	22.1%	32.6%	67.7%	22.1%

Discussion (1)

- **Amount of execution time history to be saved**
  - Needs to be determined by users
  - Considering the frequency of event occurrences and the time required to begin fault diagnosis support after the detection of timing violation
  - Timing fault detection can perform regardless of the amount of execution time history
- **Code recompile**
  - Programs have to be recompiled if a user modifies a timing constraint
  - Only some of the source code files need to be recompiled

Discussion (2)

- **Time synchronization between adjacent nodes**
  - Previous run-time monitors for distributed systems need global time synchronization
  - Eased assumption of our approach can be an advantage in the case of mobile network
    - Elapsed Time on Arrival in WSN

Conclusion

- **Breadcrumbs**
  - Monitoring timing constraints of end-to-end event flows
  - No IPC, No extra thread, No extra network packets
  - Explicitly identifying event flow instances
- **Future Work**
  - Port Breadcrumbs to diverse real-time distributed systems
  - Adaptive scheduling policy based on remaining time of event flow instances

Thank you

Related Work

- **Hardware-assisted monitoring**
  - Pros: Non-Intrusiveness
  - Cons: High cost & Lack of portability
- **Software-only monitoring**
  - Insert code for generating events
  - Pros: Flexibility
  - Cons: Intrusiveness

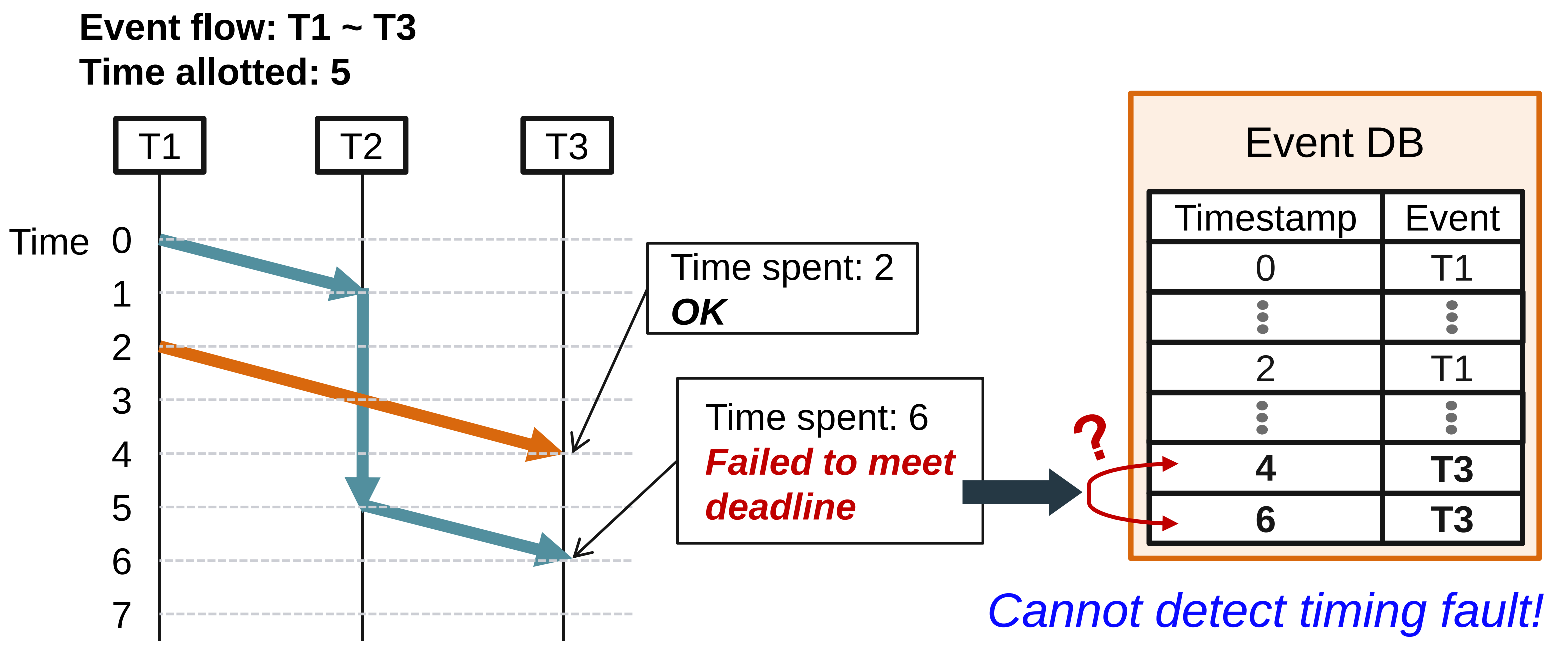
Post-mortem analysis

- **Run-time monitoring**
  - Collect & Analyze events at run-time
  - Real-Time Logic (RTL) : Timing constraints & behavioral conditions

Challenges in Run-time Monitoring

- **End-to-end timing constraints of event flows**
  - Multiple intermediate paths
  - Out-of-order execution

→ Hard to detect timing fault



Timestamp	Event
0	T1
⋮	⋮
2	T1
⋮	⋮
4	**T3**
6	**T3**

Cannot detect timing fault!