A Decentralized Approach for Monitoring Timing Constraints of Event Flows

Hyoseung Kim*, Shinyoung Yi*, Wonwoo Jung*, and Hojung Cha†
*LIG Nex1 Co., Ltd., Korea †Yonsei University, Korea
{hyoseung.kim, shinyoungyi, wonwoo.jung}@lignex1.com, hjcha@cs.yonsei.ac.kr

Abstract

This paper presents a run-time monitoring framework to detect end-to-end timing constraint violations of event flows in distributed real-time systems. The framework analyzes every event on possible event flow paths and automatically inserts timing fault checks for run-time detection. When the framework detects a timing violation, it provides users with the event flow's run-time path and the time consumption of each participating software module. In addition, it invokes a timing fault handler according to the timing fault specification, which allows our approach to aid the monitoring and management of the deployed systems. The experimental results show that the framework correctly detects timing constraint with insignificant overhead and provides related diagnostic information.

1. Introduction

Providing guaranteed timing constraints in a distributed real-time system remains a difficult issue. The abstraction level of real-time system software has increased and system performance has improved; however, deriving a performance and timing estimation model is extremely complicated. In addition, external environmental factors, such as unexpected workload surges, severe weather conditions, and network disconnection, affect systems outside the ideal conditions of a laboratory. Despite formal verification methods and real-time scheduling, a real-time system can cause a timing constraint violation due to unexpected behavior of system components that imperfectly deal with the real-life environment. For these reasons, gradual system complement through repeated testing and feedback on timing faults is now a generally used process in the development of real-time systems; however, extensive time and costs are required to develop and stabilize the system.

Run-time monitoring to detect and debug timing constraint violations of distributed real-time systems is therefore an important research area. Run-time monitoring helps identify the origin of timing faults and clarifies time consumption of each module on event propagation routes. We argue that monitoring timing constraints of distributed real-time systems should focus on the timing analysis of event flows rather than on timing constraints of a single event or a set of timing constraints of irrelevant events. To understand timing issues of distributed real-time systems, challenges generally regarded as part of separate abstraction layers, such as interrupt handling, task scheduling, and temporal network congestion, should be considered together. Applications, device drivers, network stacks, and hardware devices can be adapted to meet timing requirements of these systems, and monitoring event flows across separate abstraction layers would provide developers with timing knowledge of various factors in a complex timing model, helping them efficiently design the real-time system.

Previous researchers [1][2][32] investigating run-time monitoring of real-time systems used Real-Time Logic (RTL) [3] to define their timing constraints. RTL is a suitable specification for representing timing constraints, as well as behavioral conditions, but monitors based on RTL mainly focus on event occurrences and invariant relationships between events; thus they are inadequate to deduce a run-time path of event flow and time consumption that causes timing violations of each module on the path. The monitors depend on external threads to record event occurrences for timing verification and require exchanging collected event occurrences between distributed nodes to check inter-node timing constraints. This makes them inappropriate for applying to a mobile ad-hoc network and a wide area network, where the cost of transmitting network packets is expensive. Other run-time approaches, including data loss and latency monitoring [5], confidence threshold based monitoring [4], and end-to-end delay distribution analysis [6], have been studied, but these are not focused on detection and detailed diagnosis of timing constraint violations.

Therefore, this paper presents Breadcrumbs[1], a run-time monitoring framework to detect end-to-end timing

[1] Breadcrumbs comes from cutting bread off and scattering it to find a way back home, as in the fairy tale "Hansel and Gretel."

constraint violations of event flows in distributed systems. The advantages of Breadcrumbs are as follows:

- Breadcrumbs detects timing constraint violations of event flows occurring in a single system and in a distributed system.
- Users can identify a distributed node and a software module where the event flow violation is detected and obtain historical records of the event flow up to when the fault occurred.
- Users only need to specify timing constraints for the target system in a simple format. Other user interventions, such as advance analysis of possible path of event flows and manual code manipulation, are not required.

To achieve these advantages, Breadcrumbs first requires timing constraint specifications of event flows to users. Then, it finds every event on possible event flow paths and inserts the event flow monitoring code at each event. Breadcrumbs detects timing violations by transparently embedding timing information into event flow instances instead of using an external monitoring thread to record events. Through this decentralized monitoring approach, Breadcrumbs is able to detect timing violations of event flows without creating an extra monitoring thread and without transmitting additional network packets to check inter-node timing constraints. When a timing fault occurs, Breadcrumbs executes an appropriate fault handler in the timing constraint specification and provides diagnostic information, namely execution time records distributed on the event flow path.

The practicality of our approach has been proven by the implementation of Breadcrumbs for applications using Rational Rose Real-Time (RoseRT) [14], a real-time modeling tool, and Data Distribution Service (DDS) [16], a network middleware for real-time distributed systems. With the increasing popularity of COTS tools like RoseRT and DDS for modeling and developing real-time systems in the industry, we see the target environment as a good candidate for monitoring timing constraints. In the experimental section, we confirm that Breadcrumbs correctly detects timing constraint violations with less than 1% of computation overhead in a distributed environment.

2. Related Work

Monitoring of real-time systems has been studied to collect relevant information from the target systems and to present analyses based on that information. In hardware monitoring approaches [23][24], dedicated hardware devices are used to detect event occurrences. The device snoops the busses of the target systems and matches bus signals to store events for post-processing. Non-intrusiveness is the advantage of these hardware monitoring approaches, but high costs and a lack of portability are disadvantages, as these approaches require dedicated hardware that depends on the target system. ARTS [22] is a software monitoring approach that introduced a real-time monitoring for distributed systems. The ARTS kernel regards a state change of a process as an event. The kernel captures events and sends them from the target host to the remote host periodically for further processing. Dodd et al. [20] proposed a monitoring system that inserts code for generating events into applications and an operating system kernel to detect event occurrences in a distributed environment. The software approaches [20][22] have the disadvantage of intrusiveness, which means monitors may perturb or influence target systems, but the advantage of flexibility due to their independence from low-level hardware details. The aforementioned hardware and software monitoring approaches mainly focus on the detection of events, so they are useful in post-mortem analysis of collected data, but are insufficient for run-time detection of timing constraint violations.

Run-time monitoring techniques have been proposed not only to collect relevant events from target systems, but also to process these collected events in run-time. In most research about run-time monitoring, RTL [3] is used in the definition of timing constraints. Chodrow et al. [2] proposed two general methods for synchronous and asynchronous monitoring of real-time constraints in a single system. A constraint-graph based algorithm for detecting timing constraint violations is also described in their paper. Raju et al. [21] and Jahanian et al. [32] extended timing constraint specification and the violation detection algorithm of [2] to distributed real-time systems. They show that minimizing the amount of information exchanged between processors (distributed nodes) for constraint violation detection is NP-hard. Mok et al. [1] proposed a method to catch timing constraint violation at the earliest possible time, but their method does not support the monitoring of distributed systems. In these run-time monitoring approaches [2][21][32][1], IPC is used to store event occurrences of application tasks into a separate monitor task, and timing constraint violations are checked by the monitor task to minimize the use of target program resources. As IPC typically uses locks in its implementation for mutual exclusion between tasks, the use of IPC may cause unpredictable delay in the target systems [25]. Though a recently generalized multi-core environment is expected to reduce the calculation burden of monitor tasks, the delay caused by IPC still exists.

Moreover, [2][32][21][1] are not suitable to monitor end-to-end timing constraints of event flows. As RTL formula distinguishes event order only with timestamps of events, it is hard to check timing constraint violations when an event flow has multiple intermediate paths and the previously initiated event

flow instance finishes later than the termination of the latter event flow instances. As the granularity of timestamp determines the minimum observable spacing between two consecutive events, a fine-grained timer is required to collect frequent event occurrences. On the other hand, Breadcrumbs explicitly maintains the sequence number of each event flow instance, so that it can detect consecutive events regardless of timestamp granularity and accurately detect timing constraint violations, even in the case of significant timing variations between different event flow instances.

Recently, many studies have focused on monitoring the wireless sensor network, which consists of resource-constraint distributed nodes. SNMS [29] focuses on collecting relevant data for post-mortem analysis with minimal overhead, but does not provide facilities for run-time detection. PD2 [5] collects performance related factors, such as data latency and message loss, and is motivated by PDB [11], which is a performance debugging for distributed systems of black-box components. However, PD2 performs event logging only when a user manually triggers monitoring. This saves the battery resources of sensor network nodes but is not suitable to detect temporal constraint violations. [8][9][10][30] gather run-time information of wireless sensor networks for debugging purposes. With these techniques, developers can use collected data to find program bugs, but facilities are not provided to detect timing constraint violations. Wang et al. [6] addressed that it is difficult to satisfy the worst-case QoS requirement due to the nature of wireless connectivity in a sensor network environment, and thus investigated the distribution of an end-to-end delay in multi-hop sensor networks to provide probabilistic QoS guarantees. Simulation techniques [26][27][29][31] are introduced to reduce time in deploying systems to the real environment, but they should be used as a complement to the run-time monitoring approaches rather than as a substitute.

3. Event Flow and Timing Constraints

In this section we define the term *event flow* as used in our paper and present the timing constraint specification for the event flow.

3.1. Event Flow Definition

The computation of a real-time system can be seen as a sequence of event occurrences [2]. "Event" refers to state changes in the system, e.g., interrupt occurrences; IPC; execution of specific calculation routines, libraries, or system-calls; and sending and receiving network packets. In this paper, we assume that functions are the basic units causing these state changes in the program model. At each function, two events occur as function call and return. At each execution context calling the function and at each node in a distributed system running the same program, independent event instances occur.

Table 1. 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

An event occurrence can cause another event, and this event can in turn cause other events. We define the set of event occurrences in this causal relationship as the event flow. The specification of the event flow is used to set the monitoring scope of complex and diverse events in the system and to distinguish these events from other irrelevant events.

The event flow is defined by exactly two events, which differ from each other, and is a collection of causal chains, starting from e_1 and ending at e_n. e_1 is the initial event that initiates the event flow, and e_n is the final event that indicates the end of the event flow. By specifying the initial event and the final event, the event flow is established. There is a single e_1 and e_n, but for $\forall\, i \in \{2, ..., n-1\}$, there can be many different e_i. This means that the event flow includes all possible events on the path from the initial event to the final event.

The instance of event flow is a single causal chain of events. This means that every instance of a specific event flow has the same initial event and final event, but each instance can have different intermediate events. For example, in a distributed system, an event flow can be established with an initial event as sending a message from one particular node to a central server and with a final event as receiving the message at the server. Here, our definition for the event flow instance means that the network routing path from the node to the server may change in each event flow instance.

3.2. Timing Constraint Specification

Based on the definition for event flows, we specify timing constraints to be used by Breadcrumbs. The specification is written by users, but this is not difficult because it requires only minimal knowledge of an initial event and a final event in the target programs. Table 1 shows a timing constraint specification for an event flow.

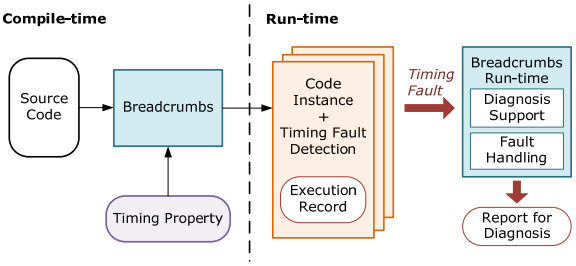

Figure 1. System overview of Breadcrumbs

An initial event, a final event, a deadline, and a fault handling method for an event flow are essential to describe the timing constraint specification. The initial event and final event are specified by a set of function names, execution context IDs, and node IDs. The initial event must be set as a start position of a function, while the final event can be specified as both a start and an end position of a function. The execution context in the specification includes not only threads and interrupt handlers, but also schedulable and logically independent application-level execution models, such as *task* in TinyOS execution model [7], Microsoft Windows Message Loop [13], and Capsule in Rational RoseRT [14]. In the case that a single program is executed on several nodes in a distributed system, the node ID field in the specification can be represented as a set of node IDs. The fault handling method is executed when a timing constraint violation of the event flow occurs. If the periodicity field is set as "yes," then the event flow is evaluated whether it is started at every *Period* ± *Error Bound* or not.

4. Monitoring Architecture

Figure 1 shows the overall architecture of Breadcrumbs, which has the event flow timing fault detection, the timing fault diagnosis support, and the timing fault handling. Breadcrumbs utilizes the timing constraint specification and the source code of the program as inputs and inserts timing fault detection routines into the original code through the event flow path analysis. The Breadcrumbs run-time is compiled via the modified code. When an instance of the modified code executes the event flow, it performs the routines for timing fault detection and saves execution time records for fault diagnosis inside its execution context. When a timing violation is detected, the Breadcrumbs run-time is executed in the fault detected execution context and invokes the fault diagnosis support and the fault handling.

4.1. Event Flow Path Analysis

To insert timing fault detection routines for event flows into target source codes, we need the source

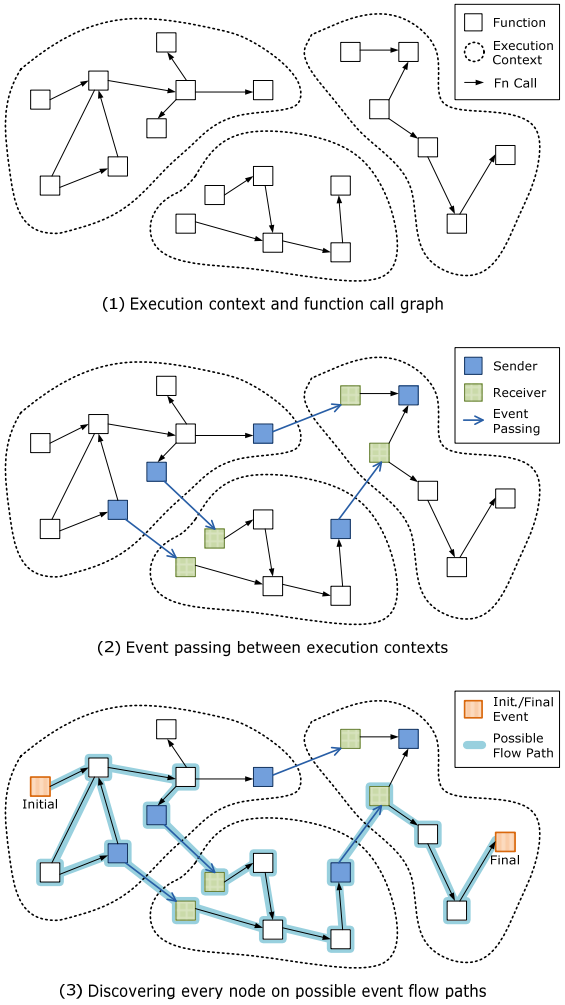

Figure 2. Process of event flow path analysis

code and related timing constraint specifications to analyze the event flow path. Code analysis is conducted as presented in Figure 2: (1) identify execution contexts from source code and construct a function call graph in each execution context; (2) in each execution context, find sender and receiver functions for passing messages to other execution contexts and indicate each message passing between senders and receivers as an edge in the graph; (3) identify every node on possible paths from the initial event to the final event.

There have been several previous efforts in constructing a call graph, as shown in Figure 2.1. We use a context-insensitive call graph for simplicity, and well-known tools, such as GNU cflow [15], can be used to construct this type of call graph. In the case that indirect calls using function pointers exist in the code, a user's explicit description about indirect function calls might be required to complete the call graph. Breadcrumbs constructs a directed edge graph

in which each node is a function. Message senders and receivers in Figure 2.2 include system calls or libraries for IPC in a single system and network message send and receive functions in a distributed system. With a program using a network middleware, like DDS [16], which provides separated send and receive handlers according to message types, more detailed event paths can be identified in this stage. With a program implementing networking with the TCP and UDP socket interfaces, the relationship of the sender and receiver can be identified by port numbers. In Figure 2.3, the procedure of discovering every node on the possible event flow paths uses a simple Breath-First Search. This search is started from the function causing the initial event and identifies every node on the paths to the function causing the final event.

4.2. Timing Fault Detection

Breadcrumbs detects timing violations by transparently embedding timing information into event flow instances. After starting the event flow, the deadline in the timing constraint specification is granted to the event flow instance. Time consumption during the execution of the event flow instance is deducted from the granted time, and this enables the event flow instance to detect the case of the granted time being exhausted, namely deadline expiration, by itself. Breadcrumbs inserts the timing fault checks running this procedure into the original source code.

Using the result of the event flow path analysis, we can identify the position where the timing fault checks are inserted. We need the variables T_{remain}, T_{check}, and *SeqNo* to be declared in each execution context running the event flow, as these variables save the timing information of the currently running event flow instance. T_{remain} saves the remaining time of the instance, while T_{check} saves the system time when T_{remain} is updated. *SeqNo* is the instance's identifier and is used to distinguish it from other instances. If periodicity is checked in the timing constraint specification, T_{next} is declared in the execution context, which raises an initial event; T_{next} records the expected time of the next period start. RAM usage for these variables is as follows: $\{sizeof(T_{remain}) + sizeof(T_{check}) + sizeof(SeqNo)\} \times (\#\ of\ execution\ contexts) + sizeof(T_{next})$.

The timing fault checks are inserted at initial and final event functions, at senders and receivers for a message passing between execution contexts, and at other intermediate functions on the event flow paths. The content of the inserted routine depends on the insert position. To begin with, the routine shown in Figure 3 is inserted at the prologue of the initial event function. *TC_1* is the name of the timing constraint specification, and *CID* is an identifier of the currently running execution context. In this code, the purpose of

```
#define TC_1_Period   100
#define TC_1_ErrorBound 5
#define TC_1_Deadline  20

Initial_Event_Function()
{
    /* Initialize event flow remaining time */
    local var Tcurr = GetCurrentTime();
    TC_1[CID].Tcheck = Tcurr;
    TC_1[CID].Tremain = TC_1_DEADLINE;
    TC_1[CID].SeqNo  = SetUniqueSeqNo();
    /* Check periodicity */
    if (TC_1[CID].Tnext is unavailable) {
        TC_1[CID].Tnext = Tcurr + TC_1_Period;
    }
    else {
        if (TC_1[CID].Tnext - TC_1_ErrorBound < Tcurr
         && Tcurr < TC_1[CID].Tnext + TC_1_ErrorBound)
            TC_1[CID].Tnext += TC_1_Period;
        else
            TC_1_ErrorHandler();
    }
    ...
```

Figure 3. Inserted routine at initial event

```
if (TC_1[CID].Tremain ≠ unavailable) {
    local var Tcurr = GetCurrentTime();
    TC_1[CID].Tremain -= (Tcurr - TC_1[CID].Tcheck);
    TC_1[CID].Tcheck = Tcurr;
}
```

Figure 4. Inserted routine at intermediate functions

```
if (TC_1[CID].Tremain ≠ unavailable) {
    local var Tcurr = GetCurrentTime();
    TC_1[CID].Tremain -= (Tcurr - TC_1[CID].Tcheck);

    if (TC_1[CID].Tremain < 0) TC_1_ErrorHandler();

    TC_1[CID].Tremain = unavailable;
}
```

Figure 5. Inserted routine at final event

TC_1[CID] is to show that each execution context manages their own T_{remain}, T_{check}, and *SeqNo*. The real implementation of *TC_1[CID]* may be the same as in the code or allocate these variables in each execution context's stack area without explicitly using *CID*. At the position of the initial event occurrence, the routine initializes T_{remain}, T_{check}, and *SeqNo* to start the event flow, and it checks periodicity validation if required. To assign a unique *SeqNo* to each event flow instance, our implementation uses a combination of node ID, execution context ID, and local sequence numbers.

At the prologue and epilogue of intermediate functions, except senders and receivers, which perform message passing between execution contexts, we insert the routine in Figure 4. The routine is executed only when T_{remain} is available, and it updates T_{remain} and T_{check}. It is required to write wrapper functions to identify the time consumption of black box functions, such as system-calls and library code, called by intermediate functions. The routine shown in Figure 4 is inserted at the prologue and the epilogue of the wrapper function and the black box function is called between them. The routine in Figure 5 is inserted at the final event. This evaluates the timing validity of the event flow using T_{remain}. In the case of a timing constraint violation, namely T_{remain} is less than zero, the routine executes an error handler, which reports the timing violation and

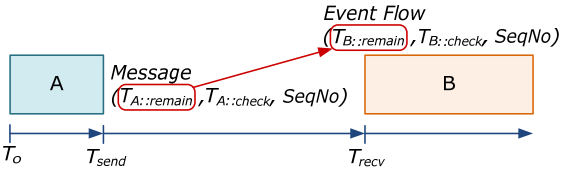

Figure 6. Timing model of event passing between execution contexts

invokes an error handling method described in the timing constraint specification. Timing validation is checked only at the final event, because the timing constraint is specific to the event flow, which requires both the initial event and the final event. Even though the initial event has occurred, the final event may not occur for several reasons, such as a stop processing by data validation check or a packet loss of unreliable network protocols. These reasons of program logic are beyond the scope of timing constraints of event flows. After the validation check, the routine concludes the event flow instance by setting T_{remain} as 'unavailable', which prevents the timing fault detection routines from being executed after the termination of the instance.

Before explaining the timing check at the inter-context message senders and receivers, we denote the timing model. Figure 6 shows a message passing from execution context A to B. T_{send} is the time when A sends a message to B, and T_{recv} is the time when B receives this message. If A and B are threads in the same system, then $T_{recv} - T_{send}$ might be caused by IPC and scheduling. If A and B are located on different nodes in a distributed system, then $T_{recv} - T_{send}$ might be caused by network stack and transmission delay. For B to progress the event flow when it receives a message, it needs to acquire timing information regarding the event flow instance in A, i.e., T_{remain}, T_{check}, and *SeqNo*. We transmit the timing information by carrying T_{remain}, T_{check}, and *SeqNo* in the message itself. This only introduces a small increase in the message size and therefore minimizes monitoring interference in the program execution. In addition, it can be transparently applied to the black box components, such as a network layer or a library code, since they can only perceive an increase in the payload. Here, time synchronization between two adjacent nodes is required to utilize the transmitted timing information. This does not mean a global clock is required; in the case of sensor networks, time differences between adjacent nodes can be solved with little overhead by Elapsed Time on Arrival [17].

Figure 7 and Figure 8 are inserted routines to the sender and the receiver, respectively. At the message sender, Breadcrumbs declares a new message type to encapsulate the original message for the preservation of the original message type. The references to the original message are replaced with the payload field of the new message instance. The timing information is

```
local var Output;

/* References to the original message are replaced with
 * with "Output.Payload".
 * e.g. OrigMsg.memberA -> Output.Payload.memberA */
...
if (TC_1[CID].Tremain ≠ unavailable) {
   local var Tcurr = GetCurrentTime();
   TC_1[CID].Tremain -= (Tcurr - TC_1[CID].Tcheck);

   Output.Tremain = TC_1[CID].Tremain;
   Output.Tcheck  = Tcurr;
   Output.SeqNo   = TC_1[CID].SeqNo;

   TC_1[CID].Tremain; = unavailable;
}
Send(Output);
...
```

Figure 7. Inserted routine at message sender

```
Receive_Handler (Input)
{
   if (Input.Tremain ≠ unavailable) {
      TC_1[CID].SeqNo = Input.SeqNo;
      TC_1[CID].Tcheck = GetCurrentTime();
      TC_1[CID].Tremain = Input.Tremain
                - (TC_1[CID].Tcheck - Input.Tcheck);
   }
   /* References to the original message are replaced
    * with "Output.Payload".
   ...
```

Figure 8. Inserted routine at message receiver

updated right before message transmission and is embedded in the new message. Sending a message to other execution contexts means the termination of the event flow instance in the sender's execution context, so T_{remain} is set to 'unavailable'. The receiver extracts the original message and timing information, i.e., T_{remain}, T_{send}, and *SeqNo,* from the received message. The receiver updates the timing information of the receiver's execution context, and this continues the event flow instance in the receiver's execution context.

4.3. Fault Diagnosis Support

To analyze the cause of a timing violation, it is helpful to identify time consumption in each software module participating in a faulty event flow instance. However, the timing fault detection alone cannot provide a run-time path and the time consumption of each function on the path. To achieve this purpose, we mark identifiers to every inserted timing fault detection routine and allocate a list data structure in each execution context. We then make the inserted routine save its identifier and T_{remain} to the list. Let the routine identifier saved in the n_{th} element of the list be $E_{id}(n)$ and its time $E_t(n)$. The execution time from the first routine to the second routine can be calculated as $E_t(2) - E_t(1)$, and the execution time from the second routine to the third routine is $E_t(3) - E_t(2)$. With the same method, we can identify the run-time path and time consumption of each intermediate function.

In the case that an event flow instance ranges over several execution contexts, the execution record of the instance causing timing violation might be overwritten

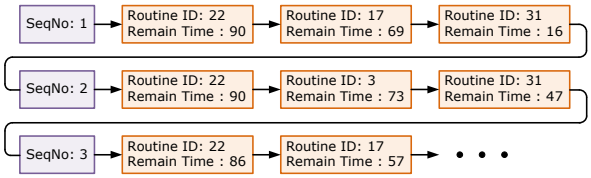

Figure 9. Repository for execution time history

in some execution contexts before the instance reaches its final event. We therefore allocate a repository for execution time histories, and each execution context accumulates the execution time list with the sequence number of the instance in the repository. Figure 9 shows the structure of the repository. The number of records to be saved needs to be determined considering the frequency of event occurrences and the time required to begin fault diagnosis support after the detection of timing violation.

As every execution context shares the sequence number of the event flow instance, execution time histories can be referenced by the sequence number when the timing violation occurs. By examining the execution time histories from every execution context participating in the fault event flow instance, we can identify the time consumption of each intermediate function and node. This may be implemented in different ways in each system. For example, it can be implemented by dumping memory or by sending a query to every execution context.

4.4. Fault Handling

When a timing violation is detected, the inserted routine invokes fault handling according to the timing constraint specification. The optimal method for timing fault handling may differ from the purpose of the timing constraints and the significance of the timing fault, so we provide examples of several strategies for handling timing faults using Breadcrumbs.

In a real-time system or a mission critical system, users may want to safely suspend the subsystem to prevent fault propagation to other subsystems. In this case, the fault handling method could be specified as *halt* or *reboot*, and the system would perform safe termination or reboot after the fault diagnosis support. User written functions could also be specified in the timing constraint specification. In a system where the timing faults only affect the quality of service, user handlers could compensate or disregard a delayed event flow instance.

5. Evaluation

In this section, we first present the details of Breadcrumbs implementation, and then describe evaluation methods and the experimental setup. Finally, we show the practicability of Breadcrumbs by presenting the experimental results.

5.1. Breadcrumbs Implementation

Test platforms: The Breadcrumbs implementation targets distributed real-time applications, written in the C language, using Rational RoseRT 7.0 [14] and RTI DDS 4.1e [16]. RoseRT is a modeling tool based on UML-RT, which provides a real-time application development environment. DDS is a network middleware for distributed real-time systems and has a data-centric publish-subscribe architecture. The target machine is the Curtiss-Wright SVME/DMV-183 single board computer [18] running the Wind River VxWorks 5.5 [19] operating system and is equipped with the PowerPC 7447A 1 GHz processor and 1 GBytes of RAM.

Timing constraint specification: Timing constraint specifications can be written in a text-based script. *Initial Event* and *Final Event* are set to the function or transition name defined in RoseRT, and *Execution Context* is set to the capsule name in RoseRT because the capsule is a schedulable and logically independent application-level execution model in RoseRT. *Node ID* is also set to the name of the top capsule in the RoseRT model and is internally converted into a numeric value by the Breadcrumbs implementation. *Deadline*, *Period*, and *Error Bound* can be specified in millisecond units.

Event flow path analysis: RoseRT provides a compilable code from a UML model and user-written code fragments within it. DDS generates an API code according to message types specified in an Interface Description Language (IDL) file. From the scope of Breadcrumbs' code analysis for a function call graph, we exclude codes automatically generated by RoseRT and DDS, not programmed by users. Message senders and receivers for inter-capsule communication are identified by analyzing the *Ports* and *Protocols* specified in the RoseRT model. Senders and receivers for DDS messages are identified by analyzing publishers and subscribers for the message types specified in the IDL file.

Code insert for timing fault detection: To insert timing information into communication messages between execution contexts, we append the new message types encapsulating the original messages to the IDL file and the RoseRT-generated source files. In obtaining the system time, we set the timer interrupt tick of the target system to 1 msec and used the tickGet() function of VxWorks APIs. Time synchronization between distributed nodes was performed before the experiment.

Fault diagnosis support and handling: We allocate 100 execution time records to be saved in each execution context for the fault diagnosis support. The

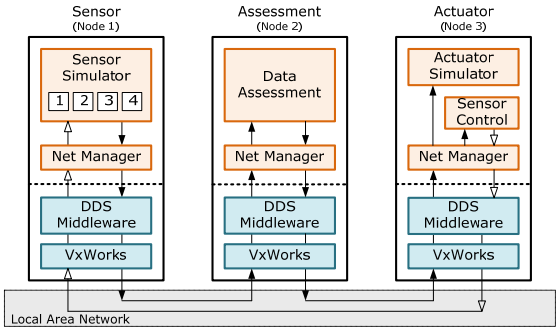

Figure 10. Target system configuration

execution time records are written in order and can be overwritten on the oldest record. When a timing constraint violation occurs, Breadcrumbs informs users of the fault occurrence by sending a notification message, and each node passes its execution time history to users if it requests the execution time records with *SeqNo*. Breadcrumbs provides *halt* and *ignore* as fault handling methods.

5.2. Experimental Setup

The real-time system architecture used in the experiment is presented in Figure 10. This architecture has three distributed nodes, and each node is connected by a 100 Mbps local area network. Messages between nodes are exchanged by DDS. RoseRT applications are installed on each node, and every capsule in each application has an individual physical thread. The role of Node 1 is a sensor. The SensorSimulator capsule has four distinctive simulators. Each simulator generates and periodically updates identifiable data, such as an aircraft track. Sensor data from SensorSimulator are forwarded to Node 2 by the NetManager capsule. The role of Node 2 is assessment; NetManager forwards received sensor data to the DataAssessment capsule and assessment results are then sent to Node 3. In Node 3, NetManager forwards received assessment results to the ActuatorSimulator capsule and the SensorControl capsule. SensorControl can send control messages to the four inner simulators of SensorSimulator in Node 1, according to the contents of the assessment results.

5.3. Timing Fault Detection

We verified whether Breadcrumbs was able to detect timing constraint violations. Table 2 shows timing constraints applied to the test application.

Timing constraint 1: The first timing constraint is applied to the event flow, which starts from the first simulator of SensorSimulator (SensorSim 1) and ends at ActuatorSimulator. The purpose of this timing constraint is to verify that Breadcrumbs accurately

Table 2. Timing constraints for test application

		Timing Constraint 1	*Timing Constraint 2*
Initial Event	***Function***	GenSensorData	OperationOrder
	Capsule	SensorSim1	SensorControl
	Node ID	Sensor (Node 1)	Actuator (Node 3)
Final Event	***Function***	End of ReadData	Start of RecvOrder
	Capsule	ActuatorSimulator	SensorSim1~4
	Node ID	Actuator (Node 3)	Sensor (Node 1)
Deadline		20 msec	10 msec
Fault Handling Method		Halt	Halt
Periodic Event	***Periodicity***	Yes	No
	Period	30 msec	-
	Error Bound	3 msec	-

detects a timing fault of a specific event flow while other event flow instances (SensorSim 2, 3, 4) that have the same intermediate path occur. As part of the experiment, we injected the buggy code that causes a processing time delay of Node 2. Breadcrumbs could correctly detect the deadline expiration of the event flow instance in ActuatorSimulator where the event flow ends. Other delayed event flows started from SensorSim 2, 3, 4 did not affect the fault detection of Timing Constraint 1. We could identify the delay factor through the fault diagnosis support. Figure 11 compares the remaining time on normal and error conditions. We were able to diagnose that the function DSectorComp of Node 2 was the obvious delay factor. Timing Constraint 1 also has a periodic property. To deliberately test the periodicity of Timer Constraint 1, we injected a bug into the periodic timer of SensorSim 1, and we verified that Breadcrumbs was able to detect the periodicity error.

Timing constraint 2: The second timing constraint is applied to an event flow that starts when SensorControl sends a control message and finishes when SensorSim 1–4 receives the command. The purpose of this timing constraint is to verify that Breadcrumbs correctly detects a timing fault when an event flow instance from a single initial event is diverged into multiple final events. First, we injected an error that causes a 10-ms delay only when an order is sent to SensorSim 3. As a result, SensorSim 1, 2, and 4 worked normally, but SensorSim 3 reported a timing violation. Second, we injected an error that causes a 10-ms delay whenever the command is sent to SensorSim 1–4, and Breadcrumbs correctly detected each timing fault of SensorSim 1–4.

5.4. Performance Overhead

In order to identify the overhead of Breadcrumbs, we first evaluate the increased payload of messages caused by Breadcrumbs. In our implementation, the size of both T_{remain} and T_{check} is 4 bytes, and *SeqNo* is 6 bytes. We defined *SeqNo* as the struct {Node ID, Capsule ID, local sequence number in capsule}, and

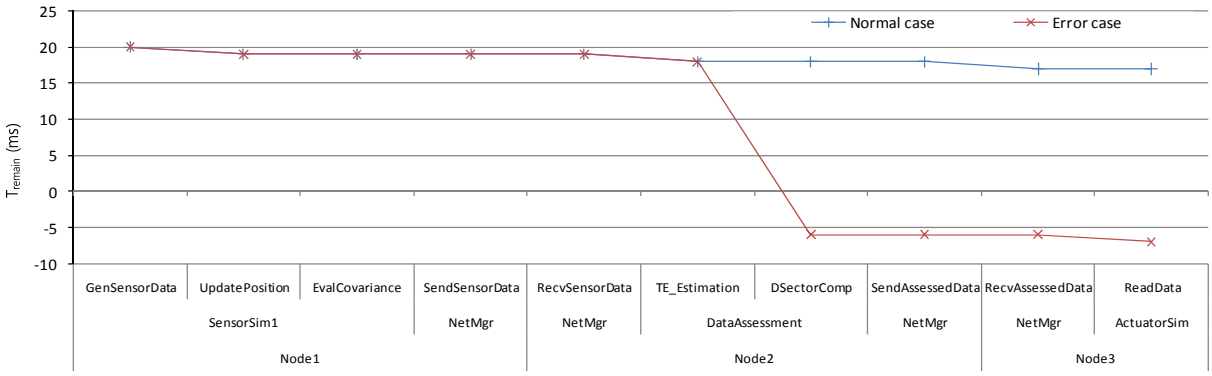

Figure 11. Execution time comparison

Table 3. CPU overhead comparison

	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%

the size of each element is 2 bytes. Therefore, a total of 14 bytes of data is appended to relevant inter-capsule communication messages and inter-node DDS messages. In the test application, the size of sensor data is 160 bytes and the size of assessed data is 200 bytes; therefore, the increased message size overhead caused by the insertion of timing information is about 7–8.75%.

To measure the calculation overhead of the payload increase and the run-time fault detection, we measured the CPU overhead (Table 3). As the count of the sensor samples increased, CPU usage also increased, but the CPU usage gap between the original program and the Breadcrumbs applied program was 0.4% (maximum) in Node 2 when the sample count was 800. In Node 3, both the original program and the Breadcrumbs applied program showed the same CPU usage, except in the case of 500 samples. The CPU usage difference between Node 2 and Node 3 is due to the difference in the number of functions called while running event flow instances. Node 2 performs calculations with a relatively long call depth, but Node 3 completes calculations with a short call depth. The fact that the data payload increased in all nodes but CPU usage of Node 3 barely increased suggests that the payload increase caused by Breadcrumbs produces little calculation overhead.

6. Discussion

The main limitation of Breadcrumbs is that the amount of execution time history to be saved is hard to be determined precisely. The amount needs to be determined by users, and the frequency of event occurrences and the time required to begin fault diagnosis support after the timing violation detection are considered. However, since the execution time history is needed at the fault diagnosis support, the timing fault detection is performed regardless of the amount of execution time history. Users may try to increase the amount when the acquisition of delay factors fails.

Application programs have to be recompiled if a user modifies a timing constraint of an event flow. Though only some of the source code files affected by the event flow need to be recompiled, this could be disadvantage for large-sized programs.

According to the definition of event flow, it is not possible to detect a timing violation until a final event occurs. Therefore, a user may have to split up the event flow when the user wants to react as fast as possible. This also could be a potential disadvantage of using Breadcrumbs.

Breadcrumbs requires time synchronization between two adjacent nodes. However, time synchronization is also required by previous run-time monitoring approaches for distributed systems [21][32], which need global time synchronization. Partial time synchronization required by Breadcrumbs is a subset of global time synchronization. In the case of mobile network, eased assumption of our approach can be an advantage; [17] showed that time synchronization between adjacent nodes can be performed with less overhead than global time synchronization.

7. Conclusion

In this paper, we presented Breadcrumbs for monitoring timing constraints of event flows based on a decentralized approach. Breadcrumbs analyzes possible paths of event flows and inserts timing constraint violation checks for run-time detection. When a timing violation is detected, Breadcrumbs identifies the run-time paths of event flows and the time consumption of each participating software

module; it also executes timing violation handlers according to timing constraint specifications. As Breadcrumbs provides an automatic code manipulation, significant user intervention, such as advance analysis of possible path of event flow and manual code manipulation, is not required. We have implemented Breadcrumbs for real-world systems and the computation overhead of Breadcrumbs was less than 1% in our experiment.

In future, we would like to port Breadcrumbs to diverse real-time distributed systems. Since Breadcrumbs has low CPU overhead and does not transmit additional network packets, we expect that our approach can be applied to the development of real-time wireless sensor network systems.

8. Acknowledgements

We would like to thank Jung Hyung Park for helpful discussions during the early stages of the project. Our thanks also go to the anonymous reviewers and shepherd for the feedbacks. This work was supported by the National Research Foundation (NRF) of the Korean government (grant no.2010-0000405).

9. References

[1] A. K. Mok, G. Liu, "Efficient run-time monitoring of timing constraints," *Proc. of the 3rd IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

[2] S. E. Chodrow, F. Jahanian, M. Donner, "Run-time monitoring of real-time systems," *Proc. of the 12th IEEE Real-Time Systems Symposium (RTSS)*, 1991.

[3] F. Jahanian, A. Goyal, "A formalism for monitoring real-time constraints at run-time," *Proc. of the 20th Symposium on Fault-Tolerant Computing Systems*, 1990.

[4] C. Lee, A. K. Mok, P. Konana, "Monitoring of timing constraints with confidence threshold requirements," *Proc. of the 24th IEEE International Real-Time Systems Symposium (RTSS)*, 2003.

[5] Z. Chen, K. G. Shin, "Post-deployment performance debugging in wireless sensor networks," *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[6] Y. Wang, M. C. Vuran, S. Goddard, "Cross-layer analysis of the end-to-end delay distribution in wireless sensor networks," *Proc. of the 30th IEEE Real-Time Systems Symposium (RTSS)*, 2009.

[7] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, K. Pister, "System architecture directions for network sensors," *Proc. of the 9th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.

[8] M. Khan, H. Le, H. Ahmadi, T. Abdelzaher, J. Han, "Dustminer: troubleshooting interactive complexity bugs in sensor networks," *Proc. of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2008.

[9] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, L. Luo, "Declarative tracepoints: A programmable and application independent debugging system for wireless sensor networks," *Proc. of the 6th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2008.

[10] M. Wachs, J. I. Choi, J. W. Lee, K. Srinivasan, Z. Chen, M. Jain, P. Levis, "Visibility: a new metric for protocol design," *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.

[11] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," *Proc. of the 19th ACM Symposium on Operating Systems Principles*, 2003.

[12] W. Archer, P. Levis, J. Regehr, "Interface contracts for tinyos," *Proc. of the 6th International Conference on Information Processing in Sensor Networks (IPSN)*, 2007.

[13] Microsoft Windows Message Loop, http://msdn.microsoft.com/en-us/library/ms644928(VS.85).aspx

[14] IBM Rational Rose Realtime, http://www.ibm.com/

[15] GNU cflow, http://www.gnu.org/software/cflow/

[16] RTI Data Distribution Service, http://www.rti.com/

[17] B. Kusy, P. Dutta, P. Levis, M. Maroti, A. Ledeczi, D. Culler, "Elapsed time on arrival: a simple and versatile primitive for canonical time synchronisation services," *Journal of Ad Hoc Ubiquitous Computing,* Vol.1, Issue 4, 2006.

[18] Curtiss-Wright SVME/DMV-183 Dual Freescale 7447A/7448, http://www.cwcembedded.com/

[19] Wind River Systems Inc., http://www.windriver.com/

[20] P.S. Dodd, C.V. Ravishankar, "Monitoring and debugging distributed real-time programs," *Software–Practice and Experience*, Vol.22, No.10, Oct. 1992.

[21] S. Raju, R. Rajkumar, "Monitoring timing constraints in distributed real time systems", *Proc. of the 13th IEEE Real-Time Systems Symposium(RTSS)*, 1992.

[22] H. Tokuda, M. Kotera, C.W. Mercer, "A real-time monitor for a distributed real-time operating system," *Proc. of the ACM Workshop Parallel and Distributed Debugging*, 1988.

[23] B. Plattner, "Real-time execution monitoring," *IEEE Transactions on Software Engineering*, Vol.SE-10, No.6, 1984.

[24] J.J.P. Tsai, K.-Y. Fang, H.-Y. Chen, "A noninvasive architecture to monitor real-time distributed systems," *Computer*, Vol.23, No.3, 1990.

[25] John D. Valois, "Lock-free linked lists using compare-and-swap," *Proc. of the 14th Annual ACM Symposium on Principles of Distributed Computing*, 1995.

[26] V. Shnayder, M. Hempstead, B. Chen, G. W. Allen, M. Welsh, "Simulating the power consumption of large-scale sensor network applications," *Proc. of the 2nd ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.

[27] P. Levis, N. Lee, "Tossim: Accurate and scalable simulation of entire tinyos applications," *Proc. of the 1st ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.

[28] B. Titzer, D. Lee, J. Palsberg, "Avrora: scalable sensor network simulation with precise timing," *Proc. of the 4th International Conference on Information Processing in Sensor Networks (IPSN)*, 2005.

[29] G. Tolle, D. Culler, "Design of an application cooperative management system for wireless sensor networks," *Proc. of the 2nd European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[30] K. Romer, J. Ma, "PDA: passive distributed assertions for sensor networks," *Proc. of the 8th Information Processing in Sensor Networks (IPSN)*, 2009.

[31] P. Li, J. Regehr, "T-check: bug finding for sensor networks," *Proc. of the 9th International Conference on Information Processing in Sensor Networks (IPSN),*2010.

[32] F. Jahanian, R. Rajkumar , S. C. V. Raju, "Runtime monitoring of timing constraints in distributed real-time systems," *Real-Time Systems* 7(3):247-273, 1994.