# Towards Responsive Context-Aware Environments

Marco Gruteser

IBM T. J. Watson Research Center, Yorktown Heights, NY 10598
and
Department of Computer Science, University of Colorado at Boulder,
Boulder, CO 80309
gruteser@cs.colorado.edu

**Abstract.** System responsiveness is critical in context-aware computing systems because of their tight coupling with the physical environment. To initiate actions quickly, applications depend on the timely delivery of context-events. However, delivery times are difficult to predict because of ever-changing system configurations. This paper presents a context-aware system that optimizes the timeliness of context-events according to application-specified requirements. To this end, it schedules periodic activations of sensor components and controls event propagation through aggregators and applications. First results indicate that the scheduler improves the miss ratio on a smart space prototype by approximately half.

## 1 Introduction

Context-aware applications aim at adapting their behavior to situation-dependent user needs. Applications typically build on an infrastructure of interpreter and aggregator components, which derive contextual information from sensor clues [11]. Changes in the contextual information are signaled through events. Although producers of contextual information and applications are conceptually independent, the applications rely on the producers to comply with Quality of Service (QoS) requirements, such as timeliness, frequency, granularity, and accuracy of sensed contextual data.

Meeting QoS requirements in ubiquitous computing systems is challenging, because systems have to adapt to available software-components and hardware resources [1, 2]. Systems should spontaneously integrate or remove devices and application components. Consider a smart space environment [3, 4, 5, 6] with an extensive context-awareness infrastructure comprised of sensors and interpreter and aggregator components. Especially, in office hotelling environments, where the owner of an office can change on a daily basis, the smart space needs to support a diverse set of personal applications and mobile devices.

In previous work, real-time system constraints have been incorporated in blackboard-based intelligent control systems [7, 8, 9]. These systems focus on reasoning and planning with predictable delays. For distributed systems, Blair *et al*

describe an architecture for QoS support in a tuplespace model [10]. It emphasizes adaptation to a changing environment through a specialized set of agents.

In this paper, we focus on adapting a context-aware infrastructure to meet application timeliness requirements for the delivery of sensed data. The key contributions are:

- a system that monitors available components and their communication relationships using an event-flow graph,
- a scheduler that controls the activation of context-components to improve the timeliness of context-events based on application-specified end-to-end timeliness constraints,
- an evaluation of the scheduling approach on a smart space implementation.

The remainder of the paper is structured as follows. Section 2 motivates the system with scenarios from the BlueSpace smart office prototype [6]. Section 3 presents a scheduling algorithm that adapts the context-aware infrastructure to end-to-end application timeliness requirements. Section 4 describes an implementation based on a blackboard-style publish/subscribe system. Finally, Sect. 5 presents an experimental evaluation of the scheduler on the BlueSpace prototype.

## 2 BlueSpace: A Motivating Example

The BlueSpace project develops a smart workspace for knowledge workers. It aims at improving worker comfort and productivity through an adaptive environment, which regulates climate and lighting, supports spontaneous collaborative work sessions, and protects from unwanted interruptions. To this end, BlueSpace combines a flexible physical workspace design with a sensor and software infrastructure supporting context-awareness and a heterogeneous set of computer-controlled output devices.

Applications execute on a central context server and communicate events through a blackboard service. Instead of receiving sensor data directly, applications typically receive aggregated or interpreted information from infrastructure components [11]. Sensors components periodically poll a sensor and update the value on the blackboard, whenever it has changed. Applications and intermediary infrastructure components can subscribe to this information.

The event-flow graph in Fig. 1 shows a set of sample BlueSpace applications and their dependencies on infrastructure components. Each arrow represents a publisher/subscriber relationship. Application components and user interfaces are represented through boxes, and infrastructure components through ovals. The components' functions are as follows. The InfoPanel UI user interface gives an overview of the workspace state and allows manual control of workspace functions. The MyTeam application presents the availability of team members at a glance, similar to an instant messenger. The Light Control, Display Control, and Temperature Control applications adjust the workspace environment according to user preferences. The Presence Status, Occupant Status, and sensor components (Temperature Sensor and Active Badge Sensor) comprise the

infrastructure components. Presence Status identifies workers by name based on the badge ID it receives from the Active Badge Sensor. It also classifies them as occupants or visitors in the workspace. Finally, Occupant Status tracks occupant availability for the MyTeam application.

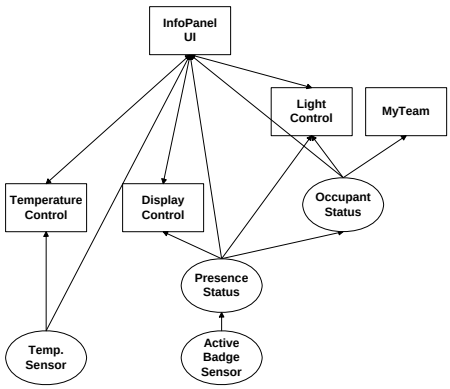


**Fig. 1.** BlueSpace event-flow diagram

Notice that these applications, even when receiving the same event, have very diverse response time requirements. For example, quick response times, say 0.1 seconds, are required for switching the ceiling light on when the occupant enters. Sending availability updates to team members is less urgent. However, both the Light Control and Occupant Status components subscribe to the same event from Presence Status.

Furthermore, consider how timeliness requirements on the infrastructure are situation-dependent. Most of the applications are designed to support the office worker; hence, there is little need to execute them when the workspace is empty. Instead, the workspace can poll the Active Badge Sensor more frequently and prioritize the workspace initialization components, especially light control. This ensures a timely workspace reaction (adjust lighting, wake up displays) when the occupant arrives. Once the worker is in the office, other workspace functions have higher timeliness requirements. Thus, the system can shift its attention from the Active Badge Sensor to more important components.

Statically tuning the system becomes more time consuming and difficult as complexity increases (in terms of the number of components and the number of dependencies). An adaptive system, however, can automatically adjust the system to the application requirements.

## 3 Scheduling Algorithm

Processor scheduling can optimize the timeliness of event delivery. First, obtaining sensor readings more frequently decreases detection time of an event. Furthermore, the scheduler controls the sequence of component activations. Thus, the scheduler can order the activation of components according to their timeliness requirements.

### 3.1 System Model and Design Considerations

The system behavior is implemented through a set of *agents* that communicate through a blackboard. Agents are all components that publish events or subscribe to events through the blackboard. Agents are activated either through *event activations* or *periodic activations*, or both. For example, the Temperature Control component is activated only when it receives an event from the user interface or the Temperature Sensor. The Temperature Sensor is periodically activated.

Furthermore, we distinguish between agents exhibiting an *application role* and an *infrastructure role* and call them *application agents* and *infrastructure agents*, respectively. Application agents initiate externally visible actions, such as controlling lighting, controlling heating, or presenting a graphic user interface. Hence, they typically specify timeliness constraints for the delivery of contextual events. Infrastructure agents produce information intended to support application agents. Thus, their timeliness requirements depend on the supported application agents. Examples are the Presence Status or Occupant Status agents in BlueSpace. An agent may exhibit both roles.

*Timeliness constraints* for events are specified with event subscriptions, thus each agent can specify different constraints for different types of events. The constraint represent an upper bound on the desired delay for events. We measure the *delay* of an agent activation as the total time between occurrence of an external event and the event activation of the agent. Notice that delay also includes the time until the system first detects an external event. Since most events are still useful after the deadline, we interpret the timeliness constraints as *soft deadlines*, that is late events are not discarded. Additionally, subscriptions and timeliness constraints can change during run-time.

### 3.2 Goal

The goal of the scheduler is to execute event and periodically activated agents according to a schedule that meets the application timeliness constraints on a *best effort* basis. Both late activations and high tardiness among the late notifications are undesirable. More formally, we seek to minimize *miss ratio* and *average tardiness*, where miss ratio is the ratio of application agents activated after their timeliness constraint to total application agent activations. Tardiness is defined as follows:

$$tardiness = \begin{cases} d - c \text{ if } d > c \\ 0 \quad\quad \text{if } d \leq c \end{cases}$$

where $d$ is the delay of an application agent activation and $c$ is the timeliness constraint specified with the corresponding event subscription.

### 3.3 Assumptions

The scheduler assumes that agents are non-preemptive and the agents' execution times as well as the event-flow graph are known. The *event-flow graph* describes the dependencies between agents. It contains a node for each agent and an edge between agents if the destination agent subscribes to events that are published by the source agent. More formally, the *subscription relation* $S \subseteq (A \times A)$, where $A$ is the set of all agents, is defined as follows. Given $(a_n, a_m) \in (A \times A)$, then $(a_n, a_m) \in S$ iff $a_n$ publishes events of type $t$ on the blackboard and $a_m$ subscribes to events of type $t$. The *event-flow graph* $G$ is then a directed acyclic graph defined by $G = (A, S)$.

### 3.4 Approach

The solution is based on an online scheduling algorithm, since external events are not known in advance. However, the scheduler pre-computes relative deadlines for event and periodic activations based on the current configuration. To this end, it applies the minimum effective deadline algorithm, which is detailed in Sect. 3.5, to compute deadlines from the timeliness constraints on the event-flow graph. Thus, the scheduler can adapt the system to different situations and timeliness constraints through updating the internal deadlines.

During run-time, the priority-driven scheduler selects agents according to the following policy. Periodically activated agents receive a lower priority than event activated agents, because the polled sensor values rarely change. Resources are typically better spent on processing already detected events. If two agents are both periodically or event activated, we assign priorities according to the *earliest-deadline-first* (EDF) policy [12].

### 3.5 Minimum Effective Deadline Algorithm for Deadline Assignment

The algorithm transforms end-to-end application timeliness constraints into relative deadlines for activations of infrastructure agents. It is based on the following heuristic: the deadline for activations of an agent $a$ is the minimum deadline of it's parents minus the execution time of $a$. If the agent specified a timeliness constraint smaller than the derived deadline, this constraint overrides the computed deadline.

More formally, consider an agent with maximum execution time $t_{exe}$, $n$ subscriptions to other agents, and $m$ subscriptions from others to itself. The agent

can specify any of the timeliness constraints $s_1, \ldots, s_n$ for its subscriptions. Additionally, every subscriber to this agent can specify a timeliness constraint, which we label as the parent constraints $p_1, \ldots, p_m$. For each subscription $i \in \{1, \ldots, n\}$ we compute a new deadline $d_i$ as follows:

$$d_i = \min\{\min\{p_1, \ldots, p_m\} - t_{exe}, s_i\}$$

If the agent is periodically activated, the period is defined as follows:

$$period = \min\{p_1, \ldots, p_m\} - t_{exe}$$

The deadlines are computed for each agent during a breadth-first traversal of the event flow graph. Traversal begins with all components, which are not target of any subscriptions. A node in the graph is visited only after all parents have been visited. Upon termination, this algorithm returns a mapping of edges in the event flow graph to deadlines.

## 4 Scheduler Implementation

Architecturally, the scheduling functionality is implemented in a layer between the agents and a backboard. This configuration enables the scheduler to interact with an existing blackboard implementation, assuming that agents use the new scheduler interface. Alternatively, the scheduler can be integrated into the blackboard itself. Figure 2 depicts the layered approach as well as the dataflow between more detailed components in these layers. Components in the left half of the diagram manage configuration changes such as adding or removing subscriptions. Components in the right half process individual events.

The blackboard maintains a *Subscription Table* and provides the *Subscription Evaluation* functionality. This component uses the Subscription Table to determine which agents subscribe to a particular event. The scheduler adds the following components onto the basic blackboard. First, *Deadline Assignment* calculates deadlines for individual agents from the end-to-end timeliness constraints. To this end, it uses publisher and subscriber information. It stores the result in the *Agent Deadline Table*. Second, *Deadline Lookup* intercepts notifications from the blackboard and assigns absolute deadlines from the relative deadlines in the Agent Deadline Table. All notification records are placed into the *Released Queue*. Finally, the *EDF-Ordering* component selects and invokes agents according to the EDF-based policy.

### 4.1 Specifying Timeliness Constraints

The scheduler extends the basic blackboard interface in the following ways:

- Publishers have to register event types before they can publish. This information is used to construct an event-flow graph.
- Subscribers specify a timeliness constraint for each subscription. Timeliness constraints are given in milliseconds according to the definition in Sect. 3.1.

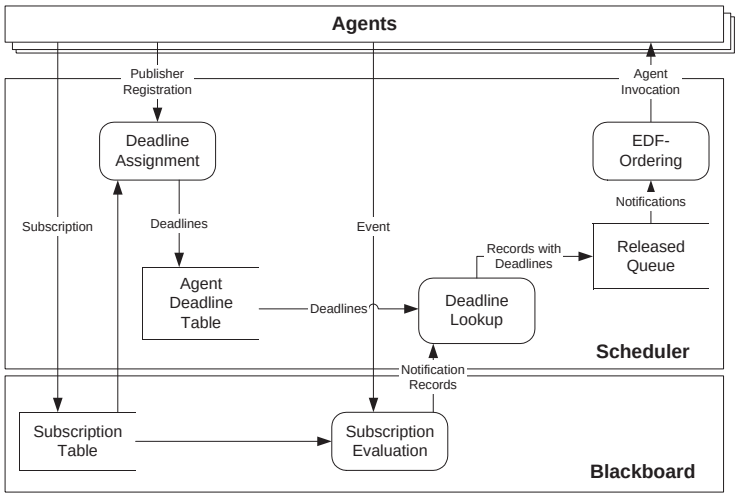


**Fig. 2.** Scheduler dataflow diagram

– Agents can also request periodic invocations. If no period is specified, the scheduler automatically determines a period based on timeliness constraints from subscribers.

The basic blackboard implementation is XML-based. Events are represented as Document Object Model (DOM) Objects and subscriptions are expressed through XPath expressions. If the XPath expression matches a node in a published event document, the event is sent to the corresponding subscriber.

### 4.2 Constructing the Event-flow Graph

The scheduling algorithm requires knowledge of the event-flow graph. The implementation generates this graph from the subscriptions and publisher registrations in two steps. First, it creates a node for every component in the system. Second, it evaluates all the subscriptions against event types published by other components. If a subscription matches an event type, an edge is inserted between the corresponding nodes in the graph.

The matching process is straightforward if subscriptions are only based on well-known event types. However, some blackboard systems offer subscriptions with sophisticated filter expressions. In this case, it depends on the event information at run-time whether a notification occurs. We propose a conservative approach to omitting edges, for this case. We only omit an edge if we can statically determine that the subscriber will never receive events from this publisher.

## 5 Experimentation

The scheduler is tested with the actual BlueSpace software. It is comprised of seven infrastructure agents and six application agents. Among them, five infrastructure agents are periodically activated to obtain sensor readings and two agents interpret sensor information. Timeliness constraints ranging from 50 to 400 ms are specified for 13 out of the 19 edges in the event-flow graph. The software, however, is disconnected from the sensor and actuator hardware to enable many repetitions of trials. The sensor modules were replaced with stubs that generate events according to the selected workload.

One trial consists of 12 bursts of 5 events (one for each sensor input) that are randomly distributed over a 60 second period. A *bursty* workload more accurately reflects the event patterns in a smart space than a constant workload. Each trial is repeated 15 times.

A simple FIFO scheduling policy serves as a baseline. In this case, all periodic agents are kept in a queue. The scheduler always selects the first element of the queue for execution. After execution is complete, a periodic agent is moved from the front of the queue to the end, whereas an event activated agent is removed from the queue until it receives the next event notification.

The scheduler logs the invocation time of each agent that specifies a timeliness constraint. The difference between invocation time and time of event origin yields the delay, from which we can determine misses and tardiness.

### 5.1 Results

Figure 3 presents a scatter plot, where miss ratio is plotted against average tardiness. Each point represents one trial of our experiment. Apart from three outliers with a miss ratio over 0.35, the EDF-based scheduler approximately reduces the miss ratio by half compared to the FIFO baseline. At the same time, the average tardiness stays comparable.

These first results suggest that scheduling effectively reduces the miss ratio. However, the miss ratio remains around 20%, due to the following reasons. First, we deliberately stressed the system by triggering multiple events simultaneously. Second, we noticed that the Java garbage collector has a significant non-deterministic influence on the results. For a full garbage collection, the system is paused for approximately 200ms.

## 6 Summary and Future Work

This paper presented first steps towards a self-tuning context infrastructure. The system seeks to improve responsiveness through timely delivery of context-events. A scheduling mechanism monitors the communication relationships between agents and schedules their activation to meet application-specified end-to-end timeliness constraints. The scheduler was evaluated on a subset of the BlueSpace smart space prototype. First results suggest that scheduling has a

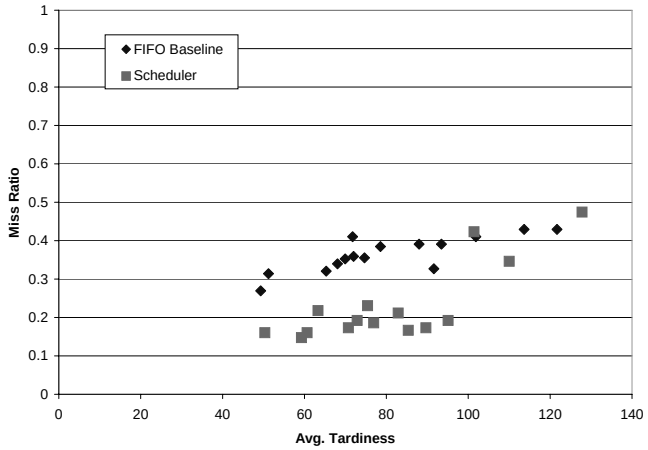


**Fig. 3.** Avg Tardiness vs. Miss Ratio scatter plot for bursty workloads

significant impact. The scheduler reduced the miss ratio on challenging workloads by approximately half.

In future work, we plan to provide more controlled degradation of service quality when the system is overloaded. Supporting other QoS attributes in the system poses additional challenges.

## Acknowledgements

I am grateful to Dirk Grunwald and Rick Han for their invaluable advice and encouragement. I am also indebted to Paul Chou for his willingess to provide great feedback on early drafts of this paper. Furthermore, I thank the other members of the BlueSpace team for designing and building the prototype that motivated this work.

## References

[1] D. Garlan and B. Schmerl. Component-based software engineering in pervasive computing environments. In *4th ICSE Workshop on Component-Based Software Engineering*, 2001.

[2] C. Herring and S. Kaplan. Component-based software systems for smart environments. *IEEE Personal Communications*, 7(4):60–61, 2000.

[3] Barry Brumitt, Brian Meyers, John Krumm, Amanda Kern, and Steven A. Shafer. Easyliving: Technologies for intelligent environments. In *HUC*, pages 12–29, 2000.

[4] Armando Fox, Brad Johanson, Pat Hanrahan, and Terry Winograd. Integrating information appliances into an interactive workspace. *IEEE Computer Graphics & Applications*, 20(3), 2000.

[5] H. Gellersen, M. Beigl, and A. Schmidt. Sensor-based context-awareness for situated computing. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[6] P. Chou, M. Gruteser, J. Lai, A. Levas, S. McFaddin, C. Pinhanez, and M. Viveros. Bluespace: Creating a personalized and context-aware workspace. Technical Report RC 22281, IBM Research, 2001.

[7] Philippe Lalanda, Francois Charpillet, and Jean Paul Haton. A real time blackboard based architecture. In *European Conference on Artificial Intelligence*, pages 262–266, 1992.

[8] F. F. Ingrand, M. P. Georgeff, and A. S. Rao. An architecture for real-time reasoning and system control. *IEEE Expert*, 7(6):34–44, 1992.

[9] B. Hayes-Roth. A blackboard architecture for control. *Artificial Intelligence*, 26:251–321, 1985.

[10] G.S. Blair, N. Davies, A. Friday, and S.P. Wade. Quality of service support in a mobile environment: An approach based on tuple spaces. In *Proc. 5 th International Workshop on Quality of Service*, 1997.

[11] Daniel Salber, Anind K. Dey, and Gregory D. Abowd. The context toolkit: Aiding the development of context-enabled applications. In *CHI*, pages 434–441, 1999.

[12] J. Liu. *Real-time Systems.* Prentice Hall, 2000.