Hazard Avoidance in Wireless Sensor and Actor Networks

Ramanuja Vedantham, Zhenyun Zhuang and Raghupathy Sivakumar
School of Electrical and Computer Engineering
Georgia Institute of Technology, Atlanta, USA
{ramv,zhenyun,siva}@ece.gatech.edu

***Abstract*— A typical wireless sensor network performs only one action: sensing the environment. The requirement for intelligent interaction with the environment has led to the emergence of Wireless Sensor and Actor Networks (WSANs). In WSANs, the sensors monitor the environment based on which the sink issues commands to the actors to act on the environment.**

In order to provide tight coupling between sensing and acting, an effective coordination mechanism is required among sensors and actors. In this context, we identify the problem of "*hazards*", which is the out-of-order execution of queries and commands due to a lack of coordination between sensors and actors. We identify three types of hazards and show with an example application, the undesirable consequences of these hazards. We also identify and enumerate the associated challenges in addressing hazards. In this context, we discuss the basic design needed to address this problem efficiently. We propose a distributed and fully localized *hazard-free* approach that addresses the problem and the associated challenges based on the design. Through simulations we study the performance of the proposed solution and two basic strategies, and show that the proposed solution is efficient for a variety of network conditions.

I. Introduction

Wireless Sensor Networks (WSNs) have a wide variety of applications in civilian, medical and military applications. However, the nodes in such a network are limited to one type of action: *sensing the environment*. With increasing requirements for intelligent interaction with the environment, there is a need to not only perceive but also control the monitored environment. This has lead to the emergence of a new class of networks capable of performing both sensing and acting on the environment, which we refer to as Wireless Sensor and Actor[1] Networks (WSAN).

The architecture for a WSAN can be seen as an extension of a wireless sensor network where the *sensors* collect information of the environment and report it to the *sink*, which processes the data and issues commands to the *actors* to act on the environment [1]. The evolution from WSNs, which can be thought of to perform only *read* operations, to WSANs, which can perform both *read and write* operations, introduces unique and new challenges that need to be addressed. In this paper, we address one such challenge. Consider a simple example of a WSAN that uses fire-detector sensors along with water-sprinkler actors. Assume that the sink has issued a *command* directive[2] C to the actors to sprinkle water in response to sensor feedback about a fire. Now, after a certain period of time t, consider the sink to issue a *query* directive Q to check if the fire has been extinguished. If, for a certain region in the WSAN, Q is delivered and executed *before* C by the network, the corresponding response by the sensors - that the fire still exists - will trigger an unnecessary reaction by the sink in the form of more directives to the actors to sprinkle more water. The execution of directives in an order different from what the sink intended it to be has thus resulted in an undesired outcome. While the undesired outcome in the above example is merely the wastage of water, depending upon the nature of the application, such outcomes can even be catastrophic (e.g. poison gas actors where one dose of the gas merely invalidates subject, but two doses can kill).

We refer to such problems where the execution order of directives is different from what the sink intends or expects it to be as *directive hazards*[3]. For brevity, we refer to the problem as simply *hazards* in the rest of the paper. Essentially, the inherent dependency between the actions performed by the sensors, and those performed by the actors, imposes a need for the sink to have control over the order in which directives will be executed, to ensure correctness of operations.

In developing solutions to avoiding hazards, two additional challenges need to be addressed: (i) The rate of execution of the directives, the *directives execution throughput*, has to be maximized in order to serve applications that are real-time. Note that most WSAN applications are likely to have real-time requirements. Revisiting the example introduced earlier, the reporting of the fire and the turning on of the sprinklers have to be done as quickly as possible to ensure effectiveness of the WSAN's application. Hence, a simplistic solution to address hazards that involves the sink waiting to hear confirmation of the previous directive execution from all nodes in the WSAN will be clearly undesirable. (ii) The solution, while avoiding hazards and maximizing directive-execution throughput, should also be designed with consideration to the conservation of communication and energy resources [2] at

[1]We refer to entities that can act on the phenomenon as *actors*. They are sometimes referred to as *actuators* in related literature.

[2]We use the term *directives* to generically refer to both commands and queries.

[3]The inspiration for the term comes from the data hazards problem when pipelining instructions in a CPU.

the WSAN nodes[4]. Specifically, the solution must incur low communication and resource overheads. In this context, we make the following contributions in this paper:

- We first identify the different types of hazards possible in a WSAN, and outline the associated challenges in developing a solution to avoid them.
- We then present a distributed and localized approach called the Neighborhood Clock Approach that addresses the hazards and the associated challenges efficiently, and compare its performance against that of two basic strategies using simulations.

The rest of the paper is organized as follows: Section II illustrates the architecture for WSAN and identifies the problem setting with an example. Section III identifies the hazards and describes the associated challenges and goals. Section IV presents the design that needs to be leveraged for hazard free operation. Section V presents a distributed and fully localized hazard avoidance approach that realizes the basic design. Section VI evaluates the performance of the proposed approach with two basic strategies for a variety of network conditions. Section VII discusses related works and Section VIII concludes the paper.

II. Model and Example Application

A. Model

In this paper, we consider an architectural model, where there is a sink to help in the coordination of sensors and actors. This is an extension of the existing architecture for wireless sensor networks, where the sink serves as the coordination entity and issues directives to both sensors and actors. For the above model, we focus on a generic class of applications, where there are *regional events occurring requiring several iterations of directives*.

Given the above architectural model and class of applications, we now present an example application to explain the problem considered in this work.

B. Example Application

Consider an automated fire extinguisher system for a large, greenhouse garden application. Let this system be equipped with two types of sensors and two types of actors. The first type of sensor is a temperature sensor that monitors the average temperature in its sensing range and uses this as a trigger to determine the presence of fire. The second type is a humidity sensor, which determines the moisture content in the air. The first of the two types of actors is a sprinkler, which sprinkles water when there is a fire within its acting range, while the second is a fan which removes moisture from the air. One of the goals is to douse any outbreak of fire by activating the sprinklers, without flooding the environment. The second goal is to maintain the moisture level in the air below a certain threshold value by activating the fans. Consider the set of directives shown in Figure 1 issued for this fire extinguisher system in the greenhouse garden application. We will use this set of directives to explain the hazards in WSANs.

Variables

T_c: Threshold value for detecting fire,
H_c: Threshold value for activating fans,
R_D: Region of interest for the directive,
T: Average temperature of region R_D,
H: Average humidity of region R_D

Directives

1	Q:	Report the average temperature in Region R_D
		If $(T > T_C)$
2	C:	Activate sprinklers in region R_D for x seconds
3	C:	Activate fans in region R_D for y seconds
4	Q:	Report the average humidity in Region R_D
		If $(H > H_c)$
5	C:	Activate fans in region R_D for z seconds

Fig. 1. Set of Directives for the Fire Extinguisher System

III. The Problem: Hazards in WSANs

A *hazard* is the *out-of-order execution of directives due to a lack of coordination between sensors, actors and the sink* that can potentially lead to undesirable changes in the physical environment. The underlying reasons for an out-of-order execution of a directive, even when they are issued in the correct sequence by the sink, are mentioned below:

- *Different Path lengths:* For sensors and actors that are randomly located in the even region, the paths taken and hence the path length differ. Even for a single node, it is possible that the paths taken by different directives are different. The key reasons for this is when the network is dynamic, either due to mobility or node failures, or, when the network or sink performs explicit load balancing. Thus the differing paths taken for delivery of a directive is the first source of hazard.
- *Different Latencies:* If we consider a single node, even if the path from the sink to the node is fixed, it is possible that the delay for two different directives is not the same. This is because the loss pattern for the delivery of the first directive may not be the same for a subsequent directive. If we consider the case, where the second directive does not incur any losses along the path, whereas the first directive has several losses and associated retransmissions, it is likely that the second directive arrives before the first.

A. CAC Hazard

Command-after-command hazard happens when the order of two sequential commands can not be guaranteed. If two sequential commands, $Command1$ and $Command2$, are issued to two different actors in the event region[5], a CAC hazard occurs if $Command2$ gets executed prior to $Command1$. Consider the fire extinguisher system mentioned in Section II. The directives issued to this system are shown in Figure 1.

[4]We use the terms, node and entity, interchangeably to refer to both sensors and actors.

[5]This definition for all three hazards can be refined further in terms the region in which the nodes need to coordinate, as we will see in Section IV.

Now, let us focus on directives $1-3$ in Figure 1. Consider the case, when $T > T_c$, i.e., there is fire in region R_D. Consider the case when directive 3 arrives before directive 2, that is, the directive for activation of fans (through command path $Command1$ in Figure 2) arrives before the directive for activation of sprinklers (through command path $Command$ 2 in Figure 2). In such a case, the fire would have spread uncontrollably to a large region (because of the fans assisting the spread of fire), before the directive 2 for activation of sprinklers arrives via path $Command2$. This may lead to undesirable results as the fire would have spread to regions beyond region R_D, and in region R_D, the order of execution of commands is reversed possibly resulting in flooding.

CAC Hazard: Consider a set of n directives, I_1,I_2,...,I_n. Let I_k and I_{k+1} be two dependent, sequential commands sent to two actors in the event region, A_x and A_y. Let $E_{k,x}$ denote the execution of the command I_k by actor A_x finishing at time, T_1, and $E_{k+1,y}$ denote the execution of command I_{k+1} by actor A_y starting at time T_2. A CAC hazard occurs when: $T_2 < T_1$

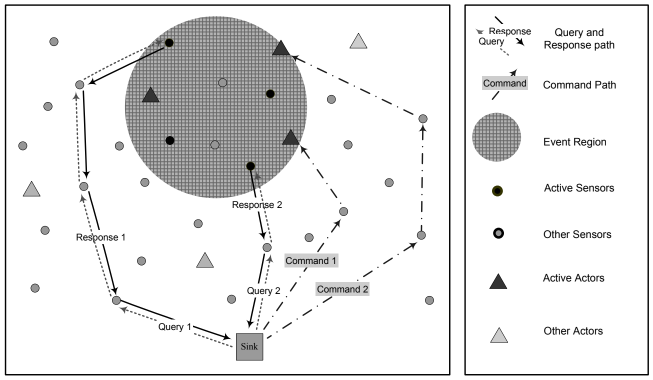


Fig. 2. Hazard Illustration

B. QAC Hazard

Query-after-command hazard occurs when a query issued after a command is executed prior to the execution of the command. Consider the fire extinguisher example and the set of directives as described in Figure 1. Let us focus on the directives $3-5$ in this figure. Consider the case, where after a fire had been detected, the sink ordered the sprinklers and later the fans to deal with this event (directives $1-3$ in Figure 1). Now, in order to check the relative humidity in the region, R_D, the sink sends the directive 4. Consider the case when the directive 4 reaches the sensors (via path $Query2$ in Figure 2) before directive 3, which was received via the command path $Command1$ in Figure 2, was executed. This corresponds to the case, when the humidity sensors receive the query before the fans have acted on the environment. So, the sensors respond to the query indicating that the humidity level is still high (because of previous action by the sprinklers). Now, consider the case when the action is completed just after the response has been sent. When the response arrives at the sink, the sink may draw the wrong conclusion that the humidity level is still high and issues a duplicate command (dir 5) to decrease the humidity in the region resulting in duplicate actions being performed.

QAC Hazard: Let I_k and I_{k+1} be two related, sequential directives, with I_k being a command sent to an event region containing an actor, A_x and I_{k+1} being a query sent to the event region comprising a sensor, S_y. Let $E_{k,x}$ denote the execution of the command I_k by the actor S_x completing at time, T_1 and $R_{k+1,y}$ denote the response to query I_{k+1} by sensor S_y initiated at time, T_2. A QAC hazard is said to have occurred when: $T_2 < T_1$

C. CAQ Hazard

Command-after-query hazard is the opposite of a QAC hazard and happens when a command issued later than a query is executed prior to the query. In the same fire extinguisher application, suppose a query is sent to the temperature sensors to detect the presence of fire (directive 1 in Figure 1). Two responses which contains the same fire information arrive at the sink at different times due to differing delays in the response path. The first response comes through response path 2 ($Response2$ in Figure 2) based on which the sink knows there is fire and sends out directive 2 to the sprinklers to extinguish the fire. This command reaches the actor with a short delay via command path 1 ($Command1$ in Figure 2). After that, the other response for the directive 1 comes through path 1 ($Response1$ in Figure 2) indicating the presence of fire. In this case, the sink will not be able to distinguish whether the response for the directive 1 is initiated before the directive 2 has been executed or after it has been executed. The sink may arrive at the wrong conclusion that the fire is still present and issue another command to extinguish the fire, which may lead to duplicate issue of commands to sprinklers leading to flooding in that region.

CAQ Hazard: Let I_k and I_{k+1} be two related, sequential directives, with I_k being a query sent to the event region consisting of sensor, S_x, and I_{k+1} being a command sent to the event region consisting of actor, A_y. Let $R_{k,x}$ denote the response to query I_k by sensor S_x initiated at time, T_1 and $E_{k+1,y}$ denote the execution of command I_{k+1} by actor S_y starting at time, T_2. A CAQ hazard is said to have occurred when: $T_2 < T_1$

While *Query-after-query* (QAQ) is not considered a hazard in WSNs, it may be a potential hazard in WSANs if the conditions have changed between the time interval for two responses. However, this requires the presence of an external entity to affect the environment and, hence, we do *not* consider QAQ as a hazard in our future discussions.

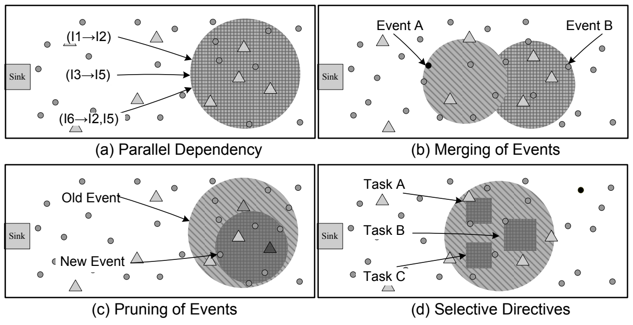


Fig. 3. Challenges Illustration

D. Challenges

So far, we have discussed the hazards in the context of single event, where complete ordering is required. However, there are several applications when this complete ordering of directives is not necessary for the entire event region. We identify these different scenarios with the following challenges.

- *Parallel Dependency among Directives*: Some applications may not require that all the directives for the event region to execute in an ordered fashion. Directives should be executed sequentially only based on the dependencies with the previous set of directives issued. Consider the case, where there are three sets of dependent directives for a particular event in the network (see Figure 3 (a)). Now, if the dependent sets of directives are completely independent across another set, then the hazard free operation is only necessary within the individual sets ($I_1 \rightarrow I_2$ or $I_3 \rightarrow I_5$ in Figure 3 (a)). Further, if a particular directive (I_6 in Figure 3 (a)) requires synchronization with respect to both sets, then that directive should be synchronized with respect to both sets.
- *Opportunistic Execution of Directives*: Thus far, we have considered applications, where it is absolutely essential to address all the hazards while executing all the directives. In certain applications, while the relative ordering of directives is important, it may not be important to execute previously issued directives if they arrive out-of-order. As an example, consider the case, where the sink sends directives $I_1 \ldots I_{10}$ sequentially. Consider the case when the application does not require that all the directives be executed, but only requires that the relative ordering between directives be maintained, if at all a directive is to be executed. In this case, if directives $I_1 \ldots I_4$ were executed sequentially, and directive I_6 arrives before I_5, then directive I_6 is executed without waiting for directive I_5. When directive I_5 arrives at a later instance of time, it will be ignored to preserve relative ordering.
- *Merging of Events*: In certain applications, it is possible that two overlapping regions of the network experience an event at about the same time (see Figure 3 (b)). If the events are unrelated, then we must allow the directives for the two set of events to proceed independently. On the other hand, if the events are the same, the set of directives will be dependent across the regions and so the directives must be addressed to both event areas so that the overlapped regions are not acted upon twice. In the fire extinguisher example, this would correspond to two fire outbreaks in close-by regions, that eventually spreads and results in an overlap of event areas. If the intensity of the fire in both regions are similar, then it should be treated as a single event with the union of both areas.
- *Pruning of Events*: When an event that had occurred in a certain region has now decreased in size, then it is necessary that the directives addressed to this event are only executed in the current event region (see Figure 3 (c)). In the fire extinguisher example, if suppose there is a fire in a region and if the intensity of fire is maximum at the center of the region, it is possible that the fire is extinguished fully in the peripheral sub-regions of the event area, while the center is still in flames. In this case, any hazard among the directives issued is only applicable in the current event region.
- *Selective Directives*: Given the vast nature of a WSAN network, it is possible that various regions of the network experience different types of event that require a different set of tasks (see Figure 3 (d)). For example, in the fire extinguisher system, it is possible that a certain region may experience fire, while another nearby region, may be experiencing flooding caused by over reacting to the fire. In this case, the problem should be addressed in such a way that it is possible to issue separate set of directives for the different regions that are not related to each other. Even if the regions overlap, as long as the events are not related to one another, the directives for each event should be executed independently.

E. Goals

Further, any hazard avoidance approach should also be efficient with respect to the following important goals:

- *Throughput*: An important goal is to increase the rate at which the queries and commands sent by the sink are processed. We define a metric called *directives execution throughput*, which is the number of directives processed per unit time, and try to maximize this metric.
- *Correctness and Overhead*: Another critical goal is that any approach that addresses the hazards and the associated challenges, does so with minimal overhead. The absence of this goal will lead to an increase in overall traffic in the network in addition to over-utilization of the sensor and actor resources. On the other hand, it should not compromise on 100% hazard free operation.

IV. Design

In this section, we formally present the basis for hazard-free operation, provide the basic mechanism of our proposed solution called the *Neighborhood Clock (NC)* approach, and show that NC is a hazard-free approach. We make the following simplifying assumptions in order to make the problem more tractable:

- *Network Model*:We consider the case, where sensors and actors are both *static*, and are randomly distributed[6] in sensor/actor field.
- *Location Information*: We assume that sensors and actors can determine their location through localization algorithms [3], [4].
- *Sensing, Acting and Communication Ranges*: We assume that sensing range (R_s) is the same as the communication range for sensors (T_s), and acting range (R_a) to be the same as the communication range for actors (T_a). Note that this assumption is not central to our approach and is mainly considered for clarity of presentation.
- *Routing Model*: We assume that there is an underlying reliable routing protocol for delivering directives and gathering responses [5], [6], [7].

A. Hazard Model and Dependency Region

The generic hazard-free operation goal can be described by the following model:

Settings: Consider a WSAN network, where the sink issues a set of directives to a set of entities[7]. Directives issued by the sink are subject to a set of dependencies determined by the sink. Let Ω denote the set of directives, Δ denote the set of entities, and Λ denote the dependency set. Each element, λ_m ($\lambda_m \in \Lambda$), defines the dependency of two directives I_i and I_j, where $I_i, I_j \in \Omega$. If I_i is required to be executed before I_j, we use $I_i \rightarrow I_j$ to denote this dependency requirement. In order to prevent all three hazards, any two directives that are dependent should be executed sequentially according to the dependency.

Goal: The design goal of an efficient hazard-avoidance approach is to determine a hazard-free execution process, which has minimum execution time subject to Ω.

Constraints: Consider any two entities D_x and D_y in Δ within the event region that are required to execute two directives I_i and I_j, with dependency requirement $I_i \rightarrow I_j$. Let $t_{I_k \cdot D_z}$ denote the time that an instruction, I_k, is executed by an entity, D_z. To prevent a hazard, it is required that $I_i \rightarrow I_j$ for both entities. Let $A(D_x)$ and $A(D_y)$ denote the execution region[8] of D_x and D_y, respectively.

If $A(D_x)$ and $A(D_y)$ do not have any overlap, the hazard prevention requirement for the dependent directives, I_i and I_j, can be satisfied by the following set of relations:

$$t_{I_j \cdot D_x} > t_{I_i \cdot D_x} \tag{1}$$

$$t_{I_j \cdot D_y} > t_{I_i \cdot D_y} \tag{2}$$

These rules are straightforward from the definitions of the three hazards described in Section III. Thus, as long as the equations 1, 2 are both satisfied, $I_i \rightarrow I_j$ for the pair of entities, D_x, D_y, is guaranteed. These equations essentially imply that the directives need to be executed in-order by each of these entities, but there is *no* ordering requirement across the two entities, D_x and D_y.

When $A(D_x)$ and $A(D_y)$ have overlapping areas, *all* of the following four hazard-avoidance rules have to be followed:

$$t_{I_j \cdot D_x} > t_{I_i \cdot D_x} \tag{3}$$

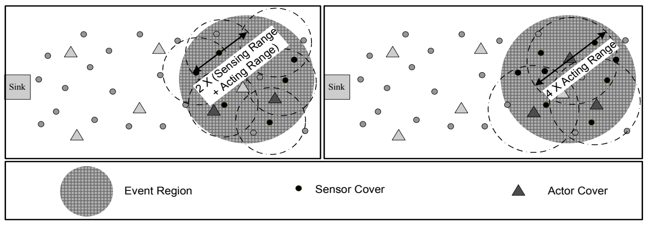
$$t_{I_j \cdot D_x} > t_{I_i \cdot D_y} \tag{4}$$

$$t_{I_j \cdot D_y} > t_{I_i \cdot D_x} \tag{5}$$

$$t_{I_j \cdot D_y} > t_{I_i \cdot D_y} \tag{6}$$

It can be shown that in order for equations (3)-(6) to be satisfied, the directives I_i and I_j need to be ordered in the region, $A(D_x) \bigcup A(D_y)$ [8]. Based on the above discussion, the following two inferences can be made:

- *Any pair of dependent directives issued to entities that do not have any overlapping execution regions can be executed concurrently across the two entities, although the relative ordering must be preserved within each entity.*
- *Any pair of dependent directives issued to entities with overlapping execution regions needs to be ordered in the union of the two regions.*



(a) Dependency Region for Sensor (b) Dependency Region for Actor

Fig. 4. Maximum Dependency Regions

Now, for a given entity, D_x, applying these rules pairwise with any other entity in the event region, we can define a region in the neighborhood of D_x called the *dependency region*, where perfect ordering is necessary. For sensors and actors that are beyond the dependency region, there is no dependency across the two regions (even if the instructions are dependent). The dependency region of a sensor can be defined as the region with radius equal to the sum of sensing and acting range ($Sensing\ Range + Acting\ Range$), while that of an actor is the region with radius as twice the acting range ($2 \cdot ActingRange$)(see Figure 4).

B. Need for Neighborhood Clock

From the above discussions, we can infer that within a dependency region, two directives (I_i and I_j) with the dependency requirement $I_i \rightarrow I_j$ have to be executed *atomically* in that order. This *atomic ordering* implies that I_j can be executed on any entity within the dependency region only after all the entities in the dependency region have executed I_i. In a WSAN without synchronization, in order to achieve this atomic ordering, the sink has to ensure that the previous directive has been executed on all the entities within the dependency region before issuing the next directive. One way to

[6]We assume that the network is connected through these sensors and actors.
[7]We refer to both sensors and actors with the common name *entity*.
[8]The execution region for a sensor is its sensing region while that of an actor is its acting region.

ensure this is to wait for a significant portion of time between successive directives so that acknowledgements are received from all the sensors and actors about the completion of the previous directive. However this is clearly not efficient and it requires central coordination by the sink for each dependency region. If the execution of directives can be *synchronized* on the entities within a dependency region efficiently, it is possible to guarantee hazard free operation in a decentralized fashion.

There are several alternatives to synchronize the execution of directives. Based on the granularity required for synchronization, the synchronization can either be time-level or event-level. In order to perform time synchronization, an underlying time synchronization mechanism is necessary. However, communication cost and resource overhead associated with performing time synchronization render this approach inefficient. Moreover, to achieve hazard-free operation, an event-level synchronization mechanism will suffice since the execution of directives can be considered as events (that require synchronization). In order to achieve event-level synchronization on the directives, a node can either use a physical clock or virtual clock. A virtual clock approach is preferred, since the physical clocks on different entities are not a reliable means of ensuring ordering without a careful design of the clock synchronization algorithm and the large amount of traffic associated with it [9]. On the other hand, a virtual clock approach only requires synchronization at a coarse level and hence requires less maintenance and communication overheads.

Based on these observations, we propose a localized virtual clock synchronization approach called the *Neighborhood Clock (NC)* approach. In this section, we consider the case of a *scalar* clock, where the clock is given by a *sequence number*.

C. Neighborhood Clock Mechanism

In this section, we propose and describe the *Neighborhood Clock* (NC) mechanism and show that it addresses all the hazards identified in Section III. For now, we assume that there is only one type of event in the event region and that all the directives are addressed to the same event region. We also assume that the set of directives for this event region are all dependent.

NC introduces the notion of a *neighborhood clock* on every sensor and actor for ordering the directives within every *dependency* region. The neighborhood clock is used to enforce synchronization between sensors and actors within a dependency region. It does *not* enforce synchronization beyond the dependency region of any sensor or actor, thereby allowing the other nodes in the event region to execute the directives concurrently.

When the sink learns about an event, the sink creates a *reference clock* for that event, and initializes this clock to a unique start value, denoted by NC_0. This *reference clock* is used to indicate the progression of directives sent by the sink. This information is flooded throughout the event region. When any sensor or actor in the event region receives the message, it initializes its neighborhood clock by the initial reference clock value. In this fashion, all nodes can synchronize their initial neighborhood clock values. Whenever the sink sends a directive, it increments the reference clock. The reference clock of the sink, RC_i, is piggybacked with the i_{th} directive. Since the RC values increase linearly for every directive sent, the neighborhood clock is ordered according to the sequence in which the directives were issued.

Each entity, D_x, maintains its own *view* of the progress in the network, based on its neighborhood clock identifier, $NC(x)$, where the view number is set to be $NC(x) + 1$. NC functions by synchronizing the views on all the sensors and actors within the dependency region, which is performed by synchronizing the NC values of all neighborhood clocks. Each sensor and actor will move to the next view only after all other sensors and actors have moved into its current view[9]. Thus, the difference between the views of any two nodes within the dependency region will be at most 1. Any entity, D_x, can only execute a directive if the NC value piggybacked is the same as the current view number. That is, if an entity is in $view_i$, it is allowed to execute the directive with NC value equal to i. By enforcing this scheme, NC can ensure the atomic execution of every directive on all the entities. Once an entity executes a directive, it notifies the completion of the directive to other entities in the dependency region. The progress of views within

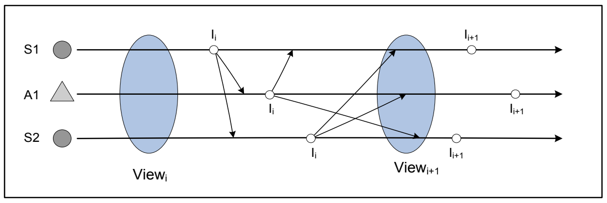


Fig. 5. View Movement

a dependency region, which consists of two sensors and one actor is illustrated in Figure 5. In the figure, at a certain time all the entities have moved into $view_i$, as illustrated in the left ellipse area. So all of them are allowed to execute the directive with $NC = i$. After the execution, each entity notifies the other entities about the execution of the directive. Whenever an entity receives notifications from all other entities, it will move to the next view, $view_{i+1}$, as illustrated in the right ellipse area.

D. Addressing Hazards using Neighborhood Clock

We will now show that by enforcing neighborhood clock synchronization, in the dependency regions, all the three hazards identified in Section III can be avoided reliably. Recall that the radius of the dependency region of a sensor is the sum of the sensing range and acting range, and that of an actor is twice the acting range. We present the proof for a generic hazard, denoted by XAY hazard, where Y and X are successive directives issued to the event region.

[9]While this assumes that there are no node failures between the execution of two directives, our approach addresses this issue by having a timeout-based mechanism. We do not present the details of this mechanism due to lack of space.

Proof: For any pair of entities D_x and D_y, let us use $d(D_x, D_y)$ to denote the distance between them. Consider two sequential directives, I_1 and I_2, with NC values i and $i+1$, respectively. Now suppose at a certain time I_1 has been executed by an actor D_x; we can infer that D_x must be in the $view_i$, as in the NC mechanism an entity can execute a directive only when the NC value piggybacked equals the view number.

For a XAY hazard in this context to happen, based on our observations in Section IV-A, it has to be the case that, say, the entity D_x executes directive I_2 before another entity within the dependency region, say D_y, executes directive I_1. Allowing D_x execute directive I_2 before D_y executes I_1 implies that D_x must be in $view_{i+1}$. But this is impossible in the Neighborhood Clock mechanism, since without receiving the notifications for the completion of directive I_1 from all the entities within the dependency, D_x can not move to $view_{i+1}$. Therefore, any generic XAX hazard cannot occur within the dependency region using the Neighborhood Clock mechanism. ■

Now, using this proof, we observe that all three hazards are addressed by the Neighborhood Clock mechanism.

- *CAC hazard avoidance*: In this case, D_x and D_y are both actors, while I_1 and I_2 are both commands, with the dependency requirement, $I_1 \rightarrow I_2$. Using the above proof, it can be observed that within a dependency region for any actor defined by a region with radius, $2 \cdot R_a$, there are no hazards. Moreover, based on the two observations made in Section IV-A, there are no hazards beyond the dependency region. Thus, CAC hazard can be completely avoided using the Neighborhood Clock mechanism.
- *QAC hazard avoidance*: In this case, D_x is a sensor and D_y is a sensor, while I_1 is a command and I_2 is a query. Again, the above proof shows that there are no hazards within a dependency region with radius, R_s+R_a. Thus, there is no possibility of a QAC hazard as the Neighborhood Clock enforces ordering within every dependency region.
- *CAQ hazard avoidance*: Here, D_x is an actor and D_y is a sensor, while I_1 is a query and I_2 is a command, with the dependency requirement, $I_1 \rightarrow I_2$. Again, using the above proof, there are no hazards within a dependency region with radius, $R_s + R_a$. Thus, there are no CAQ hazards in the NC mechanism.

V. Approach

In Section IV, we had presented the basic Neighborhood Clock mechanism with scalar neighborhood clocks and described how it addresses the different hazards. In this section, we present the Neighborhood Clock (NC) approach in detail and show that it addresses the challenges identified in Section III.

A. The Neighborhood Clock Approach

1) Construction of Dependency Regions: When an event has occurred, the sensors in the event region detect and report the event to the sink (either automatically or after they are probed). In this fashion, the sink knows the presence of the event in the region. We assume as part of the initial set up of the network that there is an underlying 2-hop neighbor discovery mechanism so that each node can advertise their node locations to its 2-hop region. In [10], the authors discuss an approach for local broadcast of beacons in order to transmit location information of the node as a basic step to ensure sensor coverage. This technique can be extended to a 2-hop neighborhood in order to transmit the location information. The neighbor discovery mechanism will allow each sensor and actor in the network to learn about all the other sensors and actors within the dependency region. This knowledge of neighbors will allow every sensor and actor node to construct a routing structure instantaneously when it receives the first directive as we will see later in this section. We will now describe the neighborhood clock structure that will be used for synchronization and describe the operations at the sink and each node in the event region.

Task	Directive List	Dependencies
Event A	1	1→3
	2	2→4
Event B	1	1→-2→3

Fig. 6. Example showing the Event List for Two Events

2) Need for a Vector Neighborhood Clock: In Section IV we had mentioned the need for a scalar neighborhood clock and described how it can be leveraged to support synchronization within the dependency region. The design of a neighborhood clock limits the scope of synchronization to a single event over all the sensors and actors in the event region. Further, it also assumes complete ordering is necessary between all the directives. As we had described in the challenges in Section III, there are several applications when this complete ordering of directives is not necessary for the entire event region. For these cases, the scalar neighborhood clock is too restrictive and will be either inefficient or will not be able to provide the desired ordering.

Vector neighborhood clock extends the idea of a scalar neighborhood clock to include an array of clocks based on the number of *dependency lists* between directives for each type of event, where a dependency list is defined by the chain of dependencies for all previous directives. Thus, if there are k different events and a maximum of j dependency lists corresponding to any event, a vector clocks will be a two dimensional array of size kXj. Consider the example shown in Figure 6, where there are two events happening in the same region. Associated with Event A, there are two dependency lists, while there is only one dependency list associated with Event B. The vector clock for this example, would be of size $2X2$, where each row represents the set of clocks for that particular event. Note that, while it may be *beneficial* to maintain a list of clocks instead of an array if there are varying number of dependency lists across different events, we have not presented it here for clarity of presentation.

3) Operations at the Sink: We consider a *real-time* model[10] for issuing directives, where the sink does *not* know the list of dependencies *apriori*. After the sink has received responses from all the sensors in the event region, the sink creates a *vector reference clock*[11] for the event. Additionally, the sink also creates an event identifier ($Event\ ID$) and records the event region corresponding to a particular $Event\ ID$. When the sink learns about an event based on the responses received, the sink initially sets its reference clock by initializing the array of scalar reference clocks. This would correspond to one row in the reference clock array. The sink then sends a $START()$ control message, where the $Event\ ID$ and the (vector) reference clock to all the sensors in the event region using the underlying delivery mechanism. When any sensor or actor in the event region receives the message, it also creates an identical vector clock array. We refer to this clock array maintained at the entities as the *vector neighborhood clock*[12]. In this fashion, all nodes can synchronize their initial vector clock values to that of the reference clock. In the example considered in Figure 6, for the $Event\ A$, the initial reference clock would correspond to (A,0,0) (assuming that the initial clock values are 0). For now, we present our approach in the context of a single event, for clarity of exposition. We revisit our approach later in this section and present how the NC approach can address multiple events.

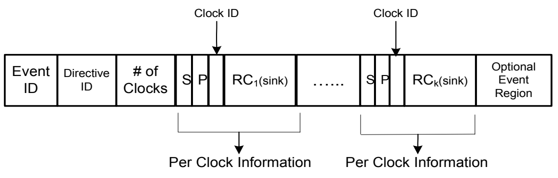


Fig. 7. Format of Header for a Directive

When the sink wants to send the i_{th} directive (query or command), and if this directive belongs to the j_{th} dependency list, then the corresponding scalar reference clock is updated by the following equation:

$$RC_j(sink) = RC_j(sink) + 1 \tag{7}$$

The event identifier, directive identifier and the number of clocks, which this directive depends on, and the information regarding these clocks is piggybacked along with the i_{th} directive. Figure 7 shows the format for each directive sent by the sink. The per clock information includes the clock identifier, the value of the reference clock and the service model (denoted by S and P symbol), which we will discuss later in this section. Here, the event region is sent as an option only if the directive is sent to an event that has increased or decreased in size or merged with another event.

[10]While the NC approach is equally effective even for an off-line model, we believe this represents a more generic and practical setting for directive initiation in WSANs. For this reason, we have tailored our mechanisms to address this model.

[11]We refer to vector reference clock as reference clock from now on.

[12]We refer to the vector neighborhood clock as vector clock for brevity

Since the RC_j values increase linearly for every directive sent from a particular directive list, the scalar reference clock combination is ordered according to the sequence in which the directives were issued from a particular dependency list, which will be leveraged to address the different types of hazards. The directives that need not be ordered are maintained by separate individual neighborhood clock values. We will now describe the actions at the sensors and actors when a directive reaches the event region.

4) Operations at the Sensors/Actors: As we had mentioned earlier, the $START()$ message received by a sensor or actor will serve as a trigger for this instantaneous construction of routing structure in the dependency region. In the NC approach, every node in the event region constructs a shortest path tree [11] to every sensor and actor in the dependency region[13] and uses a hop-to-hop reliability mechanism in the case when there are losses. This structure is efficient as one of our primary objectives is to increase the throughput in the execution of directives.

The sequence of operations in each sensor or actor in the event region is shown in Figure 8. When the sink sends the k_{th} directive, a node in the event region performs the following sequence of operations:

- If the clock list piggybacked with the directive requires synchronization with respect to neighborhood clock, $NC_m(i)$, and $RC_{l_m}(k) = NC_m(i)+1$ (m_{th} clock in the list of reference clocks, $RC_l(k)$), the action is performed and an $ACK()$ message is sent to all neighbors in the dependency region (lines 14-17, 40-47 in Figure 8). Additionally, if the node is a sensor, it responds to the directive (which will be a query) (line 42 in Figure 8).
- If the neighborhood clock, $NC_m(i)$, is at least 2 less than the reference clock identifier of the sink $RC_l(k)$ (lines 34-39, 18 in Figure 8), the directive is queued and no action is taken (line 19 in Figure 8).
- If an acknowledgement, $ACK()$, is received from any of the nodes in the dependency region for that directive, the node identifier is first added to a list maintaining all the nodes from which acknowledgements have been received for every neighborhood clock that this directive is dependent on (lines 22-25 in Figure 8). Additionally, the node checks if it has received ACKs from all nodes with in the dependency region[14] for that directive and increments all the neighborhood clocks corresponding to the different dependency lists the directive belongs (lines 26-29 in Figure 8).

In this fashion, any sensor or an actor will update its vector clock only after it has received the same vector clock values corresponding to the dependency lists to which the directive belongs from all the nodes. After it receives all the notifications, it will increment the corresponding set of clocks for this

[13]The nodes in the periphery of the event region will construct the routing structure for the nodes (within the dependency region) that are part of the event region.

[14]For the peripheral nodes, this region is the intersection of the event region and the dependency region for that node.

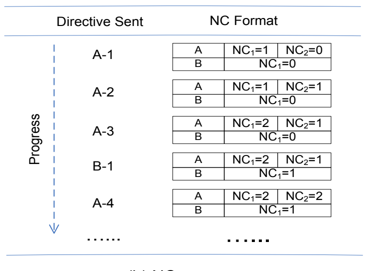
```
Variables:
1    $i$: Node id,
2    $NC_0(i) \dots NC_k(i)$: Array of Neighborhood Clocks, $E(i)$: Event ID,
3    $RC_l(k)$: List of enclosed clocks in the $k_{th}$ directive,
4    $DIR(\dots, RC_l(k), \dots)$: Directive with Clock List $RC_l(k)$,
5    $MSG_{RCV-TYPE}$: Type of message received,
6    $Flag$: Execute or not flag,
7    $DIR_E(i)$: Queue of waiting Directives
8    $L(i)$: List of 2-hop neighbors,
Receive(i)
9    If ($MSG_{RCV-TYPE} == (START()$ or SYNC())
10       For (j=1:k)
11           $NC_j(i) = NC_j(sink)$
12       End for
13       $E(i) = E(sink)$
14   If ($MSG_{RCV-TYPE}$==$DIR(\dots, RC_l(k), \dots)$)
15       Check($i, RC_l(k)$)
16       If ($Flag$==TRUE)
17           Execute(i)
18       Else
19           add $DIR$ to $DIR_E(i)$
20       end Else
21   end If
22   If ($MSG_{RCV-TYPE} == ACK$ from $k$)
23       For every $NC_l(i) \in NC_l(k)$
24               Update $NC_l(i)$
25       End for
26       For every enqueued directive $DIR_n \in DIR_E(i)$
27           Check($i, RC_l(k)$)
28               If ($Flag$==TRUE)
29                   Execute(i)
30               Else add $DIR$ to $DIR_E(i)$
31   If ($MSG_{RCV-TYPE} == REQ - ACK$)
32       Reply with its Neighbor Clocks
33   Return
Check($i, RC_l(k)$)
34   $Flag$=TRUE
35   For every $NC_m(i) \in RC_l(k)$
36       If ($RC_{l_m}(k) \neq NC_m(i) + 1$)
37       $Flag$=FALSE
38   End for
39   Return $Flag$
Execute(i)
40   If ($MSG_{RCV-TYPE}$==$QUERY$)
41       Perform Sensing Task
42       Send ($i, Response$) to Sink
43   Else if ($MSG_{RCV-TYPE}$==$COMMAND$)
44       Perform Acting Task
45   If (Ordering Required for $DIR$)
46       $Notify(i, NC_l(k))$
47   Return
Notify(i,$NC_l(k)$)
48   Send ACK to all entities in $L(i)$
49   Return
RequestACK($i, RC_l(k)$)
50   Send $REQ_ACK$ to $L(i)$
51   Return
```

Fig. 8. The NC Approach at Each Node for One Event Type with Multiple Dependency Lists

directive. If on the other hand, it receives a directive that is not dependent on any of the previous directives, it steps into the next view directly by incrementing a separate clock.

Task	Directive List	Dependencies
Event A	1	1→3
	2	2→4
Event B	1	1→-2→3

(a) Event List



(b) NC progress

Fig. 9. NC Approach Progress for the Example Considered

Figure 9 shows the progress of the neighborhood clocks for two events with dependency lists as indicated. Consider the progress of the vector clock corresponding to $Event\ A$. The nodes maintain separate clocks for the two dependency lists as shown in the figure. When directive 1 arrives, the directive is executed and the neighborhood clock, NC_1, is incremented as it is the first directive. Subsequently, when the directive 2 arrives, since it has no dependency with directive 1, the directive is executed immediately. A separate neighborhood clock, NC_2, corresponding to this second dependency list is created and the value of NC_2 is incremented. When directive 3 arrives, the execution of the directive is only determined based on the clock, NC_1, as this directive is only dependent on the first directive. If $NC_1 = 1$, that is, directive 1 has already been executed, directive 3 is executed, and the corresponding clock value, NC_1 is incremented. Thus, we can see that the progress of the execution of directives in NC approach is stalled only when there are dependency requirements (within a dependency list).

We will now describe the mechanisms in the NC approach that address the different challenges identified in Section III.

B. Parallel Dependency between Directives

This challenge pertains to independent and concurrent execution of directives when the directives belong to different dependency lists.

In the NC approach, the use of the vector clock allows it to only maintain the last executed directives corresponding to each dependency list. The vector clock in NC is an array of neighborhood clocks based on the number of dependency lists, where each list has a separate clock. For every directive, the sink encodes the dependency information in the list of clocks piggybacked along with the directive. When a node receives the directive, it only synchronizes its vector clock with respect to the list of clocks mentioned in the directive. Therefore, the nodes do not unnecessarily wait for directives that are not related to this directive, even if they were prior to this directive. Consider the event, $Event\ A$, in the example shown in Figure 9. The progress of the individual clocks is only determined based on the dependency requirements within each dependency list.

C. Opportunistic Execution of Directives

Thus far, we have considered applications, where it is absolutely essential to execute all the directives in order if they are dependent. In certain applications, it is not necessary to execute all the directives but only ensure relative ordering between the directives that are executed.

The NC approach addresses this challenge by using directive sequence numbers for scalar neighborhood clocks. Initially, when the $START()$ control message is sent, the sink will include a special control bit to include the mode of directive execution to be in the opportunistic mode. In this mode, the nodes will only ensure that the progress of the neighborhood clock array is monotonically increasing. When a directive is executed, the list of clocks on which the directive was dependent on, are all incremented by the sequence number of the directive. If at a later instant a lower sequence number directive, which has a dependency on any one of the clocks updated, is received, it will be dropped as the neighborhood clock value(s) will be higher than the ones listed in the directive. For $Event\ B$ in the example shown in Figure 9, consider the case when directive 3 was received and executed before directive 2. In the opportunistic mode of directive execution, directive 2 will be ignored if it is received at a later instant.

D. Merging of Events

This challenge pertains to the ordering of directives issued to identical types of event with overlapping event regions. In this case, the set of directives should be ordered with respect to the entire merged region.

In the NC approach, the format specified in the directive header and the $SYNC()$ control message can be leveraged to address this challenge (see Figure 7). Consider the case when an event region, $Region_A$ is executing a directive with sequence number x and another event region, $Region_B$, is executing a directive corresponding to sequence number y ($y > x$). When the sink learns from the responses that the two event regions have merged, the sink issues a $SYNC()$ command to synchronize the clocks of both event regions. This $SYNC()$ message, which is similar to the $START()$ message, will be sent to $Region_A$, where the initial vector reference clock is set to the reference clock corresponding to the event with the higher directive sequence number[15] ($Region_B$). In this fashion, nodes in $Region_A$ can synchronize its vector clock to that in $Region_B$.

[15]Note that it is equally possible to choose the reference clock for the event corresponding to the lower sequence number directive

E. Pruning of Events

This challenge concerns the region to which the hazard free operation is necessary. If an event that has occurred in a certain region has now decreased in size, then only a part of the original region needs to be addressed for hazards.

In the NC approach, the options field in the directive format can be leveraged to let the sensors and actors know the change in the event region. Based on the responses received from the event region, the sink can identify that the current event region has shrunk in size. Any directive issued by the sink after this instant, which will still be delivered to the entire event region, will include the current area information in the options field in the directive header format (Figure 7). Thus, each node in the event region can identify if it is part of the current event region. If a node is within the event region, it does synchronization according to the mechanisms in the NC approach. The nodes that are outside but within an execution range distance from the boundaries of the current event region, will also synchronize as their sensing or acting ranges may overlap with the current event region. However, nodes that are beyond this region merely increment their corresponding neighborhood clock without actually executing the directives or synchronizing their clock values. In this fashion, if at a later instant, a directive is sent to the original event region, it is still possible for the nodes outside the current event region to execute the directive and perform clock synchronization.

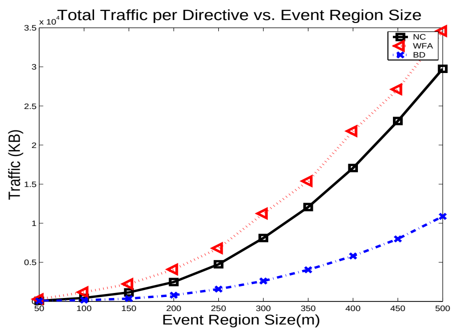
F. Selective Directives

This challenge corresponds to the different set of directives (tasks) issued to different regions of the network, where the set of directives may or may not be dependent across regions.

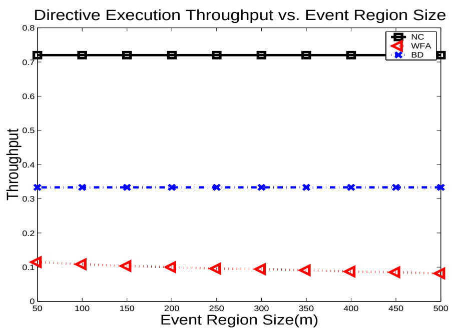
In the NC approach, this challenge can be addressed by leveraging the format of the directive header and using the $SYNC()$ control message. If the different tasks are unrelated, irrespective of whether the task regions overlap, the vector clock associated with each task will have separate event identifier (treated as separate events). However, if events are related and the dependency regions of any two nodes corresponding to the two task regions do not overlap, the nodes in those regions will still be assigned different event identifiers (see Section IV). If on the other hand, the regions are overlapping and if the tasks are dependent, then strict synchronization between these two task regions can be imposed by the mechanisms for merging. This is also done for the case when the task regions for two dependent events are within twice the acting range of one another.

VI. Performance Evaluation

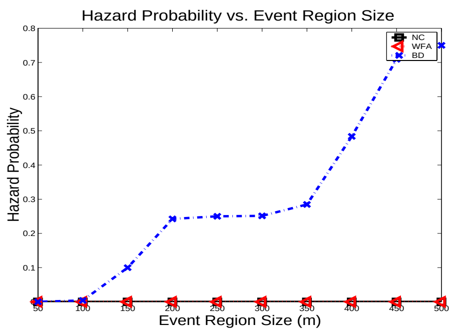
This section evaluates the performances of the proposed approach (NC approach) with two basic strategies: (i)Wait-For-All (WFA), and (ii) Bounded Delay (BD). In Wait For All (WFA) strategy, the sink issues the next directive only after it receives all the responses or notifications for the previous



(a) Total Traffic per Directive



(b) Directives Execution Throughput



(c) Correctness

Fig. 10. Performance under event region size

directive. For instance, if the sink sent out a command, it will wait for acknowledgements from all the actors before it issues another query or command. The Bounded Delay (BD) strategy works by striking a balance between the degree of correctness for avoiding hazards and the efficiency of event processing. In BD, after issuing a query, the sink waits for time T_{W_s} before issuing the next directive. Similarly, after a command is sent, the sink waits for at least T_{W_a} prior to sending another directive.

The performance metrics considered are the total traffic per directive, directives execution throughput and correctness. We study the performance of the three approaches for a variety of network conditions by varying the size of event region, sink-to-event distance and percentage of directives that are queries.

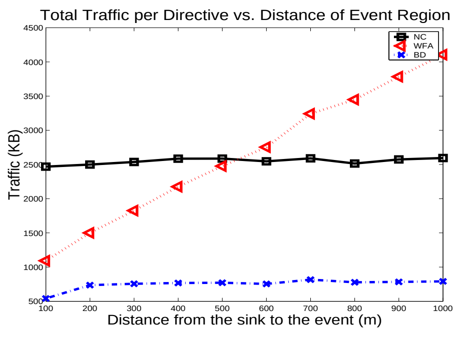
A. Simulation Environment

For all simulations, sufficiently large number of sensors and actors are randomly placed on a 2000m$\times$2000m square area to ensure connectivity. The sensing range and communication range of sensors is set to be 30m, and the acting range and communication range of actors is 60m. Thereby, for the NC approach, the neighborhood range for a query is 90m (30m+60m), while for a command, it is 120m (60m +60m). When an event is detected, a minimum sensor cover [12] and actor cover is formed. When the sink sends directives to the corresponding sensors or actors in the event region, the directives are first sent to the closest sensor or actor in the corresponding cover set. The closest sensor or actor then forwards the directives to the entire cover set. We assume CSMA/CA as the MAC protocol, with a retransmission time of 1 second when there are losses. The retransmission is repeated until successful delivery is achieved.

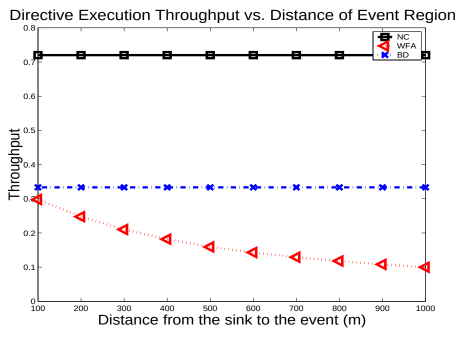
The events considered in the simulation are regional events with varying radius ranging from 50m to 500m. In the following results, if not specified explicitly, the distance from the sink to the event center is 1000m, the radius of event region is 200m and the loss rate per hop is 10%. Amongst the directives issued by the sink, the probabilities of queries and commands are both 50%. When a command is received by an actor, we consider a default value for the event-processing time, $T_{EP} = 2$ seconds. For BD, $T_{W_s} = T_{EP}$ and $T_{W_a} = 2 \cdot T_{EP}$. All results are averaged results on 10 different runs, and for each run, 100 directives are sent out and executed. The message size for all messages (directives, responses, notifications) are assumed to be 1KB, and the total amount of traffic is computed as the total KB used to execute a directive, including directive delivery, responses, and notification. The directives execution throughput is defined as the number of directives executed per second, and the correctness is measured by the probability of hazard occurrence. *Note that the correctness of NC and WFA are both 100%.*

B. Varying the Event Region Size

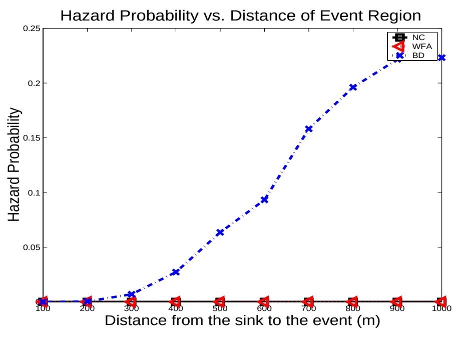
Figures 10 shows the performance results of the three approaches under varying event region size. As shown in Figure 10(a), with increasing event region size, the traffic per directive of all three approaches increases. In BD, this is mainly because of the increase in the number of nodes in the event region. While the BD achieves the best performance in terms of overhead (in fact no overhead), it is only at the expense of correctness and throughput. For NC, aside from this reason, since each node has to receive notifications from all other nodes within its dependency region, the overhead is relatively large. For the WFA strategy, since the acknowledgements to each directive has to be sent by all nodes to the sink, the traffic is very large. Figure 10(b) shows the throughput variation for increasing event size. As we can observe, NC has the largest directives execution throughput when compared with the basic approaches because the dependency region is just the 2-hop neighborhood region. Both NC and BD have constant throughput values since their mechanisms are not affected by the region sizes, while WFA's throughput drops slightly due to the fact that it must wait for more time to receive all the acknowledgements before issuing the next directive. Figure 10 (c) shows that NC and WFA are both hazard-free, while BD has increasing hazard probabilities with larger event region. The hazard probability increases from 1% to almost 75% when the event radius varies from 50m to 500m. As we do not consider QAQ to be a hazard in our simulations, the maximum hazard probability is 75%. In BD, since the sink waits for a longer time after issuing a command, QAC and CAC hazards happen with a low probability. However, a CAQ hazard is more likely to happen as the sink only waits for a smaller amount of time after issuing a query.



(a) Total Traffic per Directive

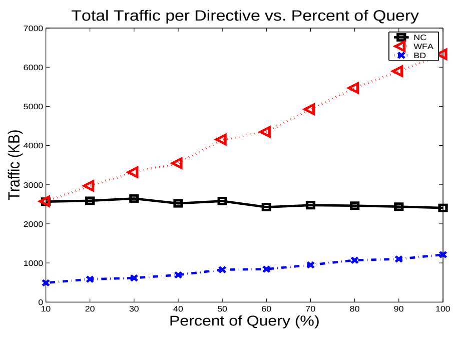


(b) Directives Execution Throughput

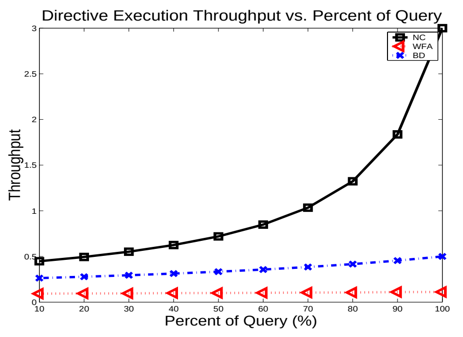


(c) Correctness

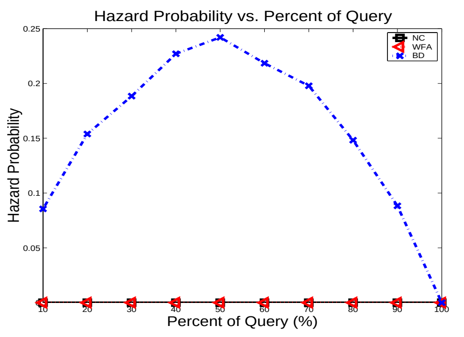
Fig. 11. Performance under different sink-to-event distance



(a) Total Traffic per Directive



(b) Directives Execution Throughput



(c) Correctness

Fig. 12. Performance under different percent of queries

C. Varying the Distance from the Sink to the Event Center

Figure 11 shows the performance results of the three approaches for varying sink-to-event distances. We can see that in WFA has a much higher overhead in dealing with far-away events due to the fact that all the sensors and actors in the event region are required to respond back to the sink. As shown in Figure 11(a), both BD and NC have (almost) constant traffic, which only increases slightly with increasing sink-to-event distance. This is because the average traffic in delivering the directive is almost a constant. Additionally in NC, the traffic generated within the dependency region will always be a constant. Figure 11(b) shows that NC has largest throughput. The throughput of WFA drops because the waiting time for issuing a directive increases with increasing sink-to-event distance. Unlike WFA, the throughput of NC and BD do not change with the sink-to-event distance, since the latency between the execution of successive directives does not depends on the distance of the event from the sink. Similar to that of increasing event region size, the hazard probability of BD is higher for a farther away event, which is shown in Figure 11(c).

D. Varying the Percent of Queries

Figure 12 shows the performance for varying probabilities of queries and commands in the set of directives issued. In these figures, the percents of queries vary from 10% till 100%. In Figure 12 (a), the traffic required per directive for WFA and BD increase, while NC experiences an slightly dropping traffic overhead. For WFA and BD, the queries are sent to the sensors in the sensor cover set, while the commands are sent to the actors in the actor cover set. Since we assume that the sensing range is smaller than the acting range, it is expected that the size of sensor cover set is larger than that of actor cover set. Thereby, a larger portion of directives being queries results in an increasing amount of traffic. However, in NC approach, sensors and actors always send execution information across their dependency region, where the dependency region for a sensor (R_s+R_a) is smaller than that of an actor $(2 \cdot R_a)$. Hence, an increase in the percentage of queries in the directives issued results in a lower traffic overhead for NC.

Figure 12 (b) shows that the directives execution throughput increases for all three approaches. For NC, the throughput increases because of the decrease in both the average execution time of a directive and the size of the dependency region. For BD, the throughput only increases slightly because of the decrease in average execution time of a directive, while for WFA, it remains a constant as the sink anyway waits for acknowledgements in this strategy. Figure 12 (c) shows the hazard probability for all three approaches. In BD, CAC and QAC hazards rarely happen in our simulation environment because of the relatively large waiting time chosen for commands and the sink-to-event distance. Thereby, initially when the query percent increases, (only) the CAQ hazard probability increases resulting in a higher hazard probability. However, as the query percent continue to increase, the overall hazard probability begin to drops, since the percent of QAQ ordering is larger (and QAQ is not considered a hazard).

VII. Related Works

A. Pipelining

The problem considered in this paper shares some similarities to pipelining of instructions in the computer architecture domain [13]. Pipelining is a very popular practice for increasing instruction level parallelism, provided the underlying instruction set has a minimum set of dependencies. In order to resolve any dependencies within the instruction set several techniques have been proposed including instruction re-ordering and register re-allocation. This is philosophically similar to what we have tried to achieve in our approach in terms of increasing the parallelism in issuing directives by having directives that are not dependent executed between dependent directives. However, in WSANs, we not only have to maximize the directive level parallelism but also region-level parallelism.

B. Parallel Programming

The hazards described in this work has some resemblance to the synchronization problem in the context of multiprogramming in the operating systems domain [14]. In order to ensure sequential access of the critical section of a piece of code, synchronization is a necessary condition. In parallel programming, software primitives such as semaphores and monitors are used to bring about synchronization. However, these approaches are not suitable in the context of WSANs.

C. Distributed Systems

The NC approach proposed in this paper shares some ideas from the distributed systems area. A distributed system consists of a set of processes that cooperate to achieve a common goal, but do not share a common global memory. To capture the causality relationship between events, both logical clock and vector clock models are used [15]. The sequence numbers enclosed in the directives in NC serve as the logical clocks, and the directive dependencies determined by the sink share similarities with the causalities. Unlike in distributed systems, where the goal is to achieve global ordering for a set of processes, NC addresses the hazard problems only within the dependency region of an entity.

VIII. Conclusions

In this paper, we have identified the problem of hazards in the context of a wireless sensor and actor network, and described the associated challenges. We have discussed the basic design philosophy needed to address hazards in an efficient fashion. We have also proposed a localized and fully distributed approach based on the design philosophy that addresses this problem effectively. Finally, we have performed extensive simulations to understand the trade-offs between the proposed solution and two basic approaches that can also be used for addressing hazards.

Acknowledgment

This work was funded in part by the National Science Foundation under grants ECS-0225497, ECS-0428329, ANI-0117840, and CCR-0313005.

References

[1] I. H. Kasimoglu I. F. Akyildiz, "Wireless Sensor and Actor Networks: Research Challenges," in *Ad Hoc Networks Journal*, 2004.

[2] I.F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A Survey on Sensor Networks," in *IEEE Communications Magazine*, Aug. 2002, vol. 40, pp. 102–116.

[3] N. Bulusu, J. Heidemann, and D. Estrin, "GPS-Less Low Cost Outdoor Localization for Very Small Devices," in *IEEE Personal Communications, Special Issue on Smart Spaces and Environments*, Oct. 2000, pp. 28–34.

[4] N. Bulusu, J. Heidemann, and D. Estrin, "Adaptive Beacon Placement," in *Proceedings of the Twenty First International Conference on Distributed Computing Systems (ICDCS-21)*, Apr. 2001.

[5] S.J. Park, R. Vedantham, R. Sivakumar, and I.F. Akyildiz, "A Scalable Approach for Reliable Downstream Data Delivery in Wireless Sensor Networks," in *Proceedings of the international symposium on Mobile Ad Hoc Networking and Computing (ACM MOBIHOC)*, May 2004.

[6] C-Y. Wan, A. Campbell, and L. Krishnamurthy, "PSFQ: A reliable transport protocol for wireless sensor networks," in *Proceedings of the international Workshop on Sensor Networks and Arch. (WSNA)*, Sept. 2002, pp. 1–11.

[7] Y. Sankarasubramaniam, O.B. Akan, and I.F. Akyilidiz, "ESRT: Event-to-Sink Reliable Transport in wireless sensor networks," in *Proceedings of the international symposium on Mobile Ad Hoc Networking and Computing (ACM MOBIHOC)*, June 2003, pp. 177–188.

[8] R. Vedantham, Z. Zhuang, and R. Sivakumar, "Hazard Avoidance in Wireless Sensor and Actor Networks," in *Technical Report, Dept. of ECE, Georgia Inst. of Technology*.

[9] W. Yuan, S. V. Krishnamurthy, and S. K. Tripathi, "Synchronization of Multiple Levels of Data Fusion in Wireless Sensor Networks," in *IEEE GLOBECOM*, 2003.

[10] S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M.B. Srivastava, "Coverage Problems in Wireless Ad-hoc Sensor Networks," in *Proceedings of the IEEE conference on Computer Communications (IEEE INFOCOM)*, Apr. 2001.

[11] Jeffrey E. Wieselthier, Gam D. Nguyen, and Anthony Ephremides, "Energy-efficient broadcast and multicast trees in wireless networks," *Mobile Networks and Applications*, vol. 7, no. 6, 2002.

[12] H. Gupta, S. Das, and Q. Gu, "Connected Sensor Cover: SelfOrganization of Sensor Networks for Efficient Query Execution," in *Proceedings of the international symposium on Mobile Ad Hoc Networking and Computing (ACM MOBIHOC)*, June 2003.

[13] J.L. Hennessy and D.A. Patterson, *Computer Architecture - A Quantitative Approach, Third Edition,* Morgan Kaufmann, 2002.

[14] A. Silberschatz, P.B. Galvin, and G. Gagne, *Operating System Concepts, Sixth Edition,* John Wiley and Sons, Inc., 2001.

[15] George Coulouris, Jean Dollimore, and Tim Kindberg, *Distributed Systems: Concepts and Design,* Addison-Wesley, 2001.