A Relaxed Synchronization Primitive for Macroprogramming Systems

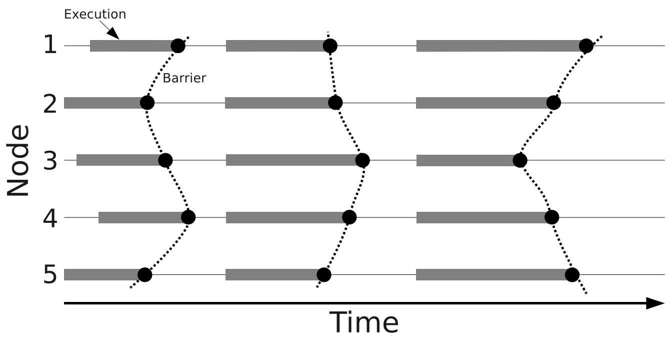
Timothy W. Hnat and Kamin Whitehouse
Department of Computer Science, University of Virginia
Charlottesville, VA, USA
{hnat,whitehouse}@cs.virginia.edu

***Abstract*— Some sensor networks have large, non-deterministic communication delays which can be problematic because nodes must decide how long to wait before acting. A conflict arises when deciding on how much information is necessary: waiting a long time will improve accuracy but is detrimental to timeliness and acting quickly will improve timeliness but worsen accuracy. We present a relaxed barrier synchronization primitive that allows the programmer to make this tradeoff. A key challenge of a relaxed barrier is correctly setting exit conditions. For example, the number of nodes, radius of influence, and deadline can be adjusted to ensure the proper tradeoff between timeliness and accuracy. We provide a solution for discovering these values that combines both simulation and hill climbing. We show that by utilizing our primitive, application accuracy can be improved and maintained for many different scenario variations.**

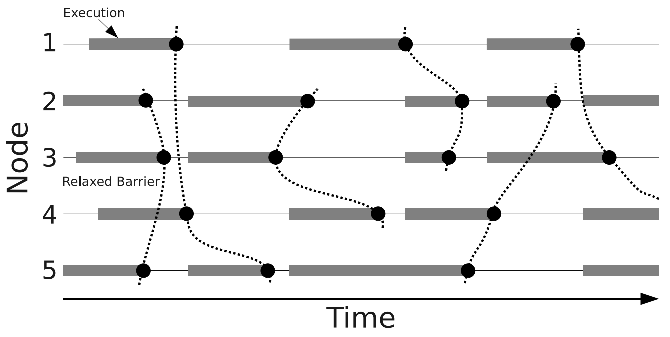
I. Introduction

Some sensor networks have large, non-deterministic communication delays due to low-power multi-hop networking, node mobility, and interference from the environment. These delays can be problematic in sensor-actuator networks because nodes must decide how long to wait for information before acting. Waiting too long is detrimental to *timeliness*: the ability of an application to respond fast enough to cause the actuation at the correct time. Not waiting long enough is detrimental to *accuracy*: a measure of error between real-world information and an application's perception of the environment. For example, if a Wireless Embedded Network (WEN) is tracking an object through a field in order to focus a camera on it then the timeliness with which it reports an object's position affects the system accuracy. If nodes report too slow, the target has moved away from the sensed location and results in a higher error. On the other hand, nodes that report too quickly could be reporting a position that is less accurate. In these cases, the timeliness and accuracy of the decision are a function of how much information is available to the nodes.

Existing solutions typically fall in two categories. First, *full synchronization*, where nodes wait until all have arrived at a barrier before continuing, is one solution but only addresses the accuracy of execution by ensuring that decisions are made with all available data. *Asynchronous execution*, where all nodes run independently, is a common solution in wireless sensor networks and typically optimizes timeliness of execution by allowing nodes to run concurrently. Deploying a full barrier in a WEN would result in failed execution due to its scale and unreliable wireless communications typically results in the majority of these primitives failing to complete. In contrast to most distributed systems, WENs typically do not employ process synchronization and run asynchronously, relying on eventual consistency and data sharing for application execution. There are situations, such as a tracking application, where it is desirable to control timeliness in order to optimize accuracy.



(a) Complete Barrier Synchronization: Nodes execute until each one reaches the barrier.



(b) Relaxed Barrier Synchronization: Nodes are allowed to continue when a majority have reached the barrier.

Fig. 1. The grey bars represent executing code on the nodes and the black dots represent barrier statements. The dotted lines show which nodes participate in each barrier.

We implement a *relaxed barrier* (Figure 1), which is like a traditional barrier except that it has different *exit conditions*: a condition that must be satisfied before a node can continue. Relaxed barriers operate along the continuum between full synchronization and asynchronous execution. In this paper, we explore two different exit conditions that affect timeliness and accuracy: the number of nodes that participate in the barrier and the deadline, although many others are possible. We also include a radius parameter for the relaxed barrier which allows it to restrict the nodes that are allowed to participate. Both the exit conditions and the spatial restriction are input parameters and a key challenge is setting them. We address this by providing two methods: (1) simulation is utilized to explore the parameter space in order to determine the best execution of the application, and (2) a hill climbing approach is utilized to facilitate simulation and run-time optimization of the parameters. These two approaches are complementary to each other and can be used to determine the best execution approach.

We evaluate our relaxed barrier on five variations of an Object Tracking Application (OTA), a wireless embedded network problem where a system of sensor nodes cooperate by sending messages to locate and focus a camera on a moving object [25]. Trials are run in a simulation environment and we show that for each variation, optimal parameter states vary and can be disjointed. In all cases, the tracking error is extremely high, greater than 20 meters, for both full synchronization (traditional barrier) and no synchronization. The standard wireless sensor network solution for this application also has a larger error, between 8 and 17 meters. Relaxed barrier is able to find and configure the system to minimize the tracking error, between 3 and 10 meters, for all scenarios.

Relaxed barriers implement serialization for a macroprogramming system called *MacroLab* [13] and provide a relaxed synchronization primitive that the programmer can control from an application or directly set the parameters. The results yield a solution that is closer to optimal than typical approaches. However, the primitive is not restricted to MacroLab and can be applied to other macroprogramming systems. Microprogramming systems can also utilize the relaxed barrier semantics; however, there is a large burden on the programmer to ensure correct execution.

II. Background and Related Work

A number of data synchronization abstractions have been proposed [13], [25] where the abstractions allow the programmer to control how the data is shared between devices. This is a version of a data coherence problem and relaxed barriers address process synchronization, a more fundamental piece of synchronization on top of which data synchronization can be built.

Typical process synchronization involves a number of techniques. For example: Barrier [20], Lock/Semaphore [7], Mutex, and Monitors [14] are only a few of the many possible primitives [9]–[11], [15], [18], [21], [22]. Process synchronization allows many threads – devices in our case – to complete a task and still maintain some type of correct runtime ordering. These approaches are well understood in the context of distributed computing; however, once we move into the world of WENs, physical constraints and timeliness make many extremely expensive or impractical to utilize.

The problem with traditional distributed computing synchronization primitives is that they do not allow any flexibility between the two extremes. Relaxed barriers are designed to fill this gap and allow the programmer the ability to control how and when the primitive exits. One paper [2] defines a loose barrier to be one that has a controlled release, although the semantics still require that all nodes participate. Traditional WEN applications fall into the no synchronization end of the spectrum and are designed in such a way that the system operates correctly without any explicit node-to-node synchronization. They mainly do so by only sharing information with their neighbors and all computation is done based on what information they have [5]. At the other end of the spectrum, full synchronization, one project [17] utilizes a two-phase locking scheme to lock variables and thus process synchronization. These types of approaches do not scale as the network grows. Relaxed barrier provides a way to vary synchronization point anywhere in between the extremes. It allows the programmer the ability to adjust parameters in order to obtain the desired synchronization behavior.

Node failure and message loss are just another aspect of WENs. Many programming abstractions fail to address these concerns. It is either assumed that the nodes will be there or the system can deliver the messages eventually. The problem with these two assumptions is that nodes will die and messages are lost [23]. A relaxed barrier is designed to handle message loss and node failure gracefully. It does so by allowing the programmer to specify parameters that complete without all nodes participating.

Traditional distributed systems can utilize a middle-ware like Map/Reduce [6] in order to handle node failures. The overhead associated with these middle-ware software layers is simply too large for an implementation on mote-class devices. Mmiddle-ware guarantees that the execution of any failed process will be restarted on another device and the resulting execution will still conform to the correct behavior. In WENs, it is simply not possible to move the computation to another node and we must have techniques that are capable of running without the information. Relaxed barrier addresses this issue by providing the programmer with a way to relax synchronization in order to address the node and message failures.

One of the problems with a barrier implementation is that it requires all nodes to participate; otherwise, we would not be true to the semantics of the synchronization primitive. Typical WEN applications are programmed without any type of synchronization. In order to apply synchronization primitives to WEN applications, we had to overcome problems with both the number of nodes and the communication channel failures. Relaxed barrier is our solution to these problems. It is designed to not only solve the problems, but it still allows the programmer the flexibility to tune their application performance and correctness based on three different criteria. The primitive is designed to be software-modifiable and can be changed dynamically on a running system in order to adapt to real-world changes.

Macroprogramming systems addresses the difficult problem of how to program a system of devices to perform a global task without forcing the programmer to develop device-specific implementations. Many macroprogramming systems have been proposed [3], [4], [12], [13], [17], [24], but none of these systems properly address the process synchronization between nodes. All these systems are good at what they have been designed to do: they make it easier for the programmer. However, the programmer still needs to understand the distributed nature of the system and account for unsynchronized execution. Mainstream acceptance of high-level macroprogramming abstractions requires that not only are they easy to understand and use, but they provide the notion of *process synchronization*. Without process synchronization, these abstractions make it difficult to reason about what the code will be doing. Relaxed barrier fills this gap between what macroprogramming abstractions are currently providing and what is needed.

III. Relaxed Barrier

The goal of a relaxed barrier is to provide an easily tunable primitive that allows the user to exploit spatial locality and concurrency of WEN nodes. By taking advantage of timeliness, the programmer has the ability to adjust the way their application runs. The key challenge for a relaxed barrier primitive is how to tune its exit conditions. Two different exit conditions (*nodes* and *deadline*) are explored as well as a spatial restriction (*radius*). These three parameters allow a relaxed barrier to be tuned for timeliness and accuracy. We provide two different solutions for tuning these parameters: *brute force search* enumerates and tests all possible parameters and *hill climbing*, which can quickly and efficiently tune the system during run-time.

We have designed the relaxed barrier to work with macroprogramming systems such as *MacroLab*, which provides a vector-based syntax similar to Matlab [1]. All data objects and sensors are abstracted as *macrovectors* in order to provide an easy-to-understand representation of the application. Programmers write code based on vector operations which are decomposed by a compiler into microprograms in nesC [8] that run on mote-class devices.

The relaxed barrier primitive is not restricted to macroprogramming systems. It can be useful for microprogramming systems such as TinyOS [19]. The difficulty then lies in the implementation. With macroprogramming systems, the compiler handles all node level details , while with microprogramming, the programmer has to handle these details.

Figure 2 shows an example of a MacroLab program that contains the relaxed barrier primitive and will be used throughout the rest of this paper. The algorithm waits for neighborhoods with at least three nodes that have detected an object and focus the camera on the computed location reported by the leader node. This application deviates from the traditional implementation with the inclusion of relaxed barrier. Line 8, the relaxed barrier primitive, accepts four arguments. First, it accepts the number of nodes required in order to exit the barrier. Next, the deadline for the primitive is specified with the radius of influence from which the node-local primitive will accept nodes. Finally, a fitness function is specified in order to allow optimizations to occur. It processes the information from the relaxed barrier and by combining existing data from the application, it computes an estimated error. This error is minimized during automatic optimization.

```
1  motes = RTS.getMotes('type', 'tmote')
2  magSensors = SVector(motes, 'magnetometer')
3  magVals = Macrovector(motes)
4  neighborMag = nReflection(motes, magVals)
5  every(1)
6     magVals = magSensors.sense()
7     RB(NODES,DEADLINE,RADIUS,fitness)
8     active = find(sum(neighborMag>500,2)>3)
9     maxNeighbor = max(neighborMag, 2)
10    leaders = find($\ldots$
11           maxNeighbor(active)==magVal(active))
12    pos = weightedAverage(neighborMag)
13    if leaders
14      focusCamera(pos)
15    end
16 end
```

Fig. 2. MacroLab code for a WEN that tracks an object. Every 1ms, nodes take a reading from their magnetometer and share the value with their neighbors. Nodes block on the relaxed barrier primitive and wait for the other nodes to report their data. If more than three nodes in a neighborhood sense a magnetometer value above a threshold, a leader is elected and a camera is focused on it.

Unlike the expected semantics of a barrier primitive, relaxed barrier allows each individual node to release when its own conditions are met. This means that all nodes will release independent of all other nodes and parallels the asynchronous execution behavior of MacroLab applications. The macroprogram appears to have the semantics that a traditional barrier would have, but the programmer understands that the relaxation of the parameters will result in asynchronous process execution. The sole purpose in allowing the relaxation of a barrier primitive is to allow the programmer control over the timeliness of their code's execution. Unlike traditional distributed systems, our application space requires that we interact with physical sensors and the physical world. In scientific data processing, a barrier is very useful to synchronize long-running simulations in order to obtain the correct result. However, these applications do not have any real-time or real world constraints on their execution. On the other hand, WENs are reacting to physical stimulus and therefore, the timeliness of execution can affect accuracy. Operating in the real world allows us to make some assumptions about the inputs to a WEN system. First, physical events will not randomly appear or move within the confines of the sensor network and second, relaxed barriers will only be applied to scenarios where the goal is to sense a physical event.

Relaxed barriers provide a way for a programmer to adjust synchronization for the best application semantics. If strict

```
RelaxedBarrier(nodes,deadline,radius)
   barrierState = {}
   while True:
       for n in incomingBarrierState:
           d = math.sqrt((x-n.x)^2+(y-n.y)^2)
           if d < radius:
               barrierState[self.id] = (n.data
                   [1][2],now())
       if len(barrierState) >= nodes:
           break
       if (now() - barrierTime) > deadline:
           break
```

Fig. 3. Relaxed barrier pseudo-code: this provides the basic idea of how the relaxed barrier functions.

semantics are desired, the relaxed barrier can be set to wait for all nodes before continuing, thus making it equivalent to a traditional barrier. However, WEN application may not need complete information about the network before continuing. By requiring only nodes that appear within certain distance of an event to participate and not the entire network, performance and correctness could be improved. Programmers should aware of the semantics of the relaxed barrier and how its parameters affect execution in the real world.

The interface to a relaxed barrier is simple. It consists of a blocking function call (Figure 3) that takes three parameters and only returns once an exit condition has been satisfied. The devices we will execute this primitive on are all single-threaded. A single thread of execution would normally be a problem for any type of blocking call. This is not the case because during the execution of relaxed barrier, it will periodically transmit messages to the other nodes in the network. Since these messages need a sink location, a compiler generates code for each node that updates synchronization variables. This frees us of the requirement to either multi-thread the system or do some special encoding of state in order to manage this barrier.

A. Exit Conditions and Spatial Limits

A barrier primitive must always include an exit clause or condition. The traditional exit condition for a barrier is a response from all nodes that they have reached their local barrier and are waiting for the continuation command. Relaxed barrier extends this concept to allow for a wide variety of exit conditions. We examine two relevant exit conditions and a spatial locality restriction for the WEN applications, although many others are possible.

1) Nodes: The number of *nodes* is a primary driving factor for the control of process synchronization. It allows the programmer to specify how many nodes need to reach the barrier position before continuing. When relaxed barrier has recorded that enough nodes have reported information, it exits, allowing execution flow to continue. It is important to note that, unlike traditional synchronization barriers, the only process that is allowed to continue is the one on the current node. The other nodes will have to wait for more information.

Another way to look at the *nodes* parameter is to compute a *percentage* of nodes required from the total nodes deployed. This allows the programmer to say: *Only proceed if 10 percent of the nodes have reported*. The number of nodes can also be used to control the amount of data sharing occurring between nodes. For example, in MacroLab, the degree that a reflected macrovector can share data could be controlled by the *nodes* exit condition of relaxed barrier.

2) Deadline: Deadlines are useful because we are targeting WENs as the application domain. WENs have two properties that make deadlines an important parameter. First, the scale of a typical system is too large to deploy efficient complete synchronization, and second, wireless communication is prone to message loss and corruption. Having hundreds or thousands of nodes participating with these two attributes will mean that a complete barrier will have a very large latency at every occurrence. By utilizing a deadline on the barrier, we allow the applications to continue executing without meeting their node requirements.

The effectiveness of deadlines is illustrated in the OTA scenario that includes a fast-moving object. Fast-moving tracked objects necessitate a quicker response from the system. By introducing a deadline to the primitive, we have provided a way to force relaxed barrier to continue even without meeting any of its other conditions.

3) Radius: Spatial locality is an important component when implementing relaxed barrier for WENs. Many applications would benefit by restricting the synchronization primitive to only operate over those nodes that are nearby. This is because physical events are located somewhere in space. They cannot happen over the whole system at the same time or at random points within the system. Relevant information is usually obtained from nodes that are nearby. This spatially-isolated property of physical events lends itself well for supporting a radius or distance limitation on the barrier.

For example, a good application of the radius limitation is applying the OTA application to multiple simultaneous targets. With a single target, the radius does not matter much; however, multiple targets would confuse the application if the range was set too large. In this case, choosing the radius is an important aspect to correctly and efficiently tracking targets. It needs to be as small as possible, yet still be large enough to capture enough nodes and provide an accurate location estimation.

We have discussed the three different parameters to our new relaxed barrier primitive for WENs. All three provide a different set of features that have a complex set of interactions depending on the application scenarios deployed. Each scenario will have a different set of optimal parameters and these parameters are hard to tune correctly without some type of simulation or real deployment. Relaxed barrier fills a gap between the two types of existing synchronization. It provides a way to trade off timeliness and application error. This is a critical piece of the programming abstraction when writing applications at the macroprogramming level.

```
distance = 500
bestError = Inf
base = [100 100 100];
while distance > 1:
  count = count + 1
  x0 = base + distance*rand(-1,1)
  error = simulate(x0)
  if error < bestError:
    bestError = error
    base = x0
  if mod(count,5) == 0
    distance = distance / 2
```

Fig. 4. Pseudo-code of a hill climbing algorithm

B. Brute Force Search

To produce an optimal solution, a brute force search is utilized to compute the error estimates for all possible parameter configurations, nodes, radius, and deadline for each OTA scenario.The drawback to performing the search is two-fold. First, the exact application scenario, including the deployment topology and the target locations and paths, must be known a priori and second, even on modern processors, the simulation will take a long time to complete.

C. Hill Climbing

In contrast to a brute force search, hill climbing adapts to deployment scenario and does not require the user to know the details in advance. The tradeoff is that hill climbing requires a period of time where it learns the parameters. In general, hill climbing has the tendency of getting stuck in local minimums or maximums; however, we exploit the relatively smooth variations of error through out this state space to efficiently utilize this algorithm.

Our variant of hill climbing (Figure 4) starts by choosing a random starting location from a valid set of parameters. It then computes five random locations that are within a specified distance from this starting location. Each location is evaluated in simulation and the location with the smallest error is chosen as the new location. The distance is then cut in half and the process is repeated until convergence. This a similar approach to simulated-annealing [16].

IV. Experimental Setup

Our evaluation is composed of a discrete event simulation that executes the logic of the macroprogram on 100 simultaneous nodes. The simulations explore the state space in all three dimensions to determine the optimal parameter choices for each scenario. Each trial runs with the same random initializations to keep the run-to-run comparison as fair as possible. Tracked targets move along a random 25 waypoint route and the simulation is complete when the target reaches the last waypoint. The topology (Figure 5) is randomly generated on a 100×100 meter area with all nodes connected through a single hop.

We evaluate relaxed barriers with five different scenarios of the OTA code presented earlier (Figure 2). The simulation environment is responsible for running the application logic, moving the tracked targets, and evaluating the correctness of the running application by utilizing the reported locations from the application code and comparing it with the actual target location resulting in an error distance. Sensing was emulated with virtual sensors that report a noisy estimate of the target distance.

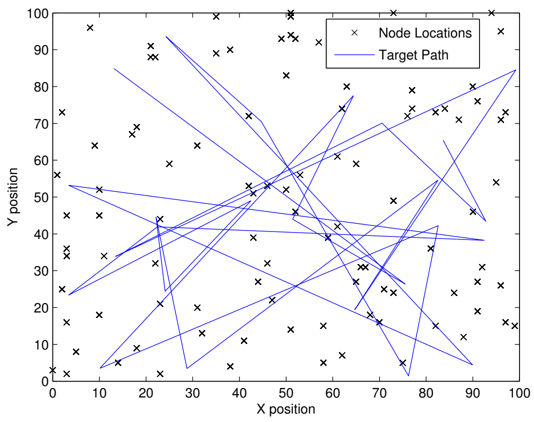


Fig. 5. Overview of the node deployment locations and an example target movement track.

A. Application Scenarios

(1) A **single target** OTA implementation is the typical scenario used by researchers. It involves a single target navigating through the sensor field. Application code senses the target and communicates the sensor readings to other nodes. A single target makes the parameters of a relaxed barrier robust to deployment variations.

(2) **Two targets** is a variation of the single target scenario that includes an additional target. An interesting aspect to adding another target is ensuring that relaxed barrier only operates on relevant sets of nodes. The *radius* parameter is the primary controlling parameter in separating out the two targets and ensuring that the primitives maintain separation as much as possible.

(3) The **fast-moving target** scenario increases the speed of a tracked target. This has the effect of limiting the amount of time and number of sensor samples obtained from a particular node during the tracking period. The high rate of speed also demonstrates the *deadline* property of the relaxed barrier primitive. Without a deadline, the synchronization primitive would be in a locked state and report information with poor timeliness.

(4) A **high-density deployment** is an example of what would happen if the sensors were deployed with a larger sensing range and able to track the single target from a greater distance. This type of deployment will become more prevalent in the future as more systems are deployed in diverse environments.

(5) Introducing a **false positive** rate is our final application scenario and all of the sensing nodes include a 1 percent

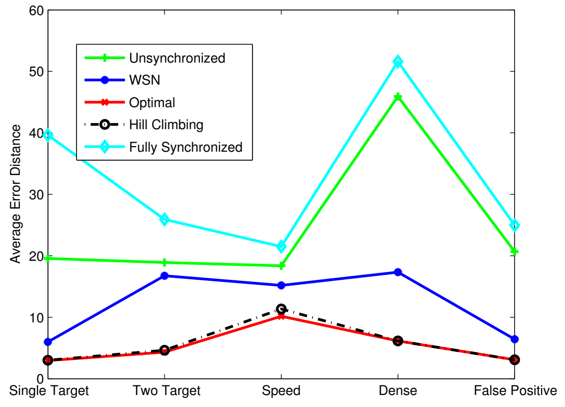


Fig. 6. Average error for five different versions of the OTA application. Each instance contains five different results: unsynchronized, WEN approach, relaxed barrier, optimal, and full synchronization. *Relaxed barrier* shows superior performance in all cases.

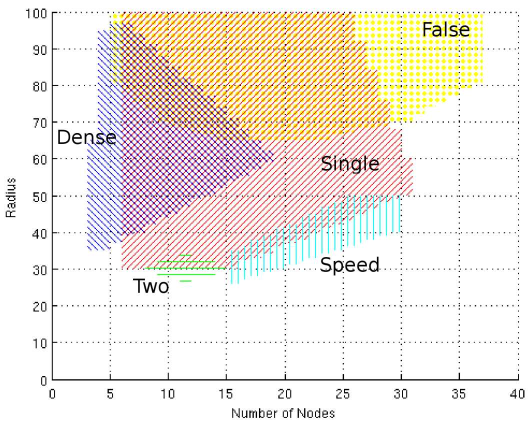


Fig. 7. State-space showing different optimal parameter values for both the radius and number of nodes in all scenarios.

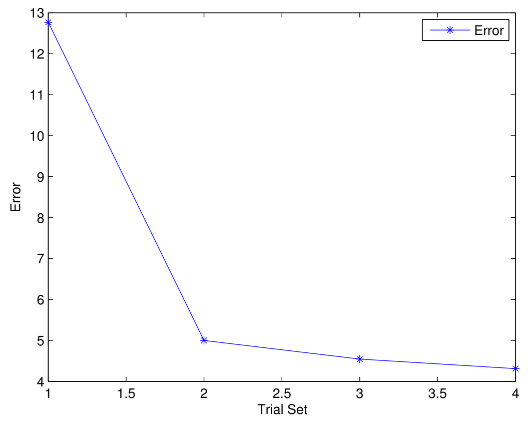


Fig. 8. Hill climbing convergence with four separate optimizing trials. Trials encompass 10 simulations and take 33 hours to run.

false positive report from their simulated sensors. This serves to model the real world better when the sensors are not perfect and the environment is dynamic. These five application variations will be used to evaluate the effectiveness of relaxed barrier in the next section.

B. Implementations

There are five implementations that generate values of interest in each scenario. First, there is the case of no synchronization. This shows us what would have happened with the current implementation of MacroLab. Second, the effects of full synchronization are examined. These two values represent the limits of relaxed barriers and represent common programming models for many distributed systems.

Another common approach, which is taken by many OTA application developers, is to wait until four neighbors report sensing values before proceeding. Finally, we evaluate our hill climbing approach in order to determine proper parameters as well as perform a brute force search of the parameter space to determine optimal values.

V. Evaluation

Our system evaluation is composed of two parts. First, we will show the effectiveness of a relaxed barrier on the tracking error for each variation. Then, we will show how hill climbing can be done efficiently.

A. Accuracy of Tracking

Figure 6 shows the accuracy gains that we can obtain by applying a relaxed barrier to the scenarios. In all cases, the relaxed barrier can perform as good or better than a typical WEN approach and much better than both the unsynchronized and full synchronization cases. Our simulation allows us to exhaustively examine the parameters and choose the best options for each case. The errors shown are the average tracking error computed from ground truth over the course of the entire simulation.

It is not sufficient to show that our primitive performs better than existing options. While this is a good result, an important component to why these results are better lies with the optimum parameters . Figure 7 shows optimal areas to assign two of the parameters: nodes and radius. These areas are defined to be within two feet from the absolute best option obtained through the brute force search. Many of the scenarios have disjointed optimal parameters and this further motivates the need for an adjustable synchronization primitive.

B. Hill Climbing Convergence

Figure 8 illustrates a run of the hill climbing algorithm on one of the scenarios. Each of the trials run for approximately 12,000 seconds. There were four separate minimizing steps taken over 36 runs and the results converge to the minimum solution produced by the brute force search simulation. Convergence takes up to five days of running time to converge on

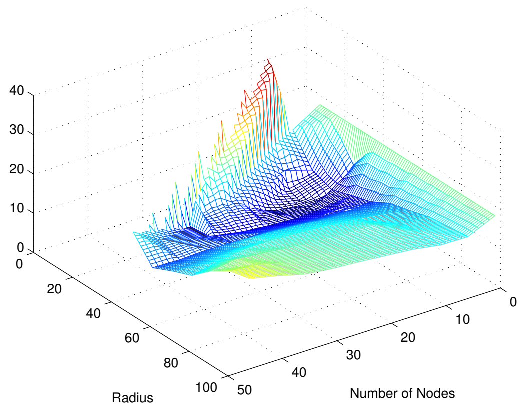


Fig. 9. Two-dimensional surface showcasing the convex, bowl-like shape the parameter space forms.

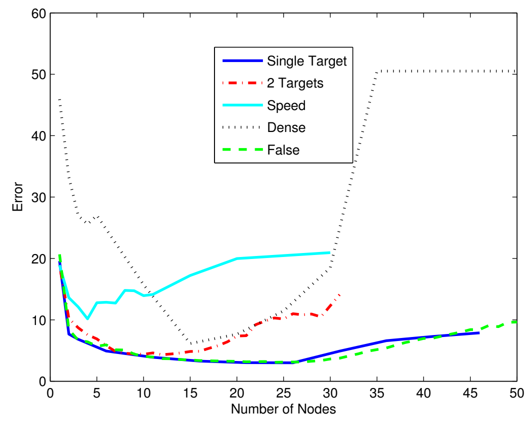


Fig. 10. The effects of varying the number of nodes that participate in the relaxed barrier. All of the scenarios form convex curves and have different points where they are minimized.

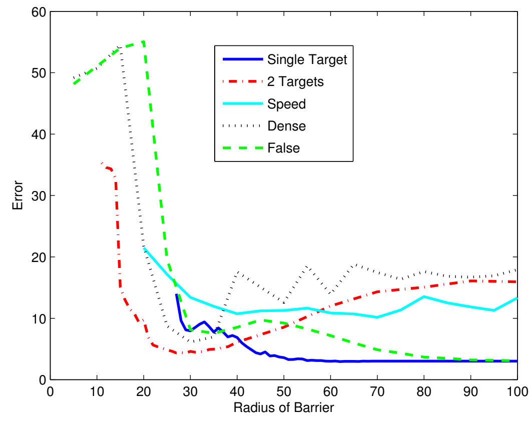


Fig. 11. The effects of varying the radius of influence on the relaxed barrier. All of the scenarios have different points where they are minimized.

the minimum solution; however, after less than ten runs (trial set 2), the error is close to what the optimal value will be. The randomness and decay of the step distance attempts to avoid any local minimums. If a local minimum has been reached, re-running the algorithm is the most effective way to resolve this error.

Because of the convex nature of the individual parameter space, hill climbing functions optimally and efficiently. There are very few occurrences where it will hit a local minimum and stop. This quick convergence means that the learning time for the application deployment is minimized and changes to the real-world deployment can be adapted to quickly. The advantage to adjusting the algorithm in the field is that one does not have to redo the brute force search and adjustments typically will only alter the optimal location a little and therefore quickly converge.

An important aspect to the effectiveness of the hill climbing approach is the ability to determine where the optimal parameter space is located. Figure 9 shows a convex, bowl-like shape that is formed with data from the brute force search. The convex shape of the parameter space plays an important role for the hill climbing algorithm. The system's goal is to minimize the error of tracking and it clearly illustrates this error is poor around the edges of the shape. Next, we will examine the three parameters for the relaxed barrier.

The first parameter, shown in Figure 10, is the effect of the number of nodes participating in a relaxed barrier. There are two key points for this set of results. First, the minimum error for each scenario occurs with different numbers of nodes and second, the curves are convex and the error results are poor at both ends. These poor results are because of either too little data (good timeliness) or too high of a latency (too many nodes).

The radius is the next parameter to examine. Figure 11 shows the performance gains that result from adjusting the radius for each scenario. Some of these curves form the desired convex shape, while others simply decrease until they approach their respective minimums. Continuously minimizing functions still allow us to determine where the proper choice of a parameter is located by looking for where the error first reaches a sufficiently low value. Small radius values typically results in large errors. This is usually due to a node being forced to wait for the object to move further away and into other nodes' sensing regions before releasing its barrier. The radius is a parameter that limits the number of nodes that can participate and although it is not an exit condition, it plays a key role in determining the optimal configuration of a relaxed barrier.

The final parameter is a deadline for the relaxed barrier primitive. Figure 12 shows the error produced by the five scenarios while modifying the deadline values. Four of the scenarios produce very little variations because of the deadline. However, the *speed* scenario has a considerable variation in error. This is because the speed of the target is fast enough to cause the deadline parameter to contribute to the error. We found that many scenarios have little variation in error but

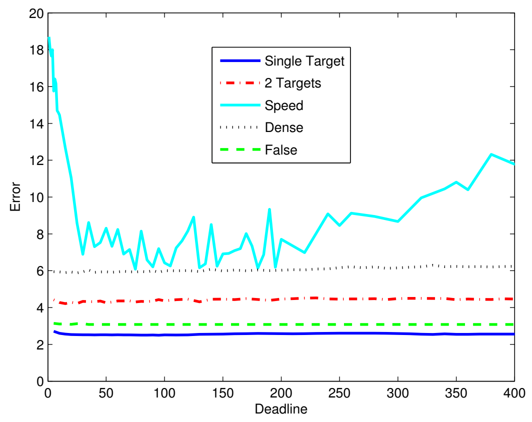


Fig. 12. The effects of varying the deadline on tracking error. The speed scenario is the only one of our examples that has a significant effect on the error. Here there is an optimal deadline because the target is moving quickly. The other examples have little error and are not interesting.

when there is variation, it can be significant.

The evaluations of the three different parameters of the relaxed barrier show that in all scenarios the optimized parameters space is different. It also shows that the curves are all convex and therefore a hill climbing approach is applicable for online optimization.

VI. Conclusion

Relaxed barriers makes programming robust WEN applications easier by providing a way to control timeliness and accuracy. This is a contrast to traditional approaches in both distributed systems and sensor networks. Programming paradigms traditionally fall into two extremes: unsynchronized and synchronized. Relaxed barriers allow the programmer to choose an amount of process synchronization and data sharing that occurs while running the application.

We present two examples of relaxed barrier exit conditions and one parameter that restrict the spatial locality. The two exit conditions, *nodes* and *deadline* along with *radius*, are the three input parameters to the relaxed barrier and directly affect timeliness and accuracy.

Hill climbing and brute force search provide a way to optimize the parameters for the variety of scenarios. The results show that the parameter space is convex, thus allowing the use of an efficient hill climbing approach. This approach typically converges on the optimal solution in less than 50 trials and is useful in real-world deployments.

Although the relaxed barrier primitive was developed for the macroprogramming language MacroLab, it would be just as effective with microprogramming systems such as TinyOS. The programmer would be responsible for writing applications in a form that can take advantage of the primitive but the principles still apply.

The future of WENs will involve a large number of devices placed throughout the physical world and require programming technologies to develop and run these systems efficiently. Programming abstractions need to remove as much complexity as possible, while still maintaining a level of control that allow applications to run correctly and efficiently. We believe that timeliness is a critical piece for correct and efficient execution and a relaxed barrier primitive can address the challenge.

References

[1] Matlab - the language of technical computing. http://www.mathworks.com/products/matlab/.

[2] J. Albrecht, C. Tuttle, A. Snoeren, and A. Vahdat. Loose synchronization for large-scale networked systems. In *Proceedings of the USENIX Annual Technical Conference (USENIX)*, 2006.

[3] A. Awan, S. Jagannathan, and A. Grama. Macroprogramming heterogeneous sensor networks using cosmos. *SIGOPS*, 2007.

[4] J. Bachrach and J. Beal. Programming a sensor network as an amorphous medium. *DCOSS*, 2006.

[5] C. Basile, K. Whisnant, Z. Kalbarczyk, and R. Iyer. Loose synchronization of multithreaded replicas. In *Reliable Distributed Systems, IEEE Symposium on*, volume 0, page 250, Los Alamitos, CA, USA, 2002. IEEE Computer Society.

[6] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *OSDI*, 2004.

[7] E. W. Dijkstra. The structure of the "the"-multiprogramming system. In *SOSP '67: Proceedings of the first ACM symposium on Operating System Principles*, pages 10.1–10.6, New York, NY, USA, 1967. ACM.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI*, 2003.

[9] A. J. Gerber. Process synchronization by counter variables. *SIGOPS Oper. Syst. Rev.*, 11(4):6–17, 1977.

[10] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, 1989.

[11] J. R. Goodman, M. K. Vernon, and P. J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. *SIGARCH Comput. Archit. News*, 17(2):64–75, 1989.

[12] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using Kairos. In *DCOSS*, 2005.

[13] T. Hnat, T. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys*, 2008.

[14] C. A. R. Hoare. Monitors: an operating system structuring concept. *Commun. ACM*, 17(10):549–557, 1974.

[15] C. A. R. Hoare and C. A. R. Hoare. Communicating sequential processes. *Communications of the ACM*, 21:666–677, 1985.

[16] S. Kirkpatrick. Optimization by simulated annealing: Quantitative studies. *Journal of Statistical Physics*, 34(5):975–986, 1984.

[17] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI*, 2007.

[18] B. Lampson and D. Redell. Experience with processes and monitors in Mesa. 1980.

[19] Levis, P., Madden, S., Polastre, J., Szewczyk, R., Whitehouse, K., Woo, A., Gay, D., Hill, J., Welsh, M., Brewer, E., et al. Tinyos: An operating system for sensor networks. *Ambient Intelligence '05*.

[20] B. Lubachevsky. Synchronization barrier and related tools for shared memory parallel programming. *International Journal of Parallel Programming*, 19(3):225–250, 1990.

[21] D. P. Reed and R. K. Kanodia. Synchronization with eventcounts and sequencers. *Commun. ACM*, 22(2):115–123, 1979.

[22] G. Schlageter. Process synchronization in database systems. *ACM Transactions on Database Systems (TODS)*, 3(3):248–271, 1978.

[23] K. Srinivasan, M. A. Kazandjieva, S. Agarwal, and P. Levis. The beta-factor: measuring wireless link burstiness. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 29–42, New York, NY, USA, 2008. ACM.

[24] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI*, 2004.

[25] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys*, 2004.