Fast Distributed Simulation of Sensor Networks using Optimistic Synchronization

Hao Jiang, Jiannan Zhai, Sally K. Wahba, Biswajit Mazumder, Jason O. Hallstrom
School of Computing, Clemson University
Email: {hjiang, jzhai, sallyw, bmazumd, jasonoh}@clemson.edu

***Abstract*—Network simulation is an important tool for testing and evaluating wireless sensor network applications. Parallel simulation strategies improve the scalability of these tools. However, achieving high performance depends on reducing the synchronization overhead among simulation processes. In this paper we present an optimistic simulation algorithm with support for backtracking and re-execution. The algorithm reduces the number of synchronization cycles to the number of transmissions in the network under test. We implement *SnapSim*, an extension to the popular Avrora simulator, based on this algorithm. The experimental results show that our prototype system improves the performance of Avrora by 2 to 10 times for typical network-centric sensor network applications, and up to three orders of magnitude for applications that use the radio infrequently.**

I. Introduction

In a typical parallel simulation system, each simulated process is executed by its own physical process. Due to the communication between simulated nodes, temporal dependencies exist between transmitters and receivers. Fig. 1 illustrates an example of such a dependency. A and B represent two simulated nodes in a network; the horizontal axis represents simulated time. Suppose that node A performs a radio transmission at simulated time tx_A. Assume that the process that simulates B runs faster than the process that simulates A. Specifically, assume that after τ_S units of physical time, node B has executed beyond tx_A, prior to node A reaching this point. Consequently, B misses the data transmission from A since it fails to read the channel at the right time. Thus, the instruction sequence at B is altered from what would be experienced in a physical run, and the simulation is erroneous. To avoid such problems, process synchronization is used to preserve causality paths between transmitters and receivers.

Consider a simulator that synchronizes each simulated node cycle by cycle; the dependency problem is easily solved. However, this strategy is inefficient, especially in distributed simulation scenarios, where synchronizations incur a physical networking delay. The state-of-the-art in parallel simulation allows simulated nodes to execute in parallel for a short interval before a synchronization is performed. The interval is calculated based on the amount of time a node can execute without missing a transmission (in any possible run). As a result, all dependencies between nodes are resolved, and no transmission will be missed. This approach is referred to as *interval synchronization*. Avrora [11], the most popular parallel simulator for sensor networks, uses this technique.

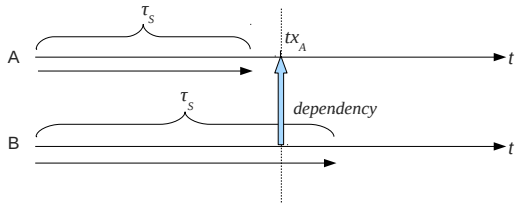

Fig. 1. Distributed Simulation Dependencies

Avrora is an instruction-level, cycle-accurate sensor network simulator capable of simulating multiple sensor nodes in parallel. It takes as input the compiled microcontroller code generated for the target hardware – the popular AVR architecture. Each simulated node is run as a separate (Java) thread. The synchronization interval is defined by the time required for the radio chip to complete the transmission of one byte of data. Fig. 2 summarizes this strategy. After each T_{radio} units of simulated time, a constant, all the nodes in the network are synchronized. Consider the following two cases: *(i)* Suppose nodes A and B are synchronized at the n_{th} synchronization point, and there is no transmission event between the n_{th} and the $(n+1)_{th}$ synchronization points. As a result, it is safe for both nodes to execute for another T_{radio} units of time. *(ii)* Suppose node A completes the transmission of a byte of data at tx_A, between the $(n+2)_{th}$ and the $(n+3)_{th}$ synchronization points. Since it will require T_{radio} time units to transmit the byte, the time at which node A starts to transmit the first bit, denoted by tb_A, must be between the $(n+1)_{th}$ and the $(n+2)_{th}$ synchronization points. Since node B is synchronized with node A at the $(n+2)_{th}$ synchronization point, it has priori knowledge of the transmission time of A. As a result, no read channel event will be missed in the simulation.

The main challenge addressed by our work is reducing the synchronization overhead. We present an optimistic distributed synchronization algorithm that uses probing execution and backtracking. Jefferson [3] has argued that general rollback is too complex and inefficient to implement. We show, however, that our design is capable of reducing the number of synchronization cycles significantly and improves performance by 2 to 10 times for representative WSN applications, compared to the interval synchronization algorithm.

The main contributions of our work are as follows: *(i)* We present an optimistic synchronization algorithm that abandons interval synchronization in favor of probing execution and backtracking. The number of synchronization cycles is reduced to an optimal number. *(ii)* We implement *SnapSim*, an exten-

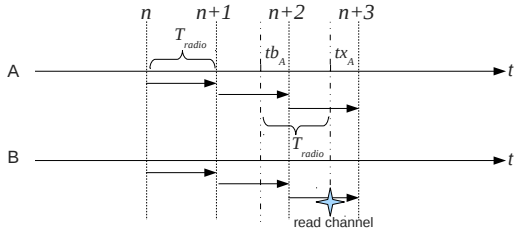

Fig. 2. The Avrora Synchronization Strategy

sion to the popular Avrora simulator, based on this algorithm. *(iii)* Finally, we evaluate the performance of SnapSim using a series of standard sensor network applications from the TinyOS [9] distribution. The results show that for typical sensor network applications, SnapSim is significantly faster than the state-of-the-art simulator.

II. Related Work

Legedza et al. [7] present two techniques to reduce the overhead associated with interval synchronization in a parallel simulation environment. The first technique, *local barriers*, provisions adjacent processes in a simulated network on the same processor, which reduces inter-process communication. The second technique, *predictive barrier scheduling*, predicts the amount of time that a process will not communicate using a combination of compile-time and runtime analysis. This enables the application of longer synchronization intervals. However, these two strategies are evaluated using several simple cases, not a full-scale sensor network application.

Jin and Gupta [5] describe *PolarLite*, a distributed simulation framework built on Avrora, which uses a synchronization strategy similar to predictive barrier scheduling, and hence improves simulation speed. Within the context of Avrora, the authors introduce two longer synchronization intervals based on radio initialization time and MAC backoff time. Radio initialization time is defined as the the shortest interval required to turn the radio back on when the radio is off; it is 11 times the interval constant T_{radio}. MAC backoff time is defined as the randomized backoff time introduced at the MAC-layer when a collision is detected; it is 1 to 32 times the interval constant T_{radio}. Introducing these longer intervals, a simulated node can proceed for a longer period of time without synchronization. However, PolarLite uses the new synchronization strategy only if one of the above scenarios occurs; otherwise it uses Avrora's constant interval synchronization strategy.

Before their work on PolarLite, Jin and Gupta [4] improved the interval synchronization strategy by leveraging sleep time on sensor nodes. Transmission and reception are disabled when the microcontroller and the radio chip are in deep sleep states, and most interrupts are disabled. In this case, the wake-up time can be predicted, hence longer synchronization intervals can be used. However, this strategy only works for a few special cases; it is not a general solution.

Wen et al. [12] introduce *DiSenS*, a distributed cycle-accurate sensor network simulator, which uses the local barriers technique introduced in [11], as well as interval synchronization. Each node periodically broadcasts its clock value; a node which reads the channel waits until all other (neighboring) nodes reach the same point. DiSenS partitions simulated nodes based on neighborhood connectivity. The nodes within each strongly-connected sub-graph are simulated on the same processor. The effectiveness of the partitioning algorithm is largely dependent on the topology of the simulated network.

Levis et al. [8] introduce *TOSSIM*, a simulator for TinyOS applications. TOSSIM is built into the TinyOS distribution and simulates native code by replacing hardware-dependent components with corresponding simulation components. TOSSIM is neither parallel, nor cycle-accurate. Landsiedel et al. [6] improve the fidelity of TOSSIM by estimating the cycles consumed by each source code statement; the estimate is not accurate. Shnayder et al. [10] describe an extension for simulating per-node power consumption. The extension consists of four components. The first instruments the simulated application to determine the power consumption of hardware peripherals. The second estimates the cycles consumed by each node. The third estimates per-node energy consumption based on a consumption model that relies on data collected from the first two components. Finally, the fourth visualizes power consumption.

Bajaj et al. [2] introduce *GloMoSim*, a library for parallel simulation of large-scale heterogeneous networks. GloMoSim mirrors the OSI model. To support rapid parallel simulation, each processor simulates multiple geographically-located nodes. A variant of local barriers, the approach reduces inter-processor message passing.

Unlike the work above, our work centers on an *optimistic* synchronization strategy in the absence of fixed intervals.

III. Algorithm Design

While interval synchronization improves performance significantly, it is still costly, especially for distributed simulators, where delays in the physical network may be significant. For example, T_{radio} is 3072 clock cycles for a Mica2 node; its microcontroller, an ATMega128, runs at 7,372,800 Hz. Thus, 2400 synchronizations are performed per simulated second. If the physical networking delay is 200 microseconds per message, there will be at least 480 milliseconds of synchronization delay introduced per simulated second! Although radio initialization time and MAC backoff time can be exploited to reduce synchronization frequency, it is still not the most efficient strategy. First, the synchronization interval remains small – 11 times T_{radio} for the radio off interval, and 1 to 32 times T_{radio} for the MAC backoff interval. Second, these strategies are performed only if the radio is turned off or message congestion is experienced in the MAC layer.

We reconsider the synchronization problem, defining an *optimistic simulation* as a parallel simulation of the network without any synchronization. First, suppose we are given a priori knowledge of all transmissions within a simulated run. Suppose further that it is possible to dynamically set the simulation speed of each processor. For any simulated time t, we define the *global nearest transmission time* as the earliest simulated transmission across all nodes in the network at or

after time t. We define a *valid* simulation as a run in which no simulated node reaches a global nearest transmission time before the node(s) that transmit at those times. In effect, such a run imposes a partial ordering on the communication dependency graph, ensuring that all transmission events are simulated prior to their corresponding reception events.

But of course, a perfectly optimistic simulator is not possible since instruction sequences cannot be predicted statically. However, if the *next* transmission time can always be determined, the number of synchronizations can be reduced to the number of transmissions (as we discuss next).

A. Probing and Backtracking

The central algorithmic idea is to use probing executions to find the nearest transmission time for each node in the network, and to backtrack nodes that have passed the global nearest transmission time. The probing execution is isolated; communication primitives are ignored. A probing execution may proceed beyond the global nearest transmission time, thereby missing a read channel event. Thus, some nodes may need to backtrack and re-execute to correct for the missed transmission. To restore a node to a correct former state, a snapshot must be taken that includes the aggregated simulation state of the node.

We assume the following: The network consists of a fixed set of N processes. Each simulated node is emulated by a single physical process; the processes are distributed. Each process is capable of suspending and resuming execution of the simulated node. The simulation is cycle-accurate and the simulated microcontrollers have the same frequency. Every physical process maintains a local simulation clock based on the number of cycles executed. Nodes do not have global knowledge, except for the size of the network. Messages may be broadcast through the (physical) process network. Each process is capable of taking a simulation snapshot and restoring the simulated node to a previous snapshot point.

Before presenting the design details, we briefly summarize the basic principles. First, we define a *wave* of execution as the execution sequence spanning between consecutive global nearest transmission times. (We consider time 0 as the first global nearest transmission time.) Snapshots are taken at each global nearest transmission time. A basic task of the algorithm is to probe the end of the current wave and backtrack any processes which exceed the end of the wave before it was found. Those processes will be replayed, enabling the reception of otherwise lost events. Initially, each process creates a snapshot before the simulation begins. Next, each process starts a probing execution, ignoring the possibility of message reception. The minimum received time becomes a candidate for the global nearest transmission time. If the node's current time is greater than the candidate nearest transmission time, it broadcasts its current time and waits for the final global transmission time, since it is unnecessary to continue probing execution when another node has an earlier transmission event. Each process counts the number of processes it has heard from in the current wave. If the count is less than the network size, the process waits at the candidate transmission point; otherwise, the minimum candidate time becomes the global nearest transmission time. We call the last broadcast message in a wave the *notify* message; all the processes in the network resume execution when it is received. If a process transmits or waits at the global nearest transmission time, it saves a snapshot and resumes execution without backtracking. Otherwise, the process backtracks to its last snapshot at the beginning of the wave before resuming execution. When it executes to the new global nearest transmission time, it saves its snapshot and continues probing of the next wave.

To formally define the probing and backtracking algorithm, we present the following *action system* program.

Probing and Backtracking Algorithm
var
$clock.j : \mathbb{N}$ (simulated time of j)
$timeNr.j : \mathbb{N}$ (global nearest transmission time)
$timeCnd.j : \mathbb{N}$ (candidate global nearest transmission time)
$count.j : \mathbb{N}$ (probing completion count)
$transmitting.j : \mathbb{B}$ (does j transmit in the current wave?)
$wait.j : \mathbb{B}$ (is j in the wait state?)
function
$executeInst.j$: probing execution of instruction at j
$broadcastClock.j$: broadcast the current simulated time
$saveSnapshot.j$: save snapshot
$restoreSnapshot.j$: restore snapshot
$writeChannel.j$: write to (simulated wireless) channel
initially
$clock.j, timeNr.j, count.j = 0;$
$timeCnd.j = \infty;$
$transmitting.j, wait.j = false;$
assign
$!wait.j \rightarrow$
  $clock.j = timeNr.j \rightarrow saveSnapshot.j;$
  $clock.j < timeCnd.j \rightarrow executeInst.j;$
  else $\rightarrow$
    $count.j := count.j + 1;$
    $broadcastClock.j;$
    $wait.j := true;$
$\square$
simulated transmission event $\bigwedge !wait.j \rightarrow$
  $count.j := count.j + 1;$
  $timeCnd.j := min(clock.j, timeCnd.j);$
  $transmitting.j := true;$
  $broadcastClock.j;$
  $wait.j := true;$
$\square$
receive broadcast message from k $\rightarrow$
  $count.j := count.j + 1;$
  $timeCnd.j := min(time.msg_k, timeCnd.j);$
$\square$
$count.j = N \rightarrow$
  $timeNr.j := timeCnd.j;$
  $count := 0;$
  $timeCnd.j := \infty;$
  $wait.j := false;$
  $clock.j > timeNr.j \rightarrow restoreSnapshot.j;$
  else if $transmitting.j \rightarrow writeChannel.j;$
  $transmitting.j = false;$

Each process j in the algorithm maintains $clock.j$, a local counter that counts the number of cycles executed. Variable $timeNr.j$ stores the most recent global nearest transmission time, and $timeCnd.j$ stores the next candidate global

nearest transmission time, which is the minimum of all the transmission times identified within the current wave. Within each wave, each node records the number of nodes that wait at or beyond a candidate transmission time, denoted by $count.j$. Variables $transmitting.j$ and $wait.j$ are Booleans that indicate whether the node transmits within the current wave and whether the node is in a waiting state, respectively.

Five functions are introduced to simplify the presentation. *executeInst.j* executes an instruction on node j without handling reception events. The execution does not write to the wireless channel since probing execution does not guarantee the validity of the simulation. *broadcastClock.j* broadcasts the current time of simulated node j throughout the physical network. Once a node broadcasts, it transits to the *wait* state. *saveSnapshot.j* saves a snapshot of the simulated node j, and *restoreSnapshot.j* restores the simulated node j to the last stored snapshot state. *writeChannel.j* writes a byte of data to the wireless channel.

Initially, $clock.j$, $timeNr.j$, and $count.j$ are set to 0; $timeCnd.j$ is set to infinity; and $transmitting.j$ and $wait.j$ are set to false. There are four rules for this program.

The first rule checks whether the process is in the *wait* state. If not, it compares the current clock value with $timeNr.j$. If they are equal, the process has reached the global nearest transmission time and saves its snapshot. (Since $clock.j$ and $timeNr.j$ are set to 0 initially, each process will save its snapshot at the beginning of the simulation.) Next, it compares the current clock value with the candidate next global nearest transmission time, $timeCnd.j$. If the current time is earlier than the candidate time, the process executes the next instruction at node j (which increments $clock.j$). Otherwise, the process is already beyond a candidate transmission, and it increments $count.j$ by 1, broadcasts a message containing the current time of the simulated node, and sets $wait.j$ to true.

The second rule is performed when a new local transmission is identified at node j while it is not in the *wait* state. In this case, the process increments $count.j$ and updates $timeCnd.j$ if the current clock value is earlier. Next, the process sets $transmitting.j$ to true, which indicates that the node has detected a transmission point. Next, node j's clock value is broadcast to all other processes, and $wait.j$ is set to true. A node detects only one transmission and broadcasts only once within each wave (by rule 1 and 2).

The third rule is performed when a process receives a broadcast message from another process k. In this case, the process increments $count.j$ and updates $timeCnd.j$ if the received time is earlier.

The fourth rule is performed when $count.j$ is equal to the network size N, indicating that the new global nearest transmission time is globally known. As a result, $timeCnd.j$ becomes the new global nearest transmission time, $timeNr.j$; $count.j$, $timeCnd.j$, and $wait.j$ are reset. ($transmitting.j$ will also be reset, at the end of the rule.) If the current time of the simulated node is later than $timeNr.j$, it backtracks to the last saved snapshot. Otherwise, if the node has a transmission point at $timeNr.j$, it writes the data to the wireless channel. If the current time equals $timeNr.j$, a new snapshot will be saved when the first rule is selected later; no backtracking will be performed.

B. Algorithm Case Analysis

Space limitations preclude the inclusion of a formal correctness proof. Hence, to clarify the operation of the algorithm, we consider several representative scenarios. Figs. 3 – 6 show 4 basic execution scenarios in a network consisting of two nodes. Larger networks are analogous. The horizontal timeline in each graph denotes simulated time (i.e., cycle count); tx denotes the simulated time of a transmission event. The horizontal curly brackets denote physical execution time (e.g., microseconds), labeled by τ_S; Δ represents an unknown networking delay. The diagonal-filled bars denote probing execution, and the hash-filled bars denote re-execution after backtracking.

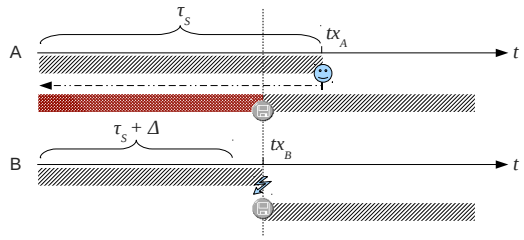

Fig. 3. Algorithm Analysis: Case 1

Consider Fig. 3. Assume that after τ_S units of physical time, process A has reached a transmission at simulated time tx_A, where it broadcasts this transmission time and waits; process B receives the broadcast at $\tau_S + \Delta$, where Δ is the networking delay from A. At this point, B has not reached its transmission time tx_B, which is smaller than tx_A. As a result, when B eventually reaches tx_B, tx_B becomes $timeNr$, and B sends the notify message since it has already received the transmission time, tx_A, from A (i.e., $count = N = 2$). Since tx_B is equal to $timeNr$, B resumes execution; node A backtracks to the last snapshot and resumes execution. A will save a snapshot when it passes tx_B.

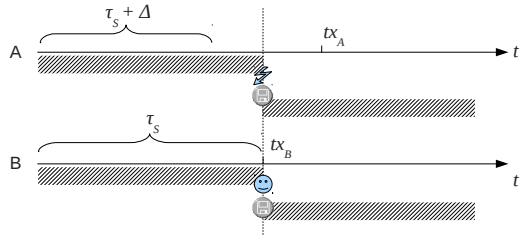

Fig. 4. Algorithm Analysis: Case 2

Next consider Fig. 4, which uses the same transmission times (i.e., tx_A is greater than tx_B). In this run, B executes to tx_B in τ_S physical time and waits. When receiving the broadcast at $\tau_S + \Delta$, A has not reached tx_B, nor its own transmission event. In this case, A continues to execute until it reaches tx_B. As a result, tx_B becomes the global nearest transmission time. Next, the notify message from A is received at B, and both A and B save snapshots and resume execution. No backtracking is necessary.

Next consider Fig. 5, again the same setting. tx_A is greater than tx_B, and in τ_S physical time, B reaches tx_B and waits.

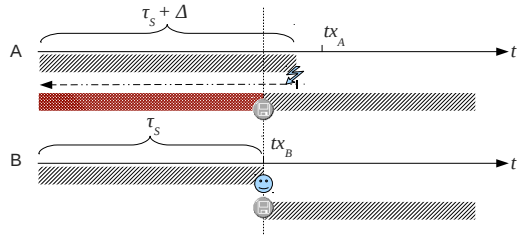

Fig. 5. Algorithm Analysis: Case 3

However, when A receives the broadcast from B, it has already passed tx_B, but has not reached its transmission time tx_A. Since A cannot find an earlier transmission time than tx_B, it stops searching, and tx_B becomes the global nearest transmission time. After A notifies B, B resumes execution; A backtracks to the last snapshot, resumes execution, and saves a new snapshot when it passes tx_B.

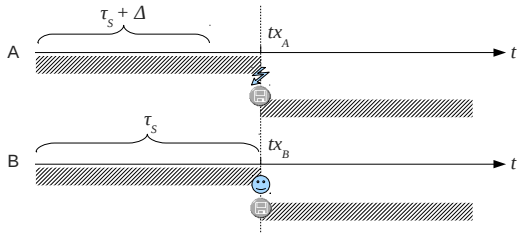

Fig. 6. Algorithm Analysis: Case 4

Fig. 6 shows another possible case. Suppose that nodes A and B transmit at the same time ($tx_A = tx_B$), and as a result, $timeNr$ is the transmission time for both nodes. No matter which node reaches $timeNr$ first, no backtracking is required since the first sender will notify the second, which will wait at tx_A/tx_B. Another possible case is that there are no transmissions in the network ($tx_A = tx_B = +\infty$). In this case, no synchronization and backtracking will be performed; the simulator will execute at its maximum speed.

The probing and backtracking algorithm is based on the idea of optimistic execution. To reduce synchronization overhead as much as possible, it probes the global nearest transmission time and backtracks nodes that have passed this time, to the last saved snapshot. New snapshots are saved at the global nearest transmission time. The number of synchronizations is limited to a minimum, assuming no prior knowledge of the global nearest transmission time. From the start of a wave, at most two rounds of execution will be performed. The algorithm compensates slower processes, since if they reach the global nearest transmission time later than the transmitting process, no backtracking will be performed at these processes. As a result, the speed of snapshotting/backtracking and the technique used to restore nodes to a valid state become the main implementation challenges.

IV. System Implementation

SnapSim is implemented using Avrora as the foundation. In our implementation, the interval synchronizer is removed, and a distributed wireless channel is implemented at each process. The channel simulates the wireless communication medium and is updated by each transmission. A vector clock is also maintained at each process and records the reported transmission time received from other nodes. Each node maintains a snapshot that contains a copy of the simulation state of the node at the last global nearest transmission time. In the following sections, we discuss key implementation details.

A. Snapshots

A snapshot includes the full simulation state of a given node; it is, conceptually, a clone of the simulation thread. A wave starts from the last snapshot time and ends at the new snapshot time. By the correctness of the algorithm, a snapshot always represents a valid simulation state. Since a node never backtracks to earlier waves, the existing snapshot is replaced when a new snapshot is saved. In detail, a snapshot in our design comprises the following state: *(i)* all state related to the microcontroller, including memory, registers, flags, etc.; *(ii)* the state of the interpreter interpreting the instructions; *(iii)* the states of all SPI and ADC devices connected to the microcontroller, including the radio; and *(iv)* the state of all auxiliary components for simulation – for instance, the event clock, device state monitor, etc. As a result, the snapshots are relatively large, and the associated checkpointing / backtracking overhead may be large if we save a full copy of the snapshot for each node. To improve efficiency, we take the following steps to minimize the time required to save and load a snapshot:

(1) Known invariants and constants are not saved in a snapshot. As an example of the former, variables that represent assembly instructions loaded on the microcontroller are invariant if the microcontroller is not reprogrammed. As a result, the snapshot size can be reduced.

(2) For peripheral devices, for instance, an SPI device, if it is untouched in the wave since the last snapshot, the previous saved state will be kept; no new save is required.

(3) For large arrays of elements, an incremental saving strategy is used since only a subset of elements may change from snapshot to snapshot. More precisely, a large array is divided into N smaller partitions, and an immaculacy bitmask of N bits tracks whether the partitions have been modified since the last snapshot. If a partition is untouched in a given wave, no save action will be performed on the partition. When a snapshot is restored, or a new snapshot is saved, the immaculacy array is reset. One example application of this strategy in our simulator concerns the array that represents the RAM of the microcontroller, which is large.

By taking the above steps to reduce the size of each snapshot, execution of the save and restore methods consumes less than 1.5% of the total running time in our evaluation.

B. Distributed Resources

In SnapSim, a vector clock is introduced at each process to maintain the local knowledge of simulated time at other processes. The vector clock is updated when a local nearest transmission time message is received or a local instruction is executed. When a node has received a broadcast message from all nodes in the network, the lowest value of the vector clock becomes the global nearest transmission time. We simulate

the communication between distributed (physical) processes by imposing an adjustable networking delay when a process broadcasts. As a result, we are able to evaluate the performance of the simulator in a range of networking scenarios.

In SnapSim, every thread maintains its own wireless channel, updated whenever a transmission event occurs. Both the transmitted data and the corresponding simulated time are recorded. The latter information is required since when another node reads the channel, it compares its clock time and the data timestamp to decide what will be read. The channel is modeled as a FIFO queue.

C. Probing and Backtracking

The probing and backtracking algorithm is implemented as described in Section III. The *wait()* and *notify()* functions of Java *Thread* are used to support the wait and notify functions in the algorithm. We identify a transmission event as the cycle where the last bit of a byte of data is written to the TX register of the radio chip; as a result, a *transmit* event will be fired on the event clock. When a node passes the current global nearest transmission time, a new snapshot needs to be saved by calling the *saveSnapshot()* method on the *Retraceable* interface of the interpreter; it recursively saves object state bottom-up in the object graph. When a restore is needed, the *restoreSnapshot()* method will be called; the state of the simulated node is restored to the snapshot taken at the last global nearest transmission time. In addition, to restore to a correct state, it is necessary to record the position where the *saveSnapshot()* function was called within the interpreter, since several functions in the interpreter may update the clock beyond the time where a snapshot needs to be taken. For example, if *saveSnapshot()* is performed when the interpreter executes a sleep loop function, when *restoreSnapshot()* is called, the program needs to be restored to the same function and continue to simulate the node.

D. Finding Exact Snapshot Points

The backtracking algorithm requires that each snapshot function be performed at the exact point when a transmission occurs. However, capturing the state of a simulated node at the exact cycle of a global nearest transmission event is non-trivial. Although the simulator is cycle-accurate, a given cycle may not "appear" during simulated execution since many instructions consume more than 1 cycle. For example, assume a snapshot is required at cycle 1000. However, an instruction executes at cycle 998 and consumes 4 cycles. When should the snapshot be taken (which must be correct and retraceable)?

To solve this problem, we exploit the event clock mechanism of Avrora. The event clock is used to update the cycle clock after each instruction is executed. Events – for instance, read channel events on the radio – are triggered based on the event clock. When the event clock reaches the scheduled time of an event, the event is fired. As a result, from the view of the event clock, events fire at their scheduled times after the executing instruction is complete. The fired event does not have an impact on the result of the last executed instruction since instructions are atomic, and it preserves the time sequence of events. We add a snapshot event to the event clock once the global nearest transmission time is obtained, and it fires when the node re-executes to the global nearest transmission time. It saves the snapshot on the cycle of the event, after the instruction that crosses the global nearest transmission time. When saving the snapshot, the snapshot event in the event clock is skipped. When the snapshot is restored, the simulator begins from the point where the event clock is updated.

E. Networking Delay

The (physical) networking delay refers to the total time required to encode a broadcast packet at a sender process, transmit the encoded packet, and decode the packet at a receiver process. To make it adjustable for evaluation, the physical networking delay is simulated in SnapSim. The networking delay is usually hundreds of microseconds in a LAN, depending on the network bandwidth and traffic, and usually more than 1 millisecond across the Internet. We use the Java Native Interface (JNI) to issue a Linux system call to *usleep*. As a result, a simulated networking delay with accuracy on the order of tens of microseconds can be configured. The networking delay is assumed to be symmetric between nodes.

V. Evaluation

In this section, we present experimental results for SnapSim, using Avrora as our baseline for comparison[1]. The experiments were performed on a desktop PC with an Intel Core i7 820 processor and 8GB RAM. The JVM heap size was set to 512MB. To limit I/O impacts, the data monitors which display simulation results were disabled for both simulators.

We assume the simulated sensor network is a complete graph; the results represent a cluster within a larger network. For our test cases, we use the TinyOS [9] applications in the Avrora test suite. Fig. 7 summarizes the simulation speed of SnapSim compared to Avrora in a network of 8 nodes, across the test suite. The applications are ordered by bitrate (i.e., transmitted bytes of data per second), from left to right. For each application, The left two bars represent the simulation speeds of SnapSim and Avrora when no networking delay is introduced. The right two bars represent the simulation speeds of SnapSim and Avrora with a networking delay of 500 microseconds, which is a typical LAN delay. The Y-axis denotes the simulation speed in MHz; the axis is logarithmically scaled. For both scenarios, it is clear that SnapSim runs significantly faster than Avrora across all applications. For applications with a very low bitrate (or no transmissions), SnapSim is up to 4000 times faster with a networking delay, and 1150 times faster without a networking delay. For most mid-range applications (i.e., HFreqSamp to OscRF), SnapSim is at least 5.5 times faster with a networking delay, and 4 times faster without a delay. For the application with the highest bitrate, XnpCount, SnapSim is more than 2.3 times faster with a networking delay, and 2.6 times faster without a delay.

[1]Although PolarLite may be a better baseline for comparison, its source code is not publicly available.

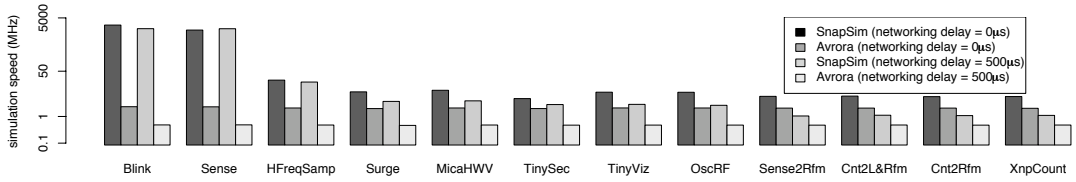

Fig. 7. Performance of Representative TinyOS Applications

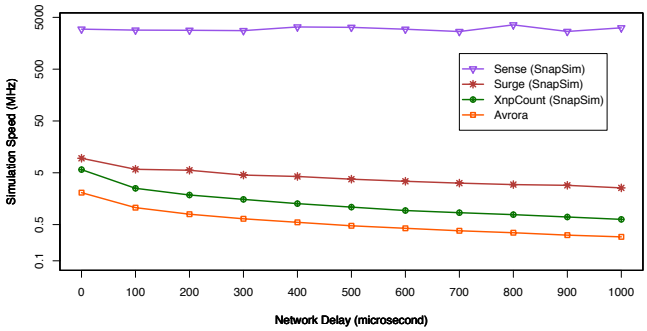

Fig. 8. Performance vs Delay (network size=8)

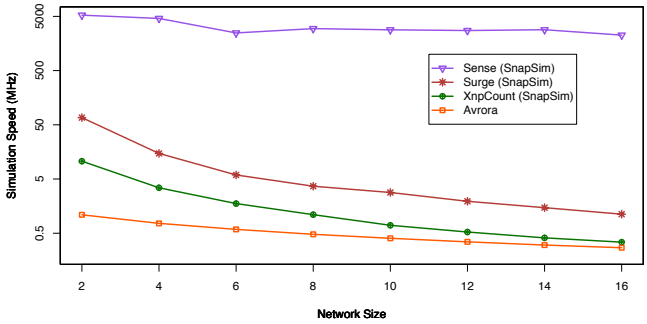

Fig. 9. Performance vs Size (networking delay=500μs)

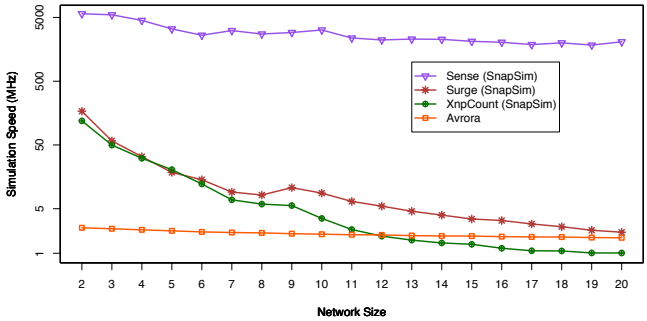

Fig. 10. Performance vs Size (no networking delay)

As seen in the figure, the network bitrate is the key factor that affects the performance of SnapSim since the number of synchronizations depends on the number of bytes transmitted over the network. Avrora runs at a relatively stable rate due to its use of a fixed interval synchronization strategy.

To further evaluate the performance of SnapSim in different scenarios, we choose 3 typical TinyOS applications with varying bitrates: *(i)* ***Sense***, a basic sensing application, which does not use the radio; *(ii)* ***Surge***, a typical tree-based sensor network application, which transmits at a relatively low bitrate; and *(iii)* ***XnpCount***, another tree-based sensor network application, which transmits at a high bitrate. In a network with 8 nodes, the bitrates of the above three applications are approximately 0 bits/s, 1820 bits/s and 9500 bits/s. Fig. 8 shows the performance of SnapSim and Avrora as the networking delay is increased, again in a network of 8 nodes. Since Avrora maintains a relatively stable speed across varying bitrates, only the curve of Surge is shown for Avrora. (The other applications run at similar speeds; the differences could not be seen in the graph.) Again, the y-axis represents simulation speed on a logarithmic scale. The networking delay is tuned from 0 to 1000 microseconds. Notice that omitting the networking delay does not mean there will be no synchronization delay; synchronization among threads also introduces a delay, although usually smaller than the networking delay. The graph reveals three key observations: *(i)* SnapSim runs faster than Avrora, with or without a networking delay. *(ii)* SnapSim simulates the Sense application at the fastest simulation speed, stably, since no synchronizations are performed. *(iii)* The Surge and XnpCount applications slow down as the networking delay increases, for both simulators. The rate of decrease is close to linear. Since the networking delay is introduced via synchronization, the number of synchronizations decides the simulation speed. Our solution reduces the number of synchronizations to an optimal level, yielding higher performance.

We next investigate the scalability of our algorithm. Fig. 9 summarizes the impact of network size on simulation speed with a networking delay of 500 microsceconds. Again, SnapSim is faster than Avrora for all three programs. Sense runs at the highest speed, stably, due to the absence of synchronization overhead. Surge and XnpCount achieve fast simulation speeds when the network size is small. As the network size increases, the data transmitted in the network increases; it directly increases the number of synchronizations, which slows down the simulation. For example, the simulation speed of XnpCount in a 16 node network is only 26% faster than Avrora. In total, 1983 bytes of data are transmitted per second network-wide, which is close to 2400 interval synchronizations per second in Avrora. However, very few wireless sensor network applications reach such a high bitrate (at steady state). Most applications work in a low duty-cycle mode to save energy. Moreover, as the network size increases, the network is usually multi-hop, not completely connected as we have assumed. The network size in our experiments can be treated as a cluster size.

Fig. 10 summarizes the impact of network size on simulation speed in the absence of a networking delay. SnapSim is still faster than Avrora in most cases even when the simulator is centralized on a single desktop. However, as seen in the bottom-right of the graph, SnapSim simulates XnpCount slower than Avrora when the network size is larger than 12. In those cases, the number of synchronizations is very close to the number of intervals in Avrora, and the more complicated synchronization mechanism of SnapSim leads to slower simulation speed.

Application	*Sync. %*	*Save %*	*Restore %*	*Total Running Time (ms)*	*Est. Memory Usage (KB)*
Surge (SnapSim)	19.22	0.71	0.68	87,947	52,408
Surge (Avrora)	82.36	–	–	372,218	35,659
XnpCount (SnapSim)	43.93	1.44	1.36	131,654	52,852
XnpCount (Avrora)	85.55	–	–	363,487	34,673

TABLE I
SYNCHRONIZATION, SNAPSHOTTING, AND BACKTRACKING OVERHEAD

We next evaluate the performance of the synchronization logic used in SnapSim and Avrora. We simulate Surge and XnpCount in a network of 8 nodes, in the absence of networking delay, for 100 simulated seconds each. We accumulate the time spent synchronizing nodes using the *NanoTime* API provided by Java. Table I summarizes the synchronization overhead. The second column of the table shows the percentage of the total running time spent synchronizing nodes. The third and fourth columns show the percentage of time consumed by the save and restore methods, respectively. The fifth column shows the total running time of each application, and the sixth column shows the estimated JVM memory usage, profiled by JProfiler [1]. As the table shows, the synchronization overhead is large for Avrora, even in the absence of networking delay. The synchronization overhead exceeds 80% for Surge, and 85% for XnpCount. In contrast, the synchronization overhead is much lower in SnapSim – approximately 20% – 40%. In our observations, a single synchronization consumes approximately 1 millisecond for Avrora and 850 microseconds for SnapSim since synchronized blocks in Java are inefficient compared to parallel execution. Another observation is that the save and restore snapshot methods consume little running time – no more than 1.5%. As a result, SnapSim simulates much faster than Avrora in most cases.

We next evaluate the JVM heap memory usage of both simulators. SnapSim generally requires a larger heap than Avrora because of its use of snapshots. As detailed in Table I, an increase of 50% heap space was witnessed. We view the increase as worthwhile to achieve better performance.

Finally, we evaluate the number of saved snapshots and restored snapshots. It is clear that the number of saved snapshots is equal to the number of bytes sent by all the nodes in the network. We find that most nodes backtrack to the start of the current wave. For example, for XnpCount in a network of 8 nodes, in the absence of networking delay, only 14.3% of snapshots are not restored, and 12.6% of them involve non-transmitting processes in a wave. Synchronization is slower than simulation, and as a result, most simulated nodes have already executed beyond the possible global nearest transmission time when the broadcast is received. Moreover, the ratio grows smaller as the networking delay increases. For the same scenario with 500 microseconds of networking delay, 9.9% of non-restored snapshots involve non-transmitting processes within a wave. However, compared to the synchronization time, the cost of re-execution is small; it does not have much of an impact on performance.

VI. CONCLUSION

Synchronization is the key to solving transmission dependency problems when simulating distributed networks. The state-of-the-art relies on interval-based synchronization techniques, which incur high overhead. In this paper, we described SnapSim, a novel distributed parallel simulator that uses a probing and backtracking algorithm to reduce the number of synchronizations.

SnapSim optimizes the number of synchronizations to match the number of transmission dependencies. We described the algorithm and provided a case analysis of its correctness. The prototype implementation is based on Avrora. The experimental results show that SnapSim is faster than Avrora in most cases. Considering a networking delay of 500 μs, SnapSim is 8 times faster than Avrora for typical TinyOS applications such as Surge when the network size is 8. Without a networking delay, SnapSim is approximately 4 times faster than Avrora.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation (award CNS-0745846).

REFERENCES

[1] ej-technologies. http://www.ej-technologies.com/.

[2] L. Bajaj, M. Takai, R. Ahuja, and R. Bagrodia. Simulation of large-scale heterogeneous communication systems. In *The IEEE Military Communication Conference*, pages 1396–1400, Washington DC, USA, october–november 1999. IEEE.

[3] D. R. Jefferson. Virtual time. *ACM Transactions on Programming Languages and Systems*, 7:404–425, July 1985.

[4] Z.-Y. Jin and R. Gupta. Improved distributed simulation of sensor networks based on sensor node sleep time. In *In International Conference on Distributed Computing in Sensor Systems*, volume 5067 of *Lecture Notes in Computer Science*, pages 204–218. Springer Berlin / Heidelberg, 2008.

[5] Z.-Y. Jin and R. Gupta. Improving the speed and scalability of distributed simulations of sensor networks. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 169–180, Washington, DC, USA, 2009. IEEE Computer Society.

[6] O. Landsiedel, H. Alizai, and K. Wehrle. When timing matters: Enabling time accurate and scalable simulation of sensor network applications. In *The 7^{th} International Conference on Information Processing in Sensor Networks*, pages 344–355, Washington, DC, USA, 2008. IEEE Computer Society.

[7] U. Legedza and W. E. Weihl. Reducing synchronization overhead in parallel simulation. In *The 10^{th} Workshop on Parallel and Distributed Simulation*, pages 86–95, Washington, DC, USA, 1996. IEEE Computer Society.

[8] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyos applications. In *The 1^{st} International Conference on Embedded Networked Sensor Systems*, pages 126–137, New York, NY, USA, 2003. ACM.

[9] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[10] V. Shnayder, M. Hempstead, B. Chen, G. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *The 2^{nd} International Conference on Embedded Networked Sensor Systems*, pages 188–200, New York NY, USA, november 2004. ACM.

[11] B. Titzer, D. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *The 4^{th} International Symposium on Information Processing in Sensor Networks*, Piscataway NJ, USA, April 2005. IEEE Press.

[12] Y. Wen, R. Wolski, and G. Moore. DiSenS: scalable distributed sensor network simulation. In *The 12^{th} ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 24–34, New York NY, USA, march 2007. ACM.