Sensornet Checkpointing: Enabling Repeatability in Testbeds and Realism in Simulations

Fredrik Österlind, Adam Dunkels, Thiemo Voigt,
Nicolas Tsiftes, Joakim Eriksson, Niclas Finne
{fros,adam,thiemo,nvt,joakime,nfi}@sics.se

Swedish Institute of Computer Science

Abstract. When developing sensor network applications, the shift from simulation to testbed causes application failures, resulting in additional time-consuming iterations between simulation and testbed. We propose transferring sensor network checkpoints between simulation and testbed to reduce the gap between simulation and testbed. Sensornet checkpointing combines the best of both simulation and testbeds: the non-intrusiveness and repeatability of simulation, and the realism of testbeds.

1 Introduction

Simulation has proven invaluable during development and testing of wireless sensor network applications. Simulation provides in-depth execution details, a rapid prototyping environment, nonintrusive debugging, and repeatability. Before deploying an application, however, testing in simulation is not enough. The reason for this, as argued by numerous researchers, is over-simplified simulation models, such as the simulated radio environment. To increase realism, testbeds are employed as an inter-mediate step between simulation and deployment. Testbeds allow applications to be run on real sensor node hardware, and in realistic surroundings. Migrating an application from simulation to testbed is, however, expensive [8, 21, 26]. The application often behaves differently when in testbed than in simulation. This causes additional time-consuming iterations between simulation and testbed.

To decrease the gap between simulation and testbed, hybrid simulation has been proposed [12, 15, 23, 24]. Hybrid simulation contains both simulated and testbed sensor nodes. Although hybrid simulation is feasible for scaling up networks – to see how real nodes behave and interact in large networks – it does not benefit from many of the advantages of traditional simulation. Testbed nodes do not offer nonintrusive execution details. Tests are not repeatable due to the uncontrollable testbed environment: the realism, one of the main advantages of hybrid simulation. Finally, test execution speed is fixed to real-time; it is not possible to increase the test execution speed as when with only simulated nodes.

We propose a drastically different approach for simplifying migration from simulation to testbed. In our approach every node exists both in testbed and

simulation. We checkpoint and transfer network state between the two domains. The entire network is at any given time, however, executing only in either simulation or testbed.

Our approach benefits from advantages of both simulation and testbeds: non-intrusive execution details, repeatability, and realism. By transferring network state to simulation, we can benefit from the inherent properties of simulation to non-intrusively extract network details. We can deterministically repeat execution in simulation, or repeatedly roll back a single network state to testbed. Finally, we benefit from the realism of real hardware when the network is executed in testbed.

To highlight benefits of moving network state between simulation and testbed, consider the following example. A batch-and-send data collection application is executing in a testbed, and is periodically checkpointed by the testbed software. One of the network checkpoints is imported into simulation, and so the simulator obtains a *realistic* network state. While in simulation, network details can be *non-intrusively* extracted and altered. Extracting details this way is comparable to having a hardware debugging interface, such as JTAG, to every node in the testbed: the full state is available without explicit software support on the nodes. In contrast to with hardware debugging interfaces, however, network execution may be continued in simulation.

Simulation provides good debugging and visualization tools, but may not accurately measure environment-sensitive parameters such as multi-hop throughput. Hence, when a collector node is about to send its batched data, the network state is moved back to testbed for accurately measuring the bulk transfer throughput on real hardware. Note that this checkpoint can be imported into testbed multiple times, *repeating* the same scenario, resulting in several throughput measurements.

The rest of this paper is structured as follows. After introducing our checkpointing approach and its application areas in Section 2, we implement it in Section 3. The approach is evaluated in Section 4. Finally, we review related work in Section 5, and conclude the paper in Section 6.

2 Sensornet Checkpointing

Sensornet checkpointing is mechanism for extracting network state from both real and simulated networks. A network checkpoint consists of the set of all sensor node states. Extracted node states can be stored local to each node for offline processing, such as in external flash. When network execution is finished, node states are extracted from external flash to be analyzed. Node state can also be transferred online to outside the network via radio or serial port. A network rollback, the opposite of checkpointing, restores a previously extracted state to the network. Both checkpointing and rolling back network state can be performed at any time, and is synchronous: states from all network nodes are extracted and restored at the same time. During checkpoint and rollback

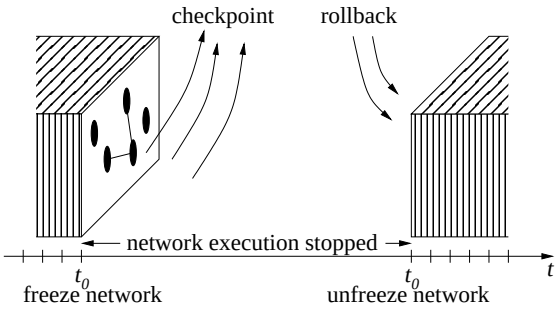

Fig. 1. Checkpointing freezes all nodes in the network at a given point in time. The state of all network nodes can be either stored on the individual nodes for offline processing, or directly downloaded to an external server.

operations, the network is frozen, so the operations are nonintrusive to regular network execution. Figure 1 demonstrates network checkpointing.

Although sensornet checkpointing is a general mechanism, in this paper we focus only on online checkpointing via wired connections. Freezing and unfreezing networks is implemented by serial port commands that stop respectively start node execution. Individual node states are transferred only when the entire network is frozen. Checkpoints could be transferred over radio links for use in deployed networks without wired connections. In this paper we only consider transferring network state between testbeds and simulation.

Node state can be analyzed to extract detailed execution details, such as internal log messages, past radio events, or energy consumption estimates. A node state can also be altered before rolling back the network state. By altering a node state, the node can be reconfigured or injected with faults.

Checkpointing is performed on both simulated and real networks, and a network checkpoint can be rolled back to either simulation or testbed. Hence, by checkpointing in one domain and rolling back to another, we transfer the network state between the domains, for example from testbed to simulation. Since checkpointing is performed non-intrusively (all nodes are frozen), we benefit from advantages associated with both simulation and testbed. For example, we can use powerful visualization and debugging tools available only in simulation on a network state extracted from testbed.

Network checkpointing can be integrated into testbed software. The software periodically checkpoints the network, and stores the state to a database. The checkpointing period is application specific, and may range from seconds to hours. By rolling back the same network state several times, testbed runs can be repeated. This is useful for rapidly evaluating interesting phases during a network lifetime. For example, this approach can be used to measure how many radio packets are required to re-establish all network routes after a node malfunction, or the time to transfer bulk data through a network.

When migrating an application from simulation to testbed, checkpointing can be used to study and compare results of the different application phases. In a data collection network, the first application phase is to setup sink routes on all collector nodes. Later application phases include collectors sending sink-oriented data, and repairing broken links. By executing only the setup phase in both simulation and testbed, and compare the two resulting network states, a developer can make sure that this phase works as expected in testbed. We see several sensor network applications that benefit from sensornet checkpointing: visualization, repeating testbed experiments, protocol tuning, fault injection, simulation model validation, and debugging.

Testbed visualization Checkpoints contain application execution details of all nodes in the network. By rolling back a testbed checkpoint to simulation, information such as node-specific routing tables, memory usage, and radio connections can be visualized using regular simulation tools. The traditional approach for testbed visualization is to instrument sensor network applications to output relevant execution details, for example by printing the current node energy consumption on the serial port. This may have a significant impact on the application, for example by affecting event ordering. The application impact of visualization via sensornet checkpointing is lower since the network is frozen when the execution details are extracted.

Repeated testbed experiments Testbed software can be customized to checkpoint the network when it is about to enter a pre-determined interesting phase. By rolling back the network state, interesting phases can be repeated in testbed, enabling targeted evaluations. Advantages of using checkpointing for evaluations are less test output variance, and faster test iterations. We get less variance in test results since the same network setup is used repeatedly. The test iterations are shorter because the network repeats *only* the evaluated execution phase.

Automated protocol tuning Repeated testbed experiments can be extended by modifying parameters between each test run. In simulation, parameters can be modified using regular simulation tools. Techniques for automatically tuning parameters, such as reinforcement learning, are often used in simulation only due to the many iterations needed. With checkpointing, we enable use of reinforcement learning in testbeds: initial learning is performed in simulation, and the system is further trained in testbed.

Fault injection in testbed Fault injection is a powerful technique for robustness evaluations. To inject errors in a testbed, a testbed checkpoint is rolled back to simulation. Errors are injected in simulation, for example processor clock skews, dropped radio packets, or rebooted sensor nodes. The network state is then again checkpointed and moved back to testbed. Fault injection helps us answer questions starting with "what would happen if…".

Simulation model validation Step-by-step comparisons of testbed and simulation execution help us validate and tune simulation models.

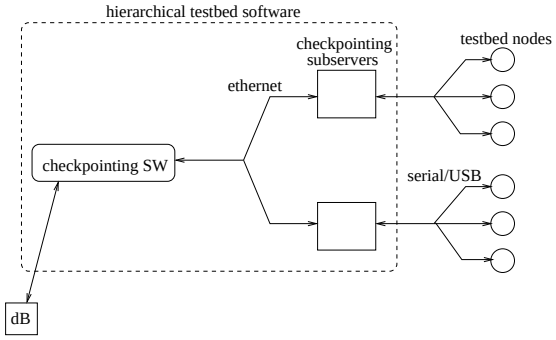

Fig. 2. The checkpointing software is hierarchical and connects the main server to all testbed nodes.

Debugging testbeds Debugging is a challenging task in wireless sensor networks [17, 27]. Debugging nodes is difficult due to the distributed nature of sensor networks in combination with the limited memory and communication bandwidth of each sensor node. Moreover, debugging approaches that depend on radio communication may be too intrusive, or may not even work in a faulty network. In contrast, simulation is well suited for debugging sensor networks. In simulation, a developer has full control of the entire network, can stop and restart the simulation, and can perform node-specific source-level debugging. Sensornet checkpointing can be used for debugging in simulation, whereas regular network execution is on real hardware. When an error is discovered in testbed, the network is checkpointed and rolled back to simulation. Simulation tools may help expose the error cause. In addition, if the testbed is periodically checkpointed, an earlier checkpoint rolled back to simulation may re-trigger the testbed error in simulation.

3 Implementation

We implement checkpointing support on Contiki [3] and the Tmote Sky sensor nodes [16]. The Tmote Sky is equipped with a MSP430F1611 processor with 10Kb RAM, 1Mb external flash, and an IEEE 802.15.4 Chipcon CC2420 packet radio [2]. The sensor nodes communicate using the serial port. The node checkpointing implementation consists of a single process that handles received commands, and forwards state requests to device drivers. The process is executed in a thread separate from the surrounding Contiki. The thread has a 128 byte stack, which is not included in the node state. We use Contiki's cooperative multi-threading library, however, the checkpointing process does not rely on Contiki-specific functionality. Hence, the node checkpointing implementation should be easily ported to other sensor network operating systems.

Devices support checkpointing by implementing two functions: one for extracting device state and one for restoring device state. We implement state

support in four different devices: the LEDs, the radio, the microcontroller hardware timers, and the memory device handling RAM.

See Figure 2 for an overview of the testbed checkpointing software. We implement the subserver on the ASUS WL-500G Premium wireless router [6]. The Tmote Sky sensor nodes are connected to the routers' USB ports, and each router is connected via ethernet to the local network. The routers run OpenWRT[13], a Linux distribution for embedded devices. For accessing sensor node serial ports, we use the Serial to Network Proxy (ser2net) application [19]. ser2net is an open source application for forwarding data between sockets and serial ports.

For checkpointing simulated networks, we use the Contiki network simulator COOJA [14]. COOJA is equipped with the MSP430 emulator MSPSim [5]. MSPSim supports emulating Tmote Sky sensors. Note that checkpointing is performed via serial ports both in real and simulated networks.

3.1 Network State

We define network state as the set of all node states. Node state consists of:

Firmware The firmware programmed on each node, i.e. the compiled program code. Unless the program is altered during run-time, the firmware does not need to be included in the node state.

External flash The 1mb external flash. Unless the external flash is altered during run-time, it does not to be included in the node state.

Memory The 10kb RAM memory. Memory state captures stack and variables.

Timer system Hardware timers, both configuration and current timer values. Note that software timers are included in the memory state.

LEDs The node's light-emitting diodes: on or off.

Radio The CC2420 radio state includes whether the radio is in listen mode, or turned off. We do not include CC2420 chip buffers in the radio state.

Although our definition of node state can be extended with more devices, the definition is in accordance to the needs we observed during this work. The definition should be easy to extend to include more components.

3.2 Communication Protocol

The communication protocol between the checkpointing software and the sensor nodes consists of three asynchronous phases: *freeze node*, *execute commands*, and finally *unfreeze node*. The three phases are asynchronous for better inter-node synchronization: all nodes are frozen before the execute command phase is

started on any of the nodes. Similarly, the *unfreeze node* phase is not initiated on any node until the *execute commands* has finished on all nodes.

The *freeze node* phase is initiated by sending a single byte command to a node. The checkpointing process, executing in the serial port interrupt, immediately performs two tasks: it disables interrupts and stops hardware timers. The node then enters the *execute command* phase. The *execute command* phase consists of the sensor node handling commands from the checkpointing software. Two commands are available: SET_STATE, and GET_STATE. The SET_STATE command prepares the node to receive node state. The GET_STATE command instructs the node to serialize and transmit its current state. When the node has finished handling commands, the *unfreeze* phase is initiated. This phase again starts hardware timers and enables interrupts.

Since the checkpointing process is executed in the serial port interrupt handler, regular node execution is disabled until the node is unfrozen. A limitation of this implementation is that the serial port interrupt has to wait for other currently executing interrupt handlers to finish. Hence, a node stuck in another interrupt handler cannot be checkpointed.

3.3 Checkpointing Thread

Checkpointing runs in its own thread. The checkpointing thread preempts the operating system to perform a checkpoint. The node state includes the operating system stack, but not the checkpointing thread stack. When restoring node memory, we do not overwrite the checkpointing thread stack.

When the checkpointing process exists after rolling back memory state, the operating system is changed including the program counter (PC) and stack pointer (SP). Figure 3 shows an overview of how the operating system memory is restored from the checkpointing thread. Figure 4 contains pseudo code of the serial port interrupt handler, the checkpointing process, and device driver checkpointing support.

3.4 Node State in Simulation

Simulated nodes extract and restore state using the same checkpointing process as real nodes. Since simulated nodes emulate the same sensor node firmware as real nodes, the same checkpointing software can be used to interact with both real and simulated nodes.

3.5 Hierarchical Network Freezing

The main server is able to directly communicate with all individual testbed sensor nodes via ser2net. To improve checkpointing synchronization – avoiding parts of the network still executing after other nodes have frozen – we modify the ser2net application to allow sending freeze and unfreeze commands to all nodes connected to a single router. For this purpose, we use ser2net's control

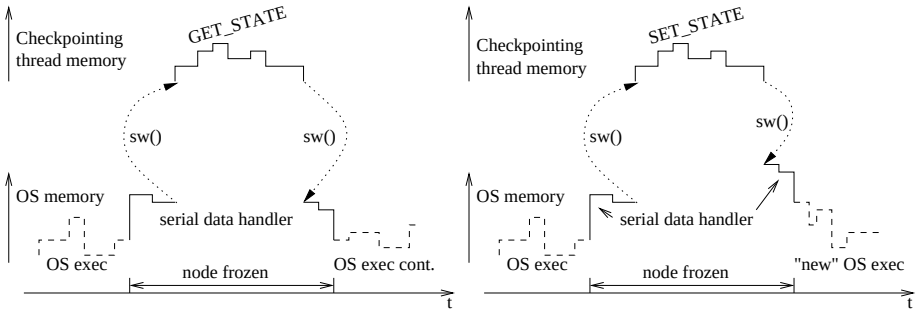

Fig. 3. The checkpointing process is run in a separate thread to enable checkpoints of the operating system stack memory. Left: after checkpointing node memory, control is returned to the OS. Right: after rolling back node memory, the operating system has changed.

port, used for allowing remote control to active ser2net connections. We add support for two new commands: *freeze_all* and *unfreeze_all*. Both commands act on all connected sensor nodes.

3.6 Lost Serial Data

Serial data bytes are sometimes lost when copying state to and from multiple nodes simultaneously. To avoid this problem, we copy state to at maximum one node per subserver. This increases the overall time to transfer network state. It does not, however, affect the inter-node synchronization since all nodes are frozen during the execute command phase. In our current implementation, no CRC check is used to ensure that checkpointed state was transferred correctly. Apart from lost data byte, we have not observed transmission errors in our experiments.

3.7 Hardware Timer Bug

We encountered what appears to be an undocumented MSP430 hardware bug in the timer system during the implementation. In Contiki, the 16-bit hardware Timer_A is configured for continuous compare mode. A compare register is configured and continuously incremented to generate a periodic clock pulse which drives the software time. According to the MSP430 User's Guide, the interrupt flag is set high when the timer value counts to the compare register.

While stress testing our system with incoming serial data, we observed that occasionally the timer interrupt was set high before the timer value incremented to equal the compare register, i.e. the *timer interrupt handler could start executing before the configured time*. A workaround to the problem is modifying the Timer_A interrupt handler to blocking wait until the timer value reaches the compare register if, and only if, the timer value is equal to the compare register

```
/* INTERRUPT HANDLER: Serial data */
handle_serial_data(char c)
  freeze_system(); /* Disable timers, interrupts */
  sw(checkpointing_process); /* Switch to checkpointing thread (BLOCKS) */
  unfreeze_system(); /* Enable timers, interrupts */

/* Checkpointing process blocking reads serial data */
checkpointing_process()
  cmd = READ_COMMAND(serial_stream);
  if (cmd CHECKPOINT)
    foreach DEVICE
      DEVICE_get(serial_stream);

  if (cmd ROLLBACK)
    foreach DEVICE
      DEVICE_set(serial_stream);

  if (cmd not UNFREEZE)
    repeat;

/* Devices drivers handle their own state */
mem_set(serial_stream)
  mem[] = read(serial_stream, mem_size);

radio_get(serial_stream)
  write(serial_stream, radio_state[], radio_size);
```

Fig. 4. Checkpointing pseudo code

minus 1. Although we have not observed it, we expect similar behavior from Timer_B.

4 Evaluation

We evaluate the sensornet checkpointing intrusiveness by checkpointing a data collection network. Furthermore, we evaluate testbed repeatability by rolling back a single network state and comparing multiple test runs. Finally, we report on the synchronization error of our implementation when checkpointing a network.

In this work, there are several artifacts of our experimental setup that influence results. We checkpoint all nodes sequentially, which can be a time-consuming operation in large networks. Furthermore, we transfer the state uncompressed: checkpointing requires copying more than 10kb per node. Using relatively simple techniques, we could reduce the checkpoint state transfer times. Since the memory is the major part of node state, and since most of the memory is often unchanged between checkpoints, diff-based techniques may significantly reduce node state size. A similar approach is using run-time compression algorithms on the nodes. To evaluate the impact of compression, we run-length encode a number of checkpointed node states. Run-length encoding eliminates long runs of zero-initialized memory. The average size of run-length encoded node states was 7559 bytes, a 26% reduction in size. Note that in this evaluation we only run-length encode node state offline – to use compression or diff-based

techniques on real nodes, the node checkpointing process needs additional functionality.

4.1 Intrusiveness: Checkpointing a Data Collection Network

To evaluate the impact of checkpointing a testbed, we implement a simple data collection network. The single-hop network consists of 5 collector nodes and a sink. We perform checkpointing on the collector nodes only. Each collector node samples sensor data, and transmits it via radio to the sink. The sampling intervals, however, differ between the collector nodes. The different send rates for the 5 nodes are: 5, 4, 3, 2, and 1 times per second.

The sink node counts the received radio packets from each collector node. Each radio packet has a sequence number reset at every test run. When the sink receives a radio packet from collector node 1 with a sequence number equal to or above 150, it decides the test run completed, and logs the test run results. Note that the sink may not have received all 150 radio packets from node 1 to end the test; some radio packets may have been lost in transmission.

We checkpoint the network at different rates during each test run. Checkpointing consists of freezing the network, downloading network state, and finally unfreezing the network. Each test run takes approximately 30 seconds network execution, i.e. without the overhead of checkpointing.

We vary the checkpointing interval on the data collection network. With our test setup, checkpointing less than once every 5 seconds had no impact on the transmitted radio packets. When checkpointing more often, we see an increase in lost radio packets. We believe this is due to the high checkpointing rate combined with not storing radio chip buffers: checkpointing when the sensor node radio chip is communicating may cause a radio packet to be dropped. Figure 5 shows the checkpointing impact on the data collection network. Note that Node 1 transmits a packet every second, and Node 5 transmits a packet every 5 seconds.

We believe the observed checkpointing impact on radio communication can be lessened by including radio chip buffers in the radio state. The current implementation clears the CC2420 radio chip buffers, but includes the radio driver state. If the node is checkpointed when the radio driver has copied only the first half of a radio packet, the radio chip buffers will be empty when rolled back, whereas the radio driver will keep copying the second half of the radio packet. The CC2420 radio chip buffers can be accessed via the radio driver. By including these in the radio state, a radio driver can be checkpointed while communicating with the CC2420 without destroying packet data. However, note also that checkpointing as often as every second is not neccessary in any of the applications discussed in Section 2.

4.2 Repeatability: Restoring State of a Pseudo-random Network

For evaluating testbed repeatability, we implement a pseudo-random testbed network with 10 sensor nodes. Each node broadcasts radio packets at pseudo-

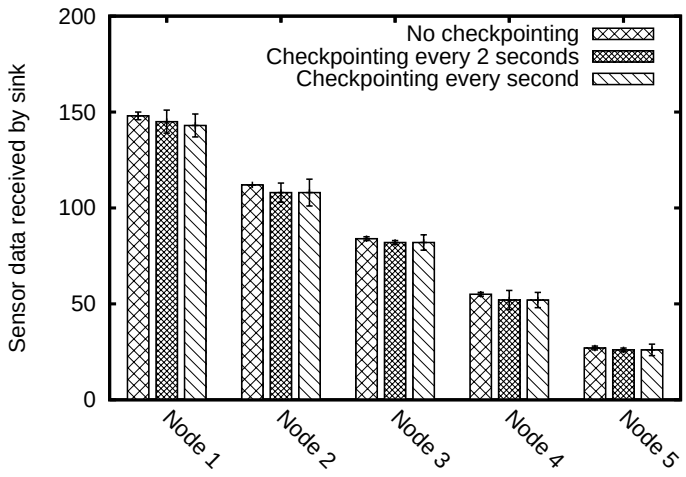

Fig. 5. Checkpointing has little impact on the data collection network, even when checkpointing every second.

random intervals, on the order of seconds. A radio sniffer outside the testbed records any received packets from the 10 nodes.

During the testbed execution we checkpoint the network once. The network state is then repeatedly rolled back to the network 10 times. The radio sniffer records and timestamps any received packets during a test run. The timestamp resolution is milliseconds. We compare the radio sniffer logs of each test run to evalute testbed repeatability. We perform two experiments using the testbed setup: with and without radio Clear-Channel Assessement (CCA).

By analyzing the sniffer logs, we see that the network execution was repeated when the network state was rolled back, i.e. the ordering of radio packets received by the sniffer node was the same for all test runs. With CCA enabled, we observe that not all packets were transmitted in every test run. This is due to the CCA check failing, an event occasionally occurring in a real testbed. Using the unique sequence number of each received packet, we can calculate the average arrival time of each packet. Figure 6 shows the average number of packets received in a test run, and each packet arrival time deviation from mean as measured by the sniffer node. The arrival times are recorded at a laptop connected to the sniffer node. The deviation hence includes scatter from both the sniffer node and the laptop.

4.3 Case Study: Testbed Synchronization

Sensornet checkpointing captures the state of all nodes in a network. For synchronous checkpointing, all nodes must be checkpointed at the same network

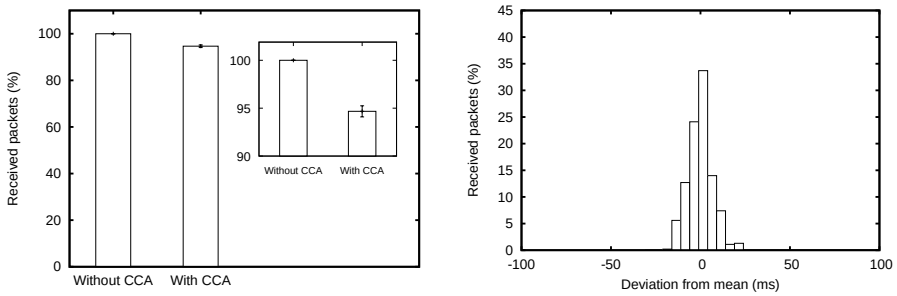

Fig. 6. Left: during the experiment without Clear-Channel Assessment (CCA), each node repeatedly transmitted the same radio packets to the sink. During the experiment with CCA, however, some packets were not retransmitted: the network behavior differed due to the radio surroundings. Right: the packet reception times, as measured by the sink, differs on the order of milliseconds showing that the pseudo-random network is repeated.

time. We freeze the entire network execution during time-consuming checkpointing operations. Hence, checkpointing synchronization error depends on when the freeze (and unfreeze) commands are received and handled by the different nodes.

In our implementation, checkpointing is performed via node serial ports. We reduce the network freeze synchronization error in two ways. First, node freeze commands are one-byte messages. Since freeze commands are sent out sequentially to each node, shorter commands results in better synchronization. Second, freeze commands originate close to the testbed nodes: at the subservers, minimizing message delivery time scatter.

We evaluate how network size affects checkpointing synchronization error. To measure synchronization error we use sensor node hardware timers. Each node dedicates a 32768Hz timer to the evaluation. The checkpointing software, connected to the testbed, continually freezes and unfreezes the testbed. Each time the network is frozen, the current hardware timer value of each node is extracted. Two consecutive node timer values are used to calculate the last test run duration: the network execution time as observed by the node itself.

We define the synchronization error of a test run as the maximum difference between all collected node durations. Hence, with perfect synchronization all nodes would report the same times during a test run, and the synchronization error would be zero. To avoid serialization effects, the order of which nodes are frozen and unfrozen each test run is random, as is the delay between each freeze and unfreeze command. We measure synchronization error on a testbed consisting of up to 30 nodes. The nodes are distributed on 4 subservers ($10 + 9 + 8 + 3$). See Figure 7 for the synchronization errors. In the distributed approach, the checkpointing software sends a single freeze command to each subserver, and the subservers forward the command to all connected nodes. In the centralized approach, the checkpointing software directly sends a command to each node.

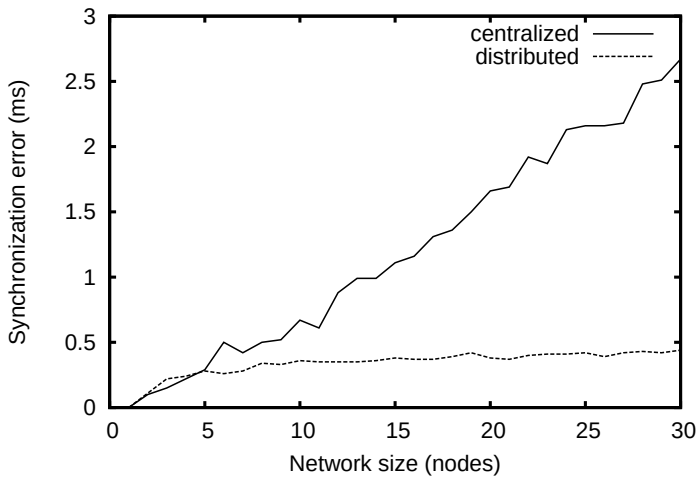

Fig. 7. Freezing and unfreezing nodes directly from subservers (distributed), as opposed to from the main server (centralized), significantly improves the synchronization: With 30 nodes distributed on 4 subservers, the synchronization error is less than 0.5 ms.

The distributed approach is visibly more scalable and provides a significantly lower synchronization error than the centralized approach. With 30 testbed nodes the average synchronization error is 0.44 ms with the distributed approach, and 2.67 ms with the centralized approach.

5 Related Work

Distributed checkpointing and rollback is a standard technique for failure recovery in distributed systems [1, 4] and is used e.g. in file systems [18], databases [20], and for playback of user actions [10]. Inspired by the substantial body of work in checkpointing, we use the technique to improve realism in sensor network simulation and repeatability in sensor network testbeds. To the best of our knowledge, we are the first to study distributed checkpointing with rollback in the context of wireless sensor networks.

Sensor network testbeds is a widely used tool for performing controlled experiments with sensor networks, both with stationary and mobile nodes [7, 25]. Our work is orthogonal in that our technique can be applied to existing testbeds without modification. There is a body of work on sensor network simulators. TOSSIM [11] simulates the TinyOS sensor network operating system. Avrora [22] emulates AVR-based sensor node hardware.

A number of sensor network simulators allow a mixture of simulated nodes and testbed nodes. This technique is often called hybrid simulation [12, 15, 23, 24]. Although hybrid simulation is designed to increase the realism by running

parts of the network in a testbed, repeatability is affected negatively. Our work is originally based on the idea of hybrid simulation, but we identify key problems with hybrid simulation and show that synchronous checkpointing lessens the effects of these problems.

Several techniques for debugging and increasing visibility have been proposed for sensor networks. Sensornet checkpointing can be used as an underlying tool when implementing debugging and visibility, and can be combined with several of the existing tools.

NodeMD [9] uses checkpoints on individual nodes for detecting thread-level software faults. The notion of checkpoints is used differently in NodeMD: checkpoints are used by threads to signal correct execution. NodeMD furthermore stores logs in a circular buffer used to debug the node when an error has been detected. Sensornet checkpointing can be combined with many the features in NodeMD: a node-level error detected by NodeMD can trigger a network-wide checkpoint, and the circular buffers can be used as an efficient way to transfer messages between testbed and simulation.

6 Conclusions

We implement checkpointing for sensor networks. Our approach enables transferring network state between simulation and testbed. Several applications benefit from the approach, such as fault injection and testbed debugging. We show that sensor network checkpointing enables repeatable testbed experiments and non-intrusive testbed execution details.

Acknowledgments

This work was partly financed by VINNOVA, the Swedish Agency for Innovation Systems and the SICS Center for Networked Systems, partly funded by VINNOVA, SSF and KKS. This work has been partially supported by CONET, the Cooperating Objects Network of Excellence, funded by the European Commission under FP7 with contract number FP7-2007-2-224053.

References

1. K. Chandy and L. Lamport. Distributed snapshots: determining global states of distributed systems. *ACM Trans. Comput. Syst.*, 3(1):63–75, 1985.
2. Chipcon AS. CC2420 Datasheet (rev. 1.3), 2005.
3. A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Workshop on Embedded Networked Sensors*, Tampa, Florida, USA, November 2004.
4. E. Elnozahy, L. Alvisi, Y. Wang, and D. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Computing Surveys*, 34(3):375–408, 2002.

5. J. Eriksson, A. Dunkels, N. Finne, F. Österlind, and T. Voigt. Mspsim – an extensible simulator for msp430-equipped sensor boards. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007.
6. ASUSTek Computer Inc. ASUSTek Computer Inc. Web page: http://www.asus.com/. Visited 2008-09-25.
7. D. Johnson, T. Stack, R. Fish, D. Flickinger, L. Stoller, R. Ricci, and J. Lepreau. Mobile emulab: A robotic wireless and sensor network testbed. In *Proceedings of IEEE INFOCOM 2006*, April 2006.
8. D. Kotz, C. Newport, R. S. Gray, J. Liu, Y. Yuan, and C. Elliott. Experimental Evaluation of Wireless Simulation Assumptions. In *Proceedings of the ACM/IEEE International Symposium on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM '04)*, pages 78–82, October 2004.
9. V. Krunic, E. Trumpler, and R. Han. NodeMD: Diagnosing node-level faults in remote wireless sensor systems. In *MOBISYS '07*, San Juan, Puerto Rico, June 2007.
10. O. Laadan, R. Baratto, D. Phung, S. Potter, and J. Nieh. Dejaview: a personal virtual computer recorder. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 279–292, Stevenson, Washington, USA, 2007.
11. P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 126–137, 2003.
12. S. Lo, J. Ding, S. Hung, J. Tang, W. Tsai, and Y. Chung. SEMU: A Framework of Simulation Environment for Wireless Sensor Networks with Co-simulation Model. *LECTURE NOTES IN COMPUTER SCIENCE*, 4459:672, 2007.
13. OpenWRT. OpenWRT Wireless Freedom. Web page: http://openwrt.org/. Visited 2008-09-25.
14. F. Österlind, A. Dunkels, J. Eriksson, N. Finne, and T. Voigt. Cross-level sensor network simulation with cooja. In *Proceedings of the First IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, Tampa, Florida, USA, November 2006.
15. S. Park, A. Savvides, and M.B. Srivastava. SensorSim: a simulation framework for sensor networks. *Proceedings of the 3rd ACM international workshop on Modeling, analysis and simulation of wireless and mobile systems*, pages 104–111, 2000.
16. J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.
17. N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 255–267, 2005.
18. M. Rosenblum and J. Ousterhout. The design and implementation of a log structured file system. In *SOSP'91: Proceedings of the 13th ACM Symposium on Operating System Principles*, 1991.
19. ser2net application. Serial to Network Proxy (ser2net). Web page: http://ser2net.sourceforge.net/. Visited 2008-09-25.
20. Sang Hyuk Son and A.K. Agrawala. Distributed checkpointing for globally consistent states of databases. *IEEE Transactions on Software Engineering*, 15(10):1157–1167, 1989.
21. M. Takai, J. Martin, and R. Bagrodia. Effects of Wireless Physical Layer Modeling in Mobile Ad Hoc Networks. In *Proceedings of MobiHoc'01*, October 2001.

22. B.L. Titzer, D.K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of the 4th international symposium on Information processing in sensor networks (IPSN)*, April 2005.
23. D. Watson and M. Nesterenko. Mule: Hybrid Simulator for Testing and Debugging Wireless Sensor Networks. *Workshop on Sensor and Actor Network Protocols and Applications*, 2004.
24. Y. Wen and R. Wolski. Simulation-based augmented reality for sensor network development. *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 275–288, 2007.
25. G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proc. IPSN/SPOTS'05*, Los Angeles, CA, USA, April 2005.
26. M. Woehrle, C. Plessl, J. Beutel, and L. Thiele. Increasing the reliability of wireless sensor networks with a distributed testing framework. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 93–97, New York, NY, USA, 2007. ACM.
27. J. Yang, M.L. Soffa, L. Selavo, and K. Whitehouse. Clairvoyant: a comprehensive source-level debugger for wireless sensor networks. *Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 189–203, 2007.