On the Potential of Software Rejuvenation for Long-Running Sensor Network Deployments

Matthias Woehrle
Computer Engineering and Networks Lab
ETH Zurich
woehrlem@tik.ee.ethz.ch

Andreas Meier
Computer Engineering and Networks Lab
ETH Zurich
a.meier@tik.ee.ethz.ch

Koen Langendoen
Embedded Software group
Delft University of Technology
The Netherlands
k.g.langendoen@tudelft.nl

ABSTRACT

Many sensor network systems encounter considerable problems after deployment despite extensive simulation and testing during the development. A fundamental issue is unforeseen problems that rarely occur, which makes them hard to reproduce. This work focuses on a class of problems that occur due to so-called software aging: The classical software engineering approach to handle software aging effects is software rejuvenation, i. e., the proactive reset of software components. In this positioning paper we discuss whether software rejuvenation can be applied to resource scarce sensor nodes, which are tightly coupled distributed system. We detail why and how software rejuvenation techniques are applicable to sensor networks and presents the basic building blocks required.

Categories and Subject Descriptors

D.2.9 [**Software Engineering**]: Management—*Life cycle*

General Terms

Reliability

Keywords

Wireless Sensor Networks, Software Health, Software Rejuvenation

1. INTRODUCTION

Wireless sensor networks are autonomous systems deeply embedded into a harsh and hostile environment [3, 4, 21]. Significant challenges arise due to the unforeseen interaction with the environment, which is dynamic and changes considerably over time. Many deployments of sensor networks have suffered from these effects. On the one hand, there are problems that can only be solved by replacing the hardware or the software (reprogramming). On the other hand, there is the class of software problems (called Mandelbugs in [12] and Aging-related bugs in [22]) that are caused by the long running, interactive and autonomous nature of sensor networks: *software aging* [13]. The embedding of software into the real, imperfect world (hardware) can cause aging effects:[1] Overflows, memory waste, and files that grow in external file systems [14]. Note that many of these problems trigger errors that are only exposed after an extended period of time. This renders them hard to detect during development, since the activation patterns necessary to trigger these errors are complex.

Software aging can be addressed using *software rejuvenation* [13] techniques. That is, the system is rebooted or re-initialized in order to bring it back to a known good state, where program execution can safely continue. Embedded system include the rudimentary technique of watchdog timers, which can be seen as a kind of software rejuvenation. Watchdogs are a suitable, last-line detectors for software problems, and take the drastic solution of rebooting the node. Previous work [8] has even proposed periodic watchdog-like reboots (*Grenade Timers*) to limit the software age to a specific maximum. Note that reboots have a quite an impact on software as all state held in memory is lost. There are different software layers that all require establishing and maintaining vital state; time synchronization, localization algorithms, sensor calibration, coordinated MAC protocols, and routing protocols are *stateful* and all necessitate an initialization phase.

Initialization due to reboots is harmful, as normal system functionality is temporarily suspended, and expensive as well, as nodes spent additional energy to rebuild their state [6]. As an example, the CTP routing protocol only reaches a stable network topology after about an hour burning precious energy [16]. Thus, rather than rebooting a node completely, we suggest the rejuvenation of those components (or layers) of the software stack that suffer from software aging. In this paper we address the question whether sensor network deployments can benefit from software rejuvenation of individual components in a distributed and tightly coupled system of sensor nodes. In particular this position paper contributes three-fold:

- We introduce software rejuvenation as an approach for sensor networks to handle the effects of software aging.

[1]There is another definition of software aging [18], which concerns maintenance and software updates and is orthogonal to the software aging discussed in this work.

SESENA'10, May 3, 2010, Cape Town, South Africa.

- We report on initial simulation experiments showing that software rejuvenation is feasible and outperforms complete reboots in many cases.
- We present the underlying building blocks and software architecture (see Figure 1) necessary to integrate software rejuvenation on a sensor node.

Our overall conclusion is that software rejuvenation is a promising approach for handling rare, age-related errors that inevitably arise in long-running sensor network deployments.

2. BACKGROUND AND RELATED WORK

Software aging and rejuvenation [11, 13] are not novel concepts. They are established software engineering approaches, e. g., suitable for systems with availability requirements such as web servers [10]. There is little work concerning software rejuvenation for sensor networks. Parvin et al. [19] discuss software rejuvenation in terms of survivability. Their approach focuses on complete clusters of sensor nodes. A cluster is rejuvenated (rebooted) triggered by the sink, which determines aging effects in the cluster. In contrast our work aims to be as little invasive as possible, i. e., not even rebooting a single node and resetting only the essential components of a node. A similar idea concerning memory safety is presented in [6], where the authors show the possibility of *recovery units*, which are isolated and independent software units and to perform localized reinitializations required by memory safety violations rather than necessitating a complete reboot of the node. The work shows that resetting only parts of an application has a significant impact on the system behavior for routing, time sync or code image distribution. While the work also concerns resetting of individual components, our software rejuvenation is orthogonal to [6] as it concerns a whole different class of software problems.

An important aspect for both classical, reboot-all and selective rejuvenation approaches is that somehow the underlying failures causing system degradation, or even complete failure, must be uncovered. Several methods were devised for *health monitoring* of sensor networks that go beyond identifying failures at the node level, and operate at the component (layer) level. A simple, yet powerful way to detect component errors is the use of contracts [17]. Contracts basically specify safety properties, i. e., verifying that nothing bad has happened. [1] shows the use contracts based on NesC interfaces in TinyOS. Deficiencies may also be detected on a higher layer in the software stack and necessitate an elaborate root-cause analysis, e. g., based on a decision tree [20]. Another method is to use models of valid behavior and attribute detected failures with regard to the model to individual components [7] in order to decide on the most-likely cause (among different possibilities) of an error.

Once a faulty component has been identified, it must be brought back to a valid state. Rebooting is attractive as it is simple and ensures that all components, including the faulty one, are set back to the clean starting state. However, all accumulated information stored in (volatile) memory is also lost, implying a costly recovery process. Our work is inspired by the idea of micro-reboots in large distributed systems [5]. By focussing rejuvenation on individual components, we try to avoid the large cost (in energy and availability) of total reboots. While the fundamental mechanism is the same, the applicability may be different as sensor network systems differ in many system aspects such as structure (highly cooperative), design (tightly integrated and considerable shared state) and resources (little memory). Nevertheless, some of the issues found in larger systems persists in sensor networks. We will detail on this in the concluding discussion.

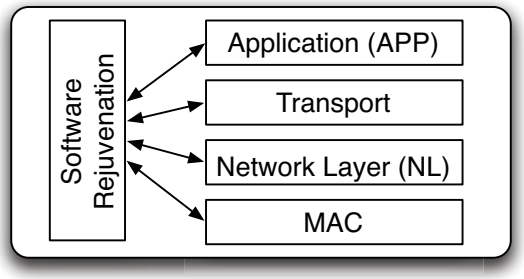

Figure 1: Overview of our software rejuvenation approach: Fine-granular monitoring and reset allows for detection of software aging and applying an according reset.

The fundamental question is whether for sensor networks a fine-granular resetting of a few affected components is possible, and hence would allow the remaining components to keep their state. If we reset a single protocol layer or component, does it create negative effects in other layers or components of the software? Furthermore, how does a partial reset compare to a complete reboot, concerning the remaining nodes in the network? Our particular interest are in the energy demands and the *reset time*. The reset time includes the duration of the actual reset as well as a subsequent time to get back to a *stable state* in all components.

In the following section, we try to answer these question with a feasibility study. We study two parts of the protocol stack, the MAC and the network layer. We study the behavior of individual component resets of a single protocol layer and the effect on overall behavior.

3. FEASIBILITY STUDY

For evaluating the benefit of software rejuvenation to sensor network deployments, we ran experiments with the standard data-gathering application provided in TinyOS 2 (`MultihopOscilloscope`) using the collection tree protocol (CTP) for routing packets to the sink. CTP maintains the network topology by means of regular beacon messages send with an adaptive interval ensuring agile response and low overhead; starting from a low interval the time between beacons is steadily increased, up to a certain maximum, to save energy [9]. This process is repeated when changes in the neighborhood, e. g., failing links, are observed. We define the network being in a *stable state* when all nodes send their beacons with the maximum beacon interval.

We perform two experiments. First, we take a look at the MAC layer, where we study the impact of the reset times of a MAC protocol on the overall system behavior. Second, we evaluate different reset strategies on the network layer. With these experiments, we try to answer the following questions: Do the independent layers have an impact on each other? Does the granular reset of these layers have any benefit over a complete reboot?

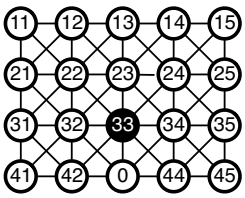

Figure 2: 4x5 Grid used for TOSSIM simulations with a example routing tree.

t_{block}	t_{send}	k_{reset}	n_{lost}	n_{uni}	n_{broad}
1 ms	6.3	0.4	0.1	1743.2	22.2
10 ms	6.3	0.1	0.6	1732.5	22.4
100 ms	6.3	0.1	0.3	1722.0	21.9
1 s	6.3	0.2	0.3	1705.0	20.7
10 s	12.3	7.9	2.6	2664.5	140.4
100 s	102.3	9.5	6.0	2467.1	165.9

Table 1: Performance of nodes when blocking the MAC protocol for t_{block}. All times in seconds.

3.1 Experimental setup

We run simulations on TOSSIM modeling a MicaZ node (CC2420 radio) with channel qualities generated by USC's Realistic Wireless Link Quality Model and Generator[2]. We do not include noise in the simulations in order to make results better comparable and ease analysis. We test with a sensor network containing 20 nodes that are arranged in a grid as illustrated in Figure 2. The sink (node 0) is located in the middle of the lower border. Each run comprises 40 minutes of data gathering (26 byte messages, no aggregation). Initial experiments with this setup show that the application converges to a stable state within 10 minutes after starting all nodes. In this state the application injects one data packet into the network every 20 s, and CTP emits a routing beacon every 512 s. After $t_r = 20$ minutes (i. e., having a stable state), we reset a component (or perform a reboot) of node 33 (cf. Figure 2). During the time interval $[t_r - 2, t_r + 10]$ minutes we evaluate the effects of the resets using the following performance metrics:

- t_{send}: Time until the first packet is sent after t_r by the reset node 33.
- k_{reset}: Number of regular nodes experiencing a reset of the beacon period.
- n_{lost}: Number of lost data messages from all nodes.
- n_{uni}: Number of unicast packets sent by all nodes.
- n_{broad}: Number of broadcast packets sent by all nodes.

The message and packet counts depend on the data rate of the monitoring application. We experimented with additional injection intervals (i. e., 2 s and 200 s) and report on these results when the network behavior deviates significantly from the default 20 s case. All results are averaged over 10 runs.

3.2 MAC layer

The time to reset and the subsequent initialization of the MAC protocol largely depend on the type of MAC protocol [15]. In particular whether or not the MAC shares state with its neighbors (e. g., slot allocation) has a noticeable effect. For stateless MAC protocols such as B-MAC, RI-MAC and X-MAC, a reset of the MAC layer usually takes only a few milliseconds, which is basically the time needed to the reboot the radio module. For stateful protocols such as the frame-based LMAC and Crankshaft protocols, the initialization time is usually much longer. For LMAC the node has to listen to one complete frame ($\leq 100ms$), whereas Crankshaft requires to receive a sync beacon ($1 - 10s$).

In order to simulate the effects of the wide range of MAC setup times, we take a general approach and block the communication in the simple CSMA protocol provided by TOSSIM for a time t_{block} after the reset as presented in Table 1. That is, the resetting node can neither send nor receive any message during this time. As an example, $t_{block} = 10$ ms corresponds to the reset of the stateless X-MAC protocol, while $t_{block} \geq 10$ s corresponds to the stateful Crankshaft protocol. As the results show, for a block time up to a second, there is only a local effect and the neighboring nodes are not affected ($k_{reset} \approx 0$). However when communication is blocked for a longer period, the MAC reset has a global effect. Neighbors detect a missing parent, trigger a route discovery phase, and re-route the traffic around the reset node 33 keeping the number of lost messages to a minimum. The effect is that a large part of the network resets their beacon period and shows an increased unicast message count, since various routes get longer. It should be noted that these particular overheads proportionally depend on the data rate: when we evaluate with a high packet generation rate of 2 s, global effects already show at a MAC block of $t_{block} = 1$ s, whereas for a rate as low as 200 s, the first global effects are seen at $t_{block} = 100$ s.

We conclude that selectively rebooting the MAC layer is a viable option; small block times have only a local effect and even long block times associated with stateful protocols have a minor impact on low-level network parameters. As a second step we take a look at the network layer to underline these results.

3.3 Network layer

The CTP routing protocol provides a modular network layer consisting of three components: the Routing Engine (RE), the Forwarding Engine (FE) and the Link Estimator (LE). LE contains the neighbor table and maintains link qualities. RE contains the control logic for parent selection based on the neighbor table. FE is responsible for the data flowing through a node and takes care of the actual forwarding, including retransmissions and the like.

Table 2 presents the results for resetting different combinations of CTP components, including resetting the complete network layer. For reference normal runs without any reset, and runs involving a complete reboot are also included. In a nutshell, we observe that the reset of the FE does not influence neighboring nodes since no additional beacons are sent. In contrast, all other reset scenarios result in a reset of the beacon interval of nodes in the neighborhood ($k_{reset} \approx 11$). Furthermore we observe that these resets, although being faster (cf. the T_{send} times), are about as expensive as a complete reboot of node 33 in terms of overhead in packet counts. Note that this increase in transmitted messages (particularly broadcasts) translates to a significant increase in energy consumption as the radio is the major consumer of energy on a sensor node. This indicates that fine-granular resets at the network layer are of little use; the

[2]`http://anrg.usc.edu/www/index.php/Downloads`

Type	t_{send}	k_{reset}	n_{lost}	n_{uni}	n_{broad}
normal	-	0	0.5	1734.6	24.3
FE	6.3	0.2	0.2	1703.6	24.2
LE	6.3	11.1	0.6	2127.0	186.2
RE	11.4	8.5	0.7	1658.7	106.9
LE+RE	10.7	12.1	0.3	1718.3	166.7
LE+RE+FE	12.3	11.2	0.8	1728.4	161.1
Reboot	25.0	10.8	1.7	1703.7	150.4

Table 2: Performance of nodes when resetting different components of the network layer. All times in seconds.

reason being that the routing layer maintains the most expensive state (i. e., the one that takes the longest to become stable) in our test application.

Note that selective resets can even prove to be counter productive as shown by the results for resetting the LE without RE, which causes adverse behavior leading to an increased number of unicasts and broadcasts. Since the LE reset clears the neighbor table without notifying the routing engine, the zeroed link state is propagated (by means of the routing beacons) to the neighbors. This in turn leads them to believe that a shorter route to the sink has become available, tricking most of them in routing their traffic over a path that is actually longer than the current one. Due to the dependency of RE on LE, it is necessary to reset RE when LE needs a reset. This shows that dependencies of components, e. g., due to shared state, need to be identified to avoid adverse behavior.

4. DISCUSSION

The results from the feasibility study in the previous section show that software rejuvenation is a promising approach for handling software aging bugs in sensor networks. In particular, we showed that fine-granular resets do not incur the full overhead, in terms of time and energy, for recovering precious state after a complete reboot as triggered by a conventional watchdog timer.

The major difficulty for general usage of rejuvenation is that the reset of stateful components is expensive. While our work focused on the effects on the network layer as a stateful component, previous work has shown similar effects for time-sync or reprogramming [6]. Many more sensor network software components require persistent state for a correct and efficient functioning (e. g., 6lowpan and TCP, FTSP, application states, sensor calibration routines). For all these stateful components, it can take substantial time before the component is fully active again. Meanwhile other components and even other nodes can be affected. In particular, these adverse affects are not necessarily limited to the node itself but can include components of neighboring nodes. Hence, stateful components should only be rejuvenated on a clear indication of their malfunctioning.

To determine the components that would benefit from a rejuvenation, it must be known which components can be reset inexpensively and which components reset needs to be considered with care. For the former ones, it is a viable option to perform resets proactively on a regular basis. For the latter ones, suitable methods need to be defined to determine aging effects and attribute them to the according component. To this end we propose to use the approach depicted in Figure 1; the typical software stack for sensor networks with individual protocol layers and one (or more) applications on top is augmented with an orthogonal software rejuvenation block. This control logic is responsible for (i) monitoring software state, (ii) attributing adverse conditions to individual components and (iii) resetting individual components. In order to determine the software components that require to be reset, we need to monitor functional and performance metrics. However, when considering resource-limited sensor nodes, traditional techniques [2, 10, 13, 22] for the monitoring, e. g., as used for server systems, cannot be employed. Hence, we need lightweight methods for monitoring and for analysis of current node and component state. In previous work, we have shown with the MoMi framework how model-based diagnosis can be realized using lightweight models and conflict detection [7]. For software rejuvenation, we may assume a similar component for detecting problems and attributing them to individual components.

When faced with multiple components as sources for problems we can take two different approaches.

- Reset all components in one go.
- Determine an order of reset. Resets are performed sequentially until the conflict is resolved. This approach tries to be less intrusive to the node software, but incurs additional latency since potentially multiple resets need to be performed.

In light of the results of Section 3, we see that a reset order may be beneficial, especially concerning components with global effects. As an example when faced with problems stemming from the network layer and the MAC, one would prefer to merely reboot the MAC layer first, since it is less intrusive. For a second step, one may decide with resetting both layers (or even a reboot), since the network layer reset costs dominate the MAC layer reset costs.

A different, yet important aspect for research is the segregation of persistent, vital state from control logic. Candea et al. [5] provide suitable design approaches that may be applied to (i) cope with stateful resources and (ii) allow for integration of fine-granular rejuvenation into applications. One of the fundamental approaches is to try to decouple components as much as possible. However, in sensor networks state is often shared between components and necessitating combined rejuvenation, as in the example of the dependency of RE and LE in Section 3.3. Data flow analysis as performed in Nucleus [6] may support better or automated separation. In case a complete separation of vital state is possible, periodic rejuvenation of the remaining state (i. e., components) becomes feasible. This would render an analysis of aging causes and attribution only necessary for those few components maintaining vital state.

5. CONCLUSION

This position paper introduced the idea of component-based software rejuvenation for long-running wireless sensor network deployments. We showed its feasibility and applicability by a controlled simulation experiment, and discussed the fundamental issues and complexities involved in operating software rejuvenation on a typical protocol stack. Future work needs to address these problems to arrive at a reliable, yet resource-efficient solution for long-running sensor network deployments.

6. ACKNOWLEDGMENTS

This work was supported by the National Competence Center in Research on Mobile Information and Communication Systems (NCCR-MICS), a center supported by the Swiss National Science Foundation under grant number 5005-67322. Koen Langendoen kindly acknowledges the support from CONET, the Cooperating Objects Network of Excellence, funded by the European Commission (contract number FP7-2007-2-224053).

7. REFERENCES

[1] W. Archer, P. Levis, and J. Regehr. Interface contracts for tinyos. In *Proc. 6th Int'l Conf. Information Processing Sensor Networks (IPSN '07)*, pages 158–165, New York, NY, USA, 2007. ACM Press.

[2] Y. Bao, X. Sun, and K. S. Trivedi. Adaptive software rejuvenation: degradation model and rejuvenation scheme. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, pages 241–248, June 2003.

[3] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli. The hitchhiker's guide to successful wireless sensor network deployments. In *Proc. 6th ACM Conf. Embedded Networked Sensor Systems (SenSys 2008)*, 2008.

[4] J. Beutel, S. Gruber, A. Hasler, R. Lim, A. Meier, C. Plessl, I. Talzi, L. Thiele, C. Tschudin, M. Woehrle, and M. Yuecel. PermaDAQ: A scientific instrument for precision sensing and data recovery in environmental extremes. In *ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, page (to appear), 2009.

[5] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot - a technique for cheap recovery. In *OSDI*, pages 31–44, 2004.

[6] Y. Chen, O. Gnawali, M. Kazandjieva, P. Levis, and J. Regehr. Surviving sensor network software faults. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 235–246, New York, NY, USA, 2009. ACM.

[7] A. de Jong, M. Woehrle, and K. Langendoen. MoMi - model-based diagnosis middleware for sensor networks. In *Proc. 4th Int'l Conf. Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens'09)*, pages 19–24, Urbana Champaign, IL, USA, December 2009. Springer.

[8] P. Dutta, J. Hui, J. Jeong, S. Kim, C. Sharp, J. Taneja, G. Tolle, K. Whitehouse, and D. Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proc. 5th Int'l Conf. Information Processing Sensor Networks (IPSN '06)*, pages 407–415, 2006.

[9] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. Technical Report SING-09-01, Stanford Information Networks Group, 2009.

[10] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi. Analysis of software aging in a web server. *IEEE Transactions on Reliability*, 55(3):411–420, Sept. 2006.

[11] M. Grottke, R. Matias, and K. S. Trivedi. The fundamentals of software aging. In *Software Reliability Engineering Workshops, 2008. ISSRE Wksp 2008. IEEE International Conference on*, pages 1–6, Seattle, WA, USA,, Nov. 2008.

[12] M. Grottke and K. S. Trivedi. Fighting bugs: Remove, retry, replicate, and rejuvenate. *Computer*, 40(2):107–109, Feb. 2007.

[13] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton. Software rejuvenation: analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on*, pages 381–390, Pasadena, CA, USA, June 1995.

[14] M. Keller, J. Beutel, A. Meier, R. Lim, and L. Thiele. Learning from sensor network data. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 383–384, New York, NY, USA, 2009. ACM.

[15] A. Meier. *Safety-Critical Wireless Sensor Networks*. PhD thesis, ETH Zurich, June 2009.

[16] A. Meier, M. Woehrle, M. Weise, J. Beutel, and L. Thiele. NoSE: Efficient maintenance and initialization of wireless sensor networks. In *Proc. Sixth Annual IEEE Communications Society Conference on Sensor, Mesh, and Ad Hoc Communications and Networks (SECON 2009)*, pages 1–9, Rome, Italy, June 2009. IEEE.

[17] B. Meyer. Design by contract: Making object-oriented programs that work. In *Technology of Object-Oriented Languages and Systems, 1997. TOOLS 25, Proceedings*, pages 360–361, Nov. 1997.

[18] D. L. Parnas. Software aging. In *Software Engineering, 1994. Proceedings. ICSE-16., 16th International Conference on*, pages 279–287, Sorrento, Italy, May 1994.

[19] S. Parvin, D. S. Kim, S. M. Lee, and J. S. Park. Achieving availability and survivability in wireless sensor networks by software rejuvenation. In *SecPerU '08: Proceedings of the 4th international workshop on Security, privacy and trust in pervasive and ubiquitous computing*, pages 13–18, New York, NY, USA, 2008. ACM.

[20] N. Ramanathan, K. Chang, R. Kapur, L. Girod, E. Kohler, and D. Estrin. Sympathy for the sensor network debugger. In *Proc. 3rd ACM Conf. Embedded Networked Sensor Systems (SenSys 2005)*, pages 255–267. ACM Press, New York, 2005.

[21] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proc. 2nd ACM Conf. Embedded Networked Sensor Systems (SenSys 2004)*, pages 214–226. ACM Press, New York, Nov. 2004.

[22] K. Vaidyanathan and K. S. Trivedi. A comprehensive model for software rejuvenation. *IEEE Trans. Dependable Secur. Comput.*, 2(2):124–137, 2005.