Distribution of Updates to IoT Nodes in a Resource-Challenged Environment

Roberth Tollefsen, Issam Rais, John Markus Bjørndalen, Phuong Hoai Ha, Otto Anshus
Department of Computer Science, UiT The Arctic University of Norway, Tromso, Norway
Corresponding author: roberth.tollefsen@uit.no

Abstract—IoT nodes need to be updated after deployment. However, doing so for nodes deployed to resource-challenged environments, like the arctic tundra, is a challenge. Because humans as the common case cannot physically visit the nodes, updating must be done from a remote update service over a back-haul data network. However, most nodes are not in range of a back-haul data network. Even when nodes are in range, they probably sleep to conserve energy and the remote cloud back-end therefore cannot communicate with them.

We report on an approach and a prototype system for distributing updates from a cloud update distribution service to nodes. We assume that nodes carry one or several local area network technologies supporting short-lived point-to-point ad-hoc communication between two nodes at a time in range of each other. We assume that a single node in each neighborhood has a back-haul network, and delegate to this node to further distribute updates inside the neighborhood.

A series of performance measuring experiments were conducted on the update distribution system when it executes on nodes with behaviors from always-on to mostly-off. We document how the distribution system behaves through a set of performance metrics. The results are very sensitive to the behavior of the nodes.

Index Terms—Arctic tundra, Resource-challenged environments, IoT, Cyber Physical System, Updates, Distribution, Scheduling

I. Introduction

An IoT cyber-physical system can be used to observe the state of the arctic tundra. Such observations are vital for research on climate change because it is first observable in the northern polar region. However, the arctic tundra is a resource-challenged environment, and the system cannot expect to have available the resources necessary to successfully operate there over time. These resources include energy, data networks, and humans. We focus on how the lack of these resources complicates updating the nodes after deployment. Nodes save energy by sleeping most of the time. Updates have to be done either by awaken sleeping nodes or when they are otherwise awake.

Without humans physically at the nodes, updates must be brought into a neighborhood over a back-haul network. While all nodes will have one or several local area network technologies on-node, at least one must also have a back-haul network available. This makes it possible for updates to both enter and be distributed inside a neighborhood.

For IoT systems the frequency of distributing updates after deployment vary. We identify two types of updating scenarios of special interest. Type 1 adds new functionality, repair bugs, and do configuration changes. The frequency of updates is low, perhaps every few weeks or months. Consequently, the distribution system can save energy by taking its time to distribute the updates. Type 2 distribute updates soon after deployment to do the initial update and configuration of the nodes, and to debug the software. The frequency is high for a short time after initial deployment. We expect that nodes need to be updated several times per hour or day for a period of a few days. To do so the distribution system must spend extra energy for a short time to reduce the time it takes to get the updates distributed to the nodes.

We report on an actual prototype system, the Update Distribution System (UDS), and its performance, for distributing updates to a neighborhood of nodes exhibiting a range of behaviors with regards to when they wakeup, and for how long they stay awake each time. Based on the results from the experiments and viewed in the context of the resource-challenged arctic tundra, we have identified suitable configurations to be used for the Type 1 and Type 2 update scenarios described above.

The paper is organized as follows. Sec. II details the related works. Sec. III details the update distribution system. Sec. IV details the experiments. Sec. V the results are discussed. Sec. VI discusses the node behaviours for suitability of use by the update distribution system. Sec. VII draws the conclusion.

II. Related work

Delay/Disruption Tolerant Networking (DTN) is NASA's solution for reliable inter networking for space missions [1]. It works within an environment that are subject to frequent disruption, possibly long delays and high error rates. Several DTN protocols exist which focus on the routing between nodes [2] [3] [4] [5]. Direct routing has the lowest energy consumption at the cost of lower delivery probability [5]. We use direct routing for the update distribution system. However, we document that the delivery probability is significantly impacted by the behaviour of the nodes.

Sleep/wake-up scheduling aims to minimize idle listening time. It can be divided into four categories: on-demand wake-up [6], synchronous wake-up [7], asynchronous wake-up [8], and duty cycling [9]. We document how synchronous and asynchronous sleep/wake-up scheduling impact the update distribution system.

Self-adapting sleep/wake scheduling has been proposed where a node chooses its action (sleep, transmit, listen) from a

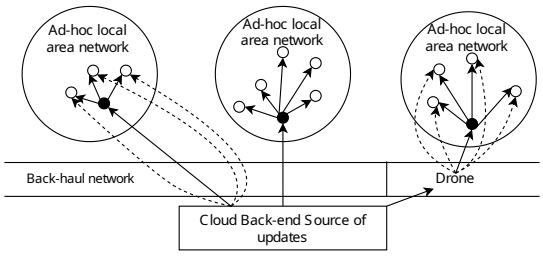


Fig. 1. The Distribution Model

probability distribution [9]. If a node decides to transmit and it fails, it will be less likely to choose to transmit the next time. One assumption in sleep/wake-up scheduling research is that idle listening is a waste of energy. We propose that updates should be distributed when nodes wake-up to do their regular tasks.

Manet and Mesh networks focus on the routing algorithm between nodes. The main characteristics are a dynamic topology, where each node is a router [10]. This paper does not focus on the routing between nodes, but rather the impact of sleep behaviours of nodes. The node topology is a star network.

III. Update Distribution System

In fig.1, three neighborhoods of nodes are shown. A solid black circle is a distribution node for a neighborhood. This node receives the content of an update either directly over a cloud back-haul network, or from a drone. Dotted lines for the cloud back-haul network case are possible communication paths which generally cannot be used because the nodes do not have such a network available. The dotted line from the drone to the nodes are possible ad-hoc paths to the nodes, but they cannot generally be used because nodes are sleeping for longer than the drone can be in the neighborhood.

The update distribution system (UDS) comprise a content distribution push side and a receiving node side. The design is based around a client/server model. The distribution node executes a server (UDS-S) actively pushing the content of updates out to its counterpart, the UDS-C, executing on the nodes.

The UDS-S has a discovery functionality to find nodes both in range and also running UDS-C. These nodes become the neighborhood. The UDS-C nodes detected by a discovery becomes targets for receiving updates. The UDS implementation comprises two Go programs. The UDS-S Go program executes at the distribution server node. The UDS-C Go program executes at each of the other nodes in the neighborhood.

For discovery, the UDS-S Go program has a list of the IP addresses to each node in the neighborhood (for this paper we ignore how this list is populated.) The UDS-S concurrently tries to get a response from all UDS-C nodes in this list. When responses come back from UDS-C at nodes which are awake, they become the targets to receive the content of an update. Presently, the implementation assumes that the server must establish an ad-hoc network with a single UDS-C node at a time. This to avoid needing to have a router functionality at one or several nodes consuming energy. Consequently, the server does sequential distribution of updates to each UDS-C in turn.

The UDS-C runs the concurrent HTTP server package supplied by Go. The client has two HTTP methods that it listens to. The first method is listening for a GET (the discover message) from the UDS-S. A response tells UDS-S that the UDS-C is awake. The second method listens for a POST request from UDS-S carrying the content of the update. UDS-C sends a response to the POST request if the transfer completed successfully.

The nodes in a neighborhood will behave in a number of ways. Always-on behavior is considered a base line behavior. It is a worst-case behavior with regard to energy consumption, we assume it is a best-case behavior with regards to minimizing the time it takes the server to send updates to all nodes. Can not realistically expect to be possible to do in a resource-challenged enviornment.

Fixed wakeup times for all nodes. This can be achieved by pre-determining the wakeup times and telling each node when to wake up. We assume that the node clocks in a neighborhood have been synchronized with each other.

Behavior where autonomous nodes themselves decides to wake up and for how long. Can result in low energy consumption, but also in increasing both the time the server spend trying to send out the updates, and the waiting times for the clients. However, we belive this behavior can be realstically used by a system of autonomous nodes deployed into a resource challenged environment.

UDS can modify the behavior of the nodes it executes on. We report on the behavior modification of extending the uptime once a node starts receiving an update. By doing so the nodes will not suddenly sleep while in the middle of sending or receiving data.

IV. Experiments

We use three node behaviors for the experiments. All the nodes have the same behavior per experiment. We conduct five performance measuring experiments on the UDS prototype while it executes on real computers (for details see later) following one of the three node behaviors.

To manage and control the experiments, we devised a software test-bench (for details see later). The test-bench is a set of tools. Some tools execute on a computer external to the experiment. This computer aids in doing the initial setup of the nodes and in collecting the results from an experiment. Other tools are executing on the nodes themselves. The UDS-S and UDS-C interacts with these tools to behave according to the behaviors.

A. Experiment Design

The five experiments are detailed in Table I. Node behavior for experiment 1 is that all nodes are always on. For experiment 2 and 3 all nodes wake up at approximately the same

TABLE I
FIVE EXPERIMENTS AND THREE NODE BEHAVIORS.
(S: IS SERVER, C: IS CLIENT)

	Wakeup times	Uptimes	Extend uptimes	Num. of wakeups
1	S: Always on C: Always on	S: Always on C: Always on	S: N/A C: N/A	S: N/A C: N/A
2	S: Every 30min C: Every 30min	S: 60sec C: 60sec	S: No C: No	S: 12 C: 12
3	S: Every 30min C: Every 30min	S: 60sec C: 60sec	S: Yes C: Yes	S: 12 C: 12
4	S: Always on C: Every 30 min	S: Always on C: 60 seconds	S: N/A C: No	S: N/A C: 12
5	S: Always on C: Every 30 min	S: Always on C: 60 seconds	S: N/A C: Yes	S: N/A C: 12

time every 30 minutes for 60 seconds each time. The clocks at each node are synchronized to each other. For experiments 4 and 5 the server node is always on, while the client nodes all wake up every 30 minutes for 60 seconds. Experiments 3 and 5 explore the effect on the performance metrics from letting the UDS modify a node's behavior by forcing extended uptimes on the node. A node's uptime is extended if it has an ongoing transmission of an update. The extension lasts until the transfer completes. The effect of extended uptimes is that nodes does not start sleeping in the middle of sending or receiving an update. Each experiment ends when all nodes with an UCS-C have received the content of an update.

The node behavior used for experiments 2 and 3 is inspired by actual deployments done to the arctic tundra. Nodes typically wake up every hour to do measurements. They also wake up when interesting events happen, like an animal passing ahead of a proximity sensor. The content of the update is not of relevance for the results reported on in this paper, only the size is. The size of an update is f xed at 1 MB. The reason for f xing the size to 1 MB is that the nodes we have developed and deployed to the arctic tundra execute software of 1 MB or less. We expect the size of updates to increase as the nodes become more advanced. However, we can always distribute the updates in 1 MB chunks. The results we report on is therefore useful even when the size of updates increases.

B. Metrics

We report on the behavior of the UDS when it executes on nodes behaving as previously described. The behavior of UDS is quantif ed through a set of performance metrics. Each experiment has a clock initialized to zero at the beginning of the experiment. The clock increases by one every second.

For the experiments, the following is of relevance when def ning the performance metrics:

Each experiment ends when all nodes with an UDS-C have received the content of an update. A wake-up event happens when a node starts or resumes execution of either UDS-S or UDS-C. The wakeup time is the clock value when a wake-up event happens during an experiment.

A sleep event happens when a node temporarily suspends or pauses its execution of either UDS-S or UDS-C. A node is def ned to be up while it executes either UDS-S or UDS-C.

A complete distribution is a distribution of an update sent from the server and fully received by a client. An incomplete distribution is a distribution of an update sent from the server but not fully received by the client.

The uptime is the elapsed time from a wake-up event until a sleep event. It is measured by subtracting the time of the sleep event from the time of the wake-up event.

A number of performance metrics quantify the behavior of the UDS distribution system: The accumulated server uptime is the sum of all its uptimes during an experiment. The accumulated client uptime is the sum of all its uptimes during an experiment. The server completion time is the elapsed time on the experiment clock until the server has completed distributions to all clients. The client completion time is the elapsed time on the experiment clock until a client has a complete distribution. The number of complete distributions per time unit is a measure of how fast the updates on average spread in a neighborhood. The number of incomplete distributions per time unit is a measure of the effectiveness of the system in spreading updates. Higher number means lower effectiveness and more time spent without getting updates to clients. The number of complete distributions per time unit during accumulated server uptime can be interpreted as how effective updates are spread during server uptime. Higher numbers means less energy spent per distribution. The number of incomplete distributions per time unit during accumulated server uptime can be interpreted as how much server uptime is wasted without getting updates delivered. High numbers means more wasted uptime and consequently more wasted energy.

C. Experiment Testbench

The UDS expects to execute on nodes that are sleeping and waking up. This complicates controlling the experiments. Therefore, while the physical nodes are always on, the testbench emulates node sleep and awake periods. The testbench tools, see Fig. 2, include the Timekeeper and the Available State Coordinator (ASC). The Timekeeper and the ASC executes at each node. The Timekeeper is responsible for scheduling wake-up and sleep events for the node it resides on. The ASC is responsible for emulating if a node is awake or not. UDS will interact with ASC to get to know if the node is meant to be awake or not.

The Timekeeper has two event alarms, wakeup and sleep. When an event alarm triggers, the Timekeeper will notify the ASC of the event. The ASC will process the event. If the current Node awake state is sleeping, upon receiving a wakeup event the state will change to awake. The sleep event is dependent on if Extend uptime is active or not. A sleep event will be denied as long as Extend uptime is active in the ASC. The status of Extend uptime is changed by the UDS.

D. Hardware and Software Platform

The hardware platform comprises 29 computers. The computers are Raspberry Pi 3B+, running Raspbian GNU/Linux 9 (Stretch, under Linux 4.19.58-v7+ armv7l). One computer

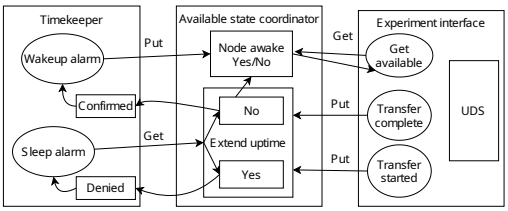
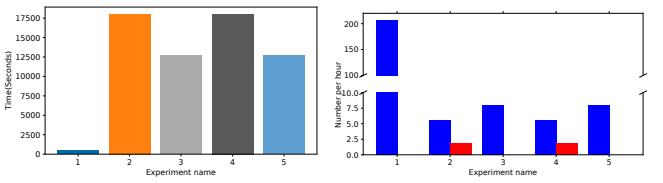


Fig. 2. Experiment testbench. Timekeeper and Available state coordinator executes on each node concurrently with the UDS.



(a) Server completion time

(b) Number of complete and incomplete distributions per hour. Blue is complete. Red is incomplete.

Fig. 3. (a) Server completion time and (b) Number of complete and incomplete distributions per hour.

executes the UDS-S, the remaining 28 computers execute a UDS-C each. The computers are interconnected between a wired network and a wireless network.

A wired Ethernet is used to manage the nodes and the experiments. It comprises a RouterBOARD 750 router and two Netgear GS116 switches. A WIFI network is used to provide communication between the UDS-S and the UDS-C nodes. It comprises a single Asus RT-AC66U (with QoS off) access point to which all 29 computers are associated. The communication between the UDS-s and the UDS-C nodes are isolated to the wireless network. However, the network we expect to be used on the arctic tundra is not necessarily a WIFI network with a central router. Examples of current radio technology available for IoT systems include NB-IoT (150 Kbps) and LTE CAT M1 (1 Mbps). Therefore, on top of the WIFI network, we emulate a more realistic bandwidth for the arctic tundra. We use Wondershaper[1] to restrict the WiFi bandwidth to 512 Kbps.

The experiments also assumes that the communication between the server and the clients happen as an ad-hoc network between server and a singel client at a time and with no central router. Consequently we restrict the server to do a transfer of an update to a single client at a time.

V. Results

A. Server completion time

The server completion time is the elapsed time on the experiment clock until the server has completed distributions to all clients, see f g. 3a. Experiment 1 takes about 600 seconds (10 minutes) to complete all distributions. All nodes are awake during the experiment. Experiment 2 and 4 take 18000 seconds (5 hrs) to complete. The nodes wake up at f xed 30 minutes intervals and have 60 seconds uptime after each wakeup. Experiment 3 and 5 take 12700 seconds (3.5 hrs) to complete. The nodes wake up at f xed 30 minutes intervals and have 60 seconds uptime after each wakeup. In addition, once a transfer to a node starts, the node uses extended uptime until the transfer completes.

The always-on behavior of the nodes has a very signif cant effect in achieving a low completion time. This is because both the server and all nodes are ready all the time to participate in the distribution of updates. When nodes sleep most of the time and they wake up at overlapping f xed intervals, applying the technique of extending uptimes has a signif cant effect on reducing the completion time. This is because the server can only establish an ad-hoc connection with a single client at a time, and consequently transfer the update to a single client at a time. With the chosen size of the update and the bandwidth of the network, the server can complete at maximum three transfers before its 60 seconds of uptime expires. It will start on the fourth, but both it and the client will go to sleep in the middle of the transfer. Extending the uptime for both has the effect of getting a fourth transfer across before the nodes sleep.

B. Number of complete and incomplete distributions per hour

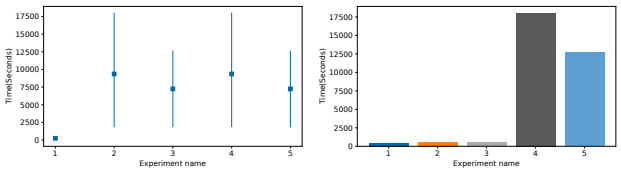
The number of complete distributions per time unit is a measure of how fast the updates on average spread in a neighborhood. The number of incomplete distributions per time unit is a measure of the effectiveness of the system in spreading updates. Higher number means lower effectiveness and more time spent without getting updates to clients. Fig. 3b shows the results. Experiment 1 does around 200 distributions/hour. Experiments 2 and 4 does 5-6, and experiments 3 and 5 does 8. Experiment 2 and 4 does around 2 incomplete distributions/hour.

Having nodes always-on is signif cant in spreading an update fast. When nodes sleep most of the time and they wake up at overlapping f xed intervals, the updates spread at least one order of magnitude slower than for the always-on case. This is because the server is only allowed to complete a distribution to three or four clients every 30 minutes.

When the server is always-on and clients sleep most of the time, the updates spread at the same rate as when all nodes sleep most of the time and they wake up at overlapping f xed intervals. Resulting in idle waiting for the server.

While applying the technique of extending uptimes has some effect on increasing how fast updates spread, it is still more than one order of magnitude slower than the always-on node behavior. If we do not apply extended uptimes, there will be one incomplete distribution for every wakeup. Ex. if the uptime is 20 seconds. Can complete 1 distribution in 17 seconds and start a new distribution which will be incomplete. If the uptime is 40 seconds. Can complete 2 distributions in 34 seconds and start a new distribution which will be incomplete.

[1] https://github.com/magnif c0/wondershaper



(a) Client completion time. Square points is average for all clients. Lines indicate minimum and maximum completion time.

(b) Accumulated server uptime.

Fig. 4. (a) Client completion time and (b) Accumulated server uptime.

C. Client completion time

The client completion time is the elapsed time on the experiment clock until a client has a complete distribution. Fig. 4a shows the average completion time over all clients per experiment. For experiment 1 the average client completion time is 252 seconds. For the other experiments the average client completion time is two orders of magnitude higher than for experiment 1. Experiment 2 and 4 have average client completion times of 10 000 seconds (almost 3 hours). Experiment 3 and 5 are lower by about 1/4 at 7500 seconds (about 2 hours). Experiments 2, 3, 4 and 5 have the same minimum client completion time seen by the first client to complete a distribution. Applying extended uptimes reduces the average completion times by 1/4.

D. Accumulated server uptime

The accumulated server uptime is the sum of all its uptimes during an experiment. Fig. 4b shows the results. Experiments 1, 2 and 3 have accumulated server uptimes being close, 487, 558 and 495, respectively. Compared to these, experiments 4 and 5 have an order of magnitude higher accumulated server uptime at 17500 and 12500 seconds, respectively. The results for 1, 2 and 3 are similar because the server in all three cases only needs to be awake approximately equally long. For experiment 1 the server is awake all the time, but the experiment ends fast.

For experiment 2, while the experiment lasts for much longer, the server sleeps most of the time. When it wakes up, it manages to three complete distributions each time during the uptime. For experiment 3 the same situation as for experiment 2 happens. However, now the server manages four complete distributions each time by extending the uptime. So experiment 2 sees shorter uptimes, but more wakeups, while 3 sees longer uptimes, and fewer wakeups. For each, this adds up to the accumulated server uptime being close.

Experiments with the server always on (4 and 5) accumulates significantly more server uptime compared to experiments where the server wakes-up every 30 minutes (2 and 3). Experiment 5 has a lower accumulated server uptime than experiment 4 because 5 applies extended uptimes for the clients. This makes the experiment end sooner.

E. Number of complete and incomplete distributions per hour over the accumulated uptime for the server

The number of complete distributions per time unit during accumulated server uptime can be interpreted as how effective updates are spread during server uptime. Higher numbers means less energy spent per distribution. The number of incomplete distributions per time unit during during accumulated server uptime can be interpreted as how much server uptime is wasted without getting updates delivered. High numbers means more wasted server uptime and consequently more wasted energy. Fig. 5a shows the results. Experiments 1, 2 and 3 can do 206, 180, and 203 complete distributions per hour of server uptime, respectively. Experiments 4 and 5 can do an order of magnitude less complete distributions at 6, and 8, respectively.

For experiments 4 and 5 when the server is awake much of the time is used idle waiting for the clients to wake up. For experiments 1, 2 and 3 when the server is awake it has a client to distribute to at all times. There is no idle wait time for the server. Only experiment 2, and 4 have incomplete distributions, at about 60, and 2, respectively. For experiment 2 each time the server wakes up, it will have one incomplete distribution. In experiment time this happens every 30 minutes, but in uptime this happens every 60 seconds. For experiment 4 the server is always-on but it has to wait for the clients to wake up. For each time the clients wake up it will have one incomplete distribution. In experiment time and in uptime for the server this happens every 30 minutes.

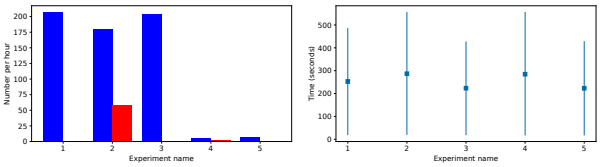
F. Accumulated client uptime

The accumulated client uptime is the sum of all its uptimes during an experiment. Fig. 5b shows the average accumulated client uptime over all clients per experiment. It also shows the spread of accumulated client uptimes. Overall, even if there are small variations, all experiments measure rather similar accumulated client uptimes. All experiments also have about the same accumulated client uptime for the first client to complete a distribution.

Experiment 1 has a slightly higher value than experiments 3 and 5 because the clients are always-on. However, the accumulated client uptime is still close to the results for 3 and 5 because experiment 1 ends faster with all distributions complete. Experiment 2 and 4 have the highest average value and highest max value, while experiment 3 and 5 both have the lowest average value as well as the smallest max values. This is because 3 and 5 both apply extended uptimes. Extended uptimes result in a shorter experiment completion time.

VI. Discussion

We rank the node behaviors and their impact on UDS based on the results from previous sections. We have two types of updating scenarios. Type 1 adds new functionality, repair bugs and do configuration changes. The frequency of updates is low. Type 2 includes debugging the software and adds the initial configuration of the nodes. The frequency of updates is high. The results show that with regards to accumulated uptimes,



(a) Number of complete and incomplete distributions per hour over the accumulated uptime for the server. Blue is for complete distributions, Red is for incomplete

(b) Accumulated client uptime. Square points is average for all clients. Vertical lines indicate the spread of accumulated client uptimes.

Fig. 5. (a) Number of complete and incomplete distributions per hour over the accumulated uptime for the server and (b) Accumulated client uptime.

and by implication the energy consumption, it does not matter a lot for the clients which of the five node behaviors that we have explored are used. However, 3 and 5 are slightly preferable to clients occupied with their own accumulated uptimes and energy consumption. On the other hand, for the server, experiments 4 and 5 are costly, and with 4 as the most costly. All in all, the experiments point at the node behavior used in experiment 3 as the one best suitable for Type 1. It provides both the lowest client and server cost with regards to accumulated uptimes and therefore also energy.

Other performance metrics explores other aspects with node behavior 3. It does well in all of them compared with 2, 4, and 5. Only versus 1 is it lacking in that it has much longer client and server completion times. However, for a Type 1 deployment, node behavior 3 is the one to use for nodes in a resource-challenged environment. For Type 2, both low server and client low completion times are needed. This is achieved with increased cost in accumulated uptime and energy usage. Node behavior 1, always-on nodes, is clearly the most suitable. Even if the cost is high, it will only be used for a short time. For Type 2, when the number of nodes in a neighborhood increases, the node behavior best suited will be the one able to spread the updates fast. Again, this is node behavior 1. When more dimensions than the few we have explored are explored, other node behaviors will turn out to improve on behavior 1 and 3 in most or all ways.

VII. Conclusion

The arctic tundra is highly sensitive to climate change and the impact of climate change will first be seen there. IoT systems must be deployed in resource-challenged environments, like the arctic tundra, to gather observations needed for climate research. In this paper we focus on some aspects of the distribution of updates to IoT system nodes. We report on the performance-related effect a few different node behaviors have on the distribution of updates. The node behaviors range from always-on to mostly-off. We conduct performance measuring experiments on an actual prototype system executing on the nodes. We explore the effect of letting the update distribution system extend the uptime of nodes to prevent them from sleeping in the middle of receiving an update.

Having always on nodes consumes energy all the time. The time to complete distribution of updates to all nodes is the lowest compared to other node behaviors. Because the system rapidly finishes doing the updates the uptime and therefore the energy consumed are the lowest. Having nodes on all the time can be activated for a short period of time while doing time sensitive updates. For a resource-challenged environment, always-on behavior cannot be used all the time.

If the nodes wakes up on a fixed schedule, the clients will see much longer waiting times before they receive an update. This is improved by applying the technique of letting the system extend the uptime for the nodes in the middle of sending or receiving updates. For situations where long waiting times before all clients receive an update is acceptable, there is no need to activate always-on node behavior.

Future work includes exploring more node behaviors and their performance-related effect on the update distribution system.

Acknowledgment

This work is supported by the Distributed Arctic Observatory (DAO) project supported by the Research Council of Norway (RCN) IKTPluss program, project number 270672.

References

[1] A. Schlesinger, B. M. Willman, L. Pitts, S. R. Davidson, and W. A. Pohlchuck, "Delay/disruption tolerant networking for the international space station (iss)," in 2017 IEEE Aerospace Conference, March 2017, pp. 1–14.

[2] A. Vahdat and D. Becker, "Epidemic routing for partially-connected ad hoc networks," Duke University, Tech. Rep., 2000.

[3] A. Lindgren, A. Doria, and O. Schelén, "Probabilistic routing in intermittently connected networks," SIGMOBILE Mob. Comput. Commun. Rev., vol. 7, no. 3, p. 19–20, Jul. 2003. [Online]. Available: https://doi.org/10.1145/961268.961272

[4] T. Spyropoulos, K. Psounis, and C. S. Raghavendra, "Spray and wait: An efficient routing scheme for intermittently connected mobile networks," in Proceedings of the 2005 ACM SIGCOMM Workshop on Delay-Tolerant Networking, ser. WDTN '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 252–259. [Online]. Available: https://doi.org/10.1145/1080139.1080143

[5] E. A. A. Alaoui and M. Lamhamdi, "Study of the energy performance of dtn protocols," in 2017 Intelligent Systems and Computer Vision (ISCV), April 2017, pp. 1–7.

[6] R. Piyare, A. L. Murphy, C. Kiraly, P. Tosato, and D. Brunelli, "Ultra low power wake-up radios: A hardware and networking survey," IEEE Communications Surveys Tutorials, vol. 19, no. 4, pp. 2117–2157, Fourthquarter 2017.

[7] A. Keshavarzian, H. Lee, and L. Venkatraman, "Wakeup scheduling in wireless sensor networks," in Proceedings of the 7th ACM International Symposium on Mobile Ad Hoc Networking and Computing, ser. MobiHoc '06. New York, NY, USA: Association for Computing Machinery, 2006, p. 322–333. [Online]. Available: https://doi.org/10.1145/1132905.1132941

[8] R. Zheng, J. C. Hou, and L. Sha, "Asynchronous wakeup for ad hoc networks," in Proceedings of the 4th ACM International Symposium on Mobile Ad Hoc Networking & Computing, ser. MobiHoc '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 35–45. [Online]. Available: https://doi.org/10.1145/778415.778420

[9] D. Ye and M. Zhang, "A self-adaptive sleep/wake-up scheduling approach for wireless sensor networks," IEEE Transactions on Cybernetics, vol. 48, no. 3, pp. 979–992, March 2018.

[10] S. Bhushan, A. K. Singh, and S. Vij, "Comparative study and analysis of wireless mesh networks on aodv and dsr," in 2019 4th International Conference on Internet of Things: Smart Innovation and Usages (IoT-SIU), 2019, pp. 1–6.