# RATFAT: ReAl-Time FAT for Cooperative Multitasking Environments in WSNs

Sebastian Schildt, Wolf-Bastian Pöttner, Felix Büsching, and Lars Wolf
Institute of Operating Systems and Computer Networks
Technische Universität Braunschweig,
Braunschweig, Germany
Email: [schildt|poettner|buesching|wolf]@ibr.cs.tu-bs.de

***Abstract*—Today, many sensor nodes are equipped with a microSD slot to provide a cost effective way to store large amounts of data. When using the FAT file system, data collected by a node can be easily read by a PC without the need for any special software or communication protocol. While several FAT implementations for microcontrollers do exist, they are not suited for real-time applications. For a WSN in an industrial scenario where a node needs to run a closed loop control program, logging to a non-real-time capable persistent storage system is not an option.**

**In this paper we present RATFAT, an efficient implementation of a flexible real-time capable FAT file system for Contiki, that can be used for applications requiring real-time guarantees.**

## I. Introduction

For cost and energy efficiency reasons wireless sensor networking applications are mostly built using relatively weak hardware platforms. A common sensor node is often powered by an 8 bit microprocessor and has a few kiB of RAM and a few tens of kiB of flash storage available. The nodes usually run cooperative multitasking runtime systems such as TinyOS[1] or Contiki[2]. In many applications there is the requirement that the system needs to be able to store data persistently. Using a microSD card is a cost-effective and easy method to store data: microSD cards are supported by modern sensor nodes, they are cheap and provide almost “unlimited” storage from the perspective of a small 8 bit microcontroller.

In several WSN applications, a sensor might be disconnected for long-term monitoring, and data will be gathered by a ferry [6] later, or a node just caches sensed data during temporary transient link failures. In addition, some systems may collect data over a longer period (e.g., about link quality) and store those information to persistent storage. Later, the collected raw data can be processed en-bloc producing a short summary that can be sent via the wireless interface [4].

Real-time capabilities are needed if a sensor network uses a TDMA scheme for communication or if a node controls actors in a closed loop control. This is usually the case for WSNs deployed in industrial settings [3]. Another field with real-time demands are healthcare applications. Here people often carry small data loggers for long-term monitoring. These devices must guarantee a stable sampling rate. In addition, medical devices might perform continuous classification of sensor data to detect emergency situations such as people falling [2]. Logging data to persistent storage should not interfere with these time-critical operations.

The Contiki operating system already provides the COFFEE file system [7] which can be used for persistent storage of data. COFFEE, however, cannot work in real time environments, as its write times are unpredictable and Contiki does not preempt its tasks. RFS, a real-time-capable file system for Contiki has been presented in [5]. However, RFS was optimized for on-board flash storages. RFS is a Ring File System designed for logging purposes, also providing wear leveling features. However, SD cards include a flash translation layer that deals with wear leveling. Just like COFFEE, RFS is not a good fit for huge SD cards, and the data stored by these file systems cannot be read on a PC without developing specialized software.

There are several implementations of FAT for small embedded systems available. For example *FatFs*[3] is a highly portable FAT implementation for small microcontrollers such as AVR, PIC or 8051 CPUs. For the popular Arduino project[4] *tinyFAT*[5] allows reading and writing FAT volumes on standard SD cards.

In this paper we present RATFAT, a FAT implementation for Contiki. RATFAT offers a similar API to COFFEE so it can act as a drop-in replacement for many applications. Due to its FAT compatibility, volumes written by RATFAT can be read on common PCs using any operating system. In contrast to other FAT implementations for microcontrollers, RATFAT can be used in real-time applications, even when using a cooperative multitasking systems such as Contiki. To achieve this, RATFAT breaks up file system operations into multiple atomic operations. This allows other processes to be scheduled between the execution of these different substeps that represent a file system operation. RATFAT has knowledge about the duration of various operations, which allows it to schedule the maximum number of substeps without violating a predefined deadline. This enables the developer to use the optimal tradeoff between file system throughput and real-time constraints for a specific application.

We implemented RATFAT on the INGA sensor node [1]

[1]http://www.tinyos.net/
[2]http://www.contiki-os.org/
[3]http://elm-chan.org/fsw/ff/00index_e.html
[4]http://www.arduino.cc
[5]http://henningkarlsen.com/electronics/library.php?id=37

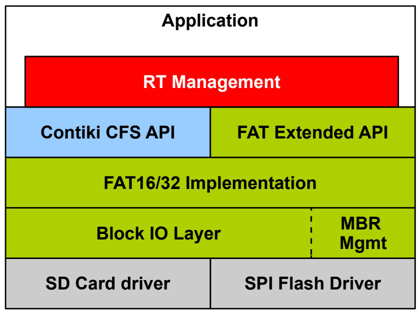


Figure 1. RATFAT Architecture

running Contiki. RATFAT should run on any node supported by Contiki that offers a suitable storage.

The remainder of this paper is organized as follows: Section II introduces RATFAT's architecture. Section III sheds some lights onto RATFAT's real-time capabilities and in section IV we compare RATFAT to COFFEE and quantify how RATFAT improves the real-time capability of a system and how much additional overhead is added by the real-time capability. Finally, in section V we conclude and give some final remarks.

## II. RATFAT DESIGN

Figure 1 gives an overview of RATFAT's architecture. The grey blocks at the bottom are drivers for storage devices that should be provided by the platform. For the RATFAT implementation, the Contiki SD card driver for INGA was extended to support SDHC cards and the functionality to determine the size of the inserted card has been added. Above the device driver layer RATFAT introduces a block abstraction layer: As the FAT file system specification is based on the device abstraction of a linear array of blocks, this is what the Block IO layer provides. Closely related to the Block IO layer is a component that can parse Master Boot Record (MBR) style partition tables. It is common to use a partition table when formatting SD cards on a PC. In this case the file system resides on a device such as `/dev/sdX` in Linux, indicating the first partition of device sdX. Alternatively, the file system can be put directly on the device (`/dev/sdX` on Linux) without using partitions. The MBR component uses the Block IO layer to read the partition table from a device, and the Block IO layer uses the MBR component to determine partition boundaries.

The actual implementation of the FAT file system sits on top of the block layer. Both, the FAT16 and FAT32 variants are supported. As FAT12 is not in widespread use anymore (it has been used mainly on floppy disks), it is not supported. Due to resource constraints, RATFAT handles only the classical DOS 8.3 naming scheme. However, since long names reside in an extended attribute and all files on a valid FAT system always possess an 8.3 alias, this is not a problem. If a file has a long filename, that name will be ignored by the implementation. The file can still be accessed by using its unique 8.3 name. As most WSN nodes have constrained resources, RATFAT takes the design choice of only using one 512 byte buffer for its operations. This increases performance significantly compared to a buffer-free FAT implementation. While this precludes performance gains due to more advanced sector caching, it is a reasonable tradeoff considering that a typical WSN node only has limited amounts of RAM (16 kiB for an INGA node).

The FAT implementation supports Contiki's Common File System (CFS) API, which means it can be used as a drop-in replacement for other CFS compatible file systems such as COFFEE. Listing 1 shows the supported CFS API calls.

As the CFS API is the lowest common denominator, RATFAT exports an extended API providing access to additional functionality. This includes functions to format an SD Card and the ability to create or remove directories. RATFAT's extended API is shown in Listing 2.

```
uint8_t cfs_fat_mount_device(struct
    diskio_device_info *dev);
void cfs_fat_umount_device();
uint16_t cfs_fat_get_last_date(int fd);
uint16_t cfs_fat_get_last_time(int fd);
uint16_t cfs_fat_get_create_date(int fd);
uint16_t cfs_fat_get_create_time(int fd);
uint32_t cfs_fat_file_size(int fd);
int cfs_fat_rmdir(char *);
int cfs_fat_mkdir(char *);
void cfs_fat_sync_fats();
int cfs_fat_mkfs(struct diskio_device_info *
    dev);
```

Listing 2. RATFAT extended API

Of special interest is the `fat_sync` function: A FAT volume contains two copies of the File Allocation Table. The rationale is that, should one of the tables get damaged, the backup can still be used. However, since synchronizing both FATs can be a very costly operation, especially when operating with little buffer space, RATFAT chooses to only keep one FAT up to date. The user can choose to sync FATs when it is convenient, for example during unmounting of a volume. This is not a compatibility problem, because in case of an error free volume, PC FAT implementations will just choose the primary FAT and ignore the state of the secondary FAT.

By using the APIs exposed by RATFAT, an application can read and write FAT volumes. However, the resulting application will not be real-time-capable. Depending on the requested operation and the state of the volume, a read or write operation will take varying amounts of times, which are not predictable for the application. RATFAT's real-time-capability is provided by a special real-time management component, that sits on top of the RATFAT core and that modifies the way RATFAT is handling application requests.

## III. REAL-TIME MANAGEMENT

A real-time system must guarantee certain response times for an application. A file system operation such as creating files, reading or writing data can take different amounts of time, depending on whether the data is distributed across different sectors, whether some meta data structures need to be modified, and whether the operation can be performed

```
int cfs_open(const char *name, int flags) ;
void cfs_close(int fd);
int cfs_read(int fd, void *buf, unsigned int len);
int cfs_write(int fd, const void *buf, unsigned int len);
cfs_offset_t cfs_seek(int fd, cfs offset t offset, int whence);
int cfs_remove(const char *name);
int cfs_opendir(struct cfs dir *dirp, const char *name);
int cfs_readdir(struct cfs dir *dirp, struct cfs dirent *dirent);
void cfs_closedir(struct cfs dir *dirp);
```

Listing 1. Supported CFS API calls

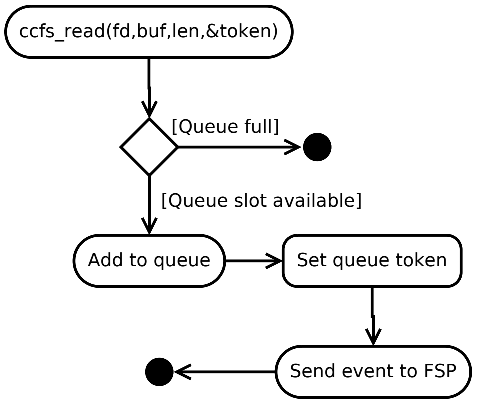


Figure 2. Queueing of file system requests

in the cache. When using RATFAT through the standard API, the calling process is blocked until the file system operation finishes. Thus, to guarantee real-time constraints, a process must always expect worst-case latencies. This often requires large time constants and precludes fulfilling application requirements. The real-time management component of RATFAT shifts file system operations to a separate *process*, the file system process (*FSP*).

As Contiki, similar to most runtime environments for small microcontrollers, only supports cooperative multitasking, not much is gained by this strategy yet: Once a process is scheduled, it will run to completion before the next process can be scheduled. Thus, the system is still blocked until an operation completes. The *FSP* works around this limitation however: It splits operations into smaller, atomic operations, which are executed through Contiki's scheduler. Thus, a write request issued by an application will be translated to several smaller operations that need to be performed. When an application issues a request to the file system, this request will be queued by RATFAT in constant time and the Contiki scheduler is used to wake up the *FSP*. Queuing operations take constant times and are not subject to any operations on the actual medium. Once scheduled, the *FSP* will execute some steps of pending operations and then reschedule itself. This gives other applications the ability to run and allows real-time applications to meet their deadline.

Based on this scenario the following requirements have been defined for RATFAT's real-time capabilities:

- Calls to RATFAT should never block.
- A cooperative-multitasking friendly file system process (*FSP*) is used to perform RATFAT operations.
- The order of operations on the file system must be retained.
- A calling process needs to be informed when an operation on the file system finished.
- The time needed for RATFAT operations should be deterministic.

The real-time version of RATFAT exports the same functions as the normal RATFAT implementation. The functions are prefixed with `ccfs` (*concurrent* cfs) instead of `cfs`. All `ccfs` functions return immediately. They will just add an entry representing the requested operation to the *FSP*'s queue, and send the process a signal. An example for queuing a `read` request is shown in Figure 2. The `token` parameter will be filled by RATFAT with an identifier that allows tracking of the operation's state. A RATFAT request is in the `QUEUED` state when it is added to the *FSP* queue. When the *FSP* starts working on the request its entry transitions to the `INPROGRESS` state. The caller of a ccfs function can query the state of a dispatched file system request at any time. Once a request is finished, the calling process will be sent an event to inform it about the result of the operation.

Usually, sensor nodes will use the SPI interface of SD cards. Since only one device can be active on the SPI bus, the *FSP* should not be interrupted while performing any SPI transactions. Doing so would result in undefined behavior and has to be avoided. Therefore, RATFAT has SD block read and write operations that are executed over SPI as atomic operations that cannot be interrupted. File system operations that require multiple block read and write operations are split up into multiple atomic operations.

RATFAT has a user-configurable constant `FAT_COOP_SLOT_SIZE_MS` that defines the maximum amount of time in milliseconds that the *FSP* is allowed to run for one invocation. The *FSP* has knowledge about the duration of different operations such as reading a byte from the SPI or writing a sector to the SD card. This enables the *FSP* to reliably predict the duration of pending operations. Based on this information, as many operations as possible are performed during one invocation. The *FSP* will make sure it yields the CPU before the next Contiki timer fires or latest after `FAT_COOP_SLOT_SIZE_MS` ms. One iteration

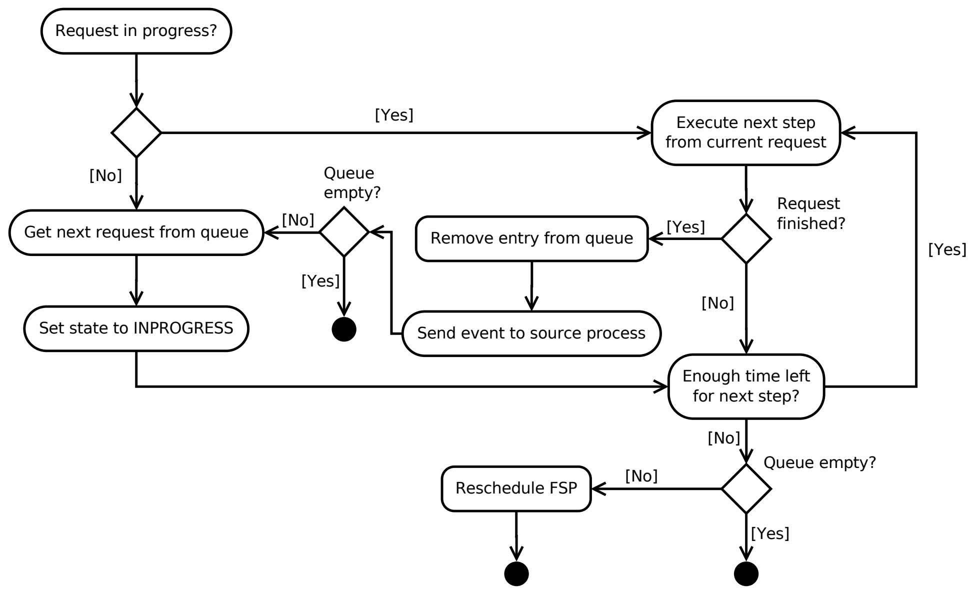


Figure 3. *FSP* process

of the *FSP* process is shown in Figure 3. By configuring `FAT_COOP_SLOT_SIZE_MS` the user can trade between responsiveness of the system (smaller slot size) and less overhead for context switching (longer slot size).

When the *FSP* process is scheduled, it will check whether a request is currently in progress. If there is a request in state `INPROGRESS` the *FSP* will execute the next step for that process, otherwise it selects the first step from the next request in the queue. The *FSP* will continue to execute steps if there are still pending requests in the queue and as long as there is still enough time left to execute the next step. When the last step of a request has been executed the calling process is signaled. If the current time slot does not allow execution of the next step, the *FSP* will reschedule itself. And if the *FSP* processed the last request in the queue it will terminate itself until it is triggered again by an external application adding a new request to the queue.

## IV. Evaluation

The evaluation was performed on a INGA sensor node [1]. The node features an Atmel ATmega 1284p processor with 16 kiB of integrated RAM and 128 kiB of integrated flash storage. INGA supports micro SD cards attached via SPI, which are used for RATFAT and COFFEE in this evaluation. For the results shown here we used a 2GB microSD card from Transcend. We found in our experiments, that writing data to the SD card is the most critical operation since write times vary and potentially involve writing (and reading) multiple blocks for book keeping purposes. Moreover, the most convenient reason for having FAT in a WSN is to be able to write data ”in the field” and easily read it on a PC afterwards. Therefore, we concentrate solely on write operations in the evaluation.

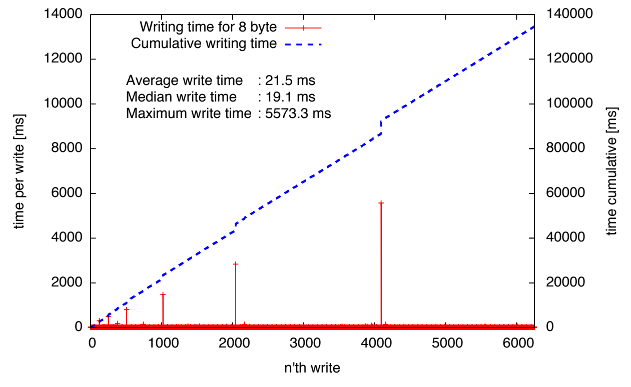
### *A. Basic throughput and timing tests*

As a baseline test, we did a very basic evaluation: For both COFFEE and RATFAT we created a file system on the SD card with default parameters and wrote 50 kiB data to a file. We choose to write data in 8 bytes steps, as small writes are common when logging sensor data. For COFFEE, we have limited the file system size to 500 kiB since COFFEE does not scale well with increasing file system sizes.

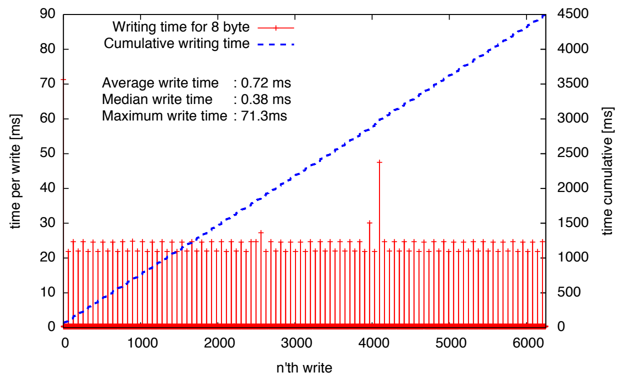
The results for COFFEE can be seen in Figure 4(a). Most writes take around 20 ms, however every once in a while writing takes significantly longer (up to 5500 ms in this test case). This is due to the fact that we exceed the maximum amount of reserved blocks for a file. In this case COFFEE creates a new bigger file, and copies the complete data from the old file to the new file. Once the new file’s size is exceeded again, the data is duplicated once more, which will take longer each time the file is extended, as there is more data to copy. This is a limitation in COFFEE, which can only be overcome if the developer knows the maximum size of a file before creating it.

We performed the same test with RATFAT in non-real-time mode by using the CFS API directly. The results can be seen in Figure 4(b). Average write times are well below 1 ms. Most writes go directly to RATFAT’s 512 byte sector cache. Once a sector is filled it needs to be written to the SD card. This will take about 25 ms. The maximum time to perform a write operation occurs at 32 kiB, taking 71 ms. This is most probably due to housekeeping of the FAT. Overall this shows, that for RATFAT maximum write times are more tightly bounded whereas COFFEE shows a trend of increasing write times.

When writing to the microSD card COFFEE reaches an overall transfer rate of 0.36 kiB/s. RATFAT reaches 10.9 kiB/s. While RATFAT performance is more consistent compared



(a) COFFEE



(b) RATFAT non-real-time

Figure 4. Time to write 8 bytes to the SD card

to COFFEE, it can still not guarantee real-time requirements. Occasionally a file system needs to perform some management and clean-up tasks. The main problem is, that even in this simple test case both file systems occasionally need a write time that is two magnitudes higher than the median write time to execute a write due to internal housekeeping procedures. So, an approximation to deal with that would be to schedule a time that is more than 2 orders of magnitude larger than the average execution time for file system operations.

In the next section we will take a look how RATFAT's behavior influences a real-time process, and how the real-time capability of the system is improved by using RATFATs real-time management module.

## B. Real-Time Performance

In this test we study the real-time capability of RATFAT. For comparison we will use the RATFAT API directly. We omit COFFEE from this comparison as the previous section already showed that it is definitively unsuitable for real-time operation.

To get an idea how much additional overhead the real-time management adds, we performed the write tests again with RATFAT's real-time API. Instead of writing using the blocking API, write requests are queued and the *FSP* process is triggered to perform the actual write. Only after RATFAT signals the successful completion of the write request, the next write will be queued. In Figure 5 the write times for the blocking API are compared to the real-time API. The overall throughput is decreased to 6.0 kiB/s which is still more than 50 % of the non-RT version. The "RATFAT-rt queueing" line shows the time it takes to enqueue a write request. This operation will be in the critical path for a real-time process. It should have a short and predictable runtime. In the experiment the average queuing time is $0.12 \pm 0.03$ ms.

In the next experiment we evaluate how RATFAT performs on a loaded system. The setup of this experiment is shown in Figure 6. We assume that we have a real-time process that needs to be triggered regularly to sample some data. We assume a sampling interval of $T$. Every $n$ sampling

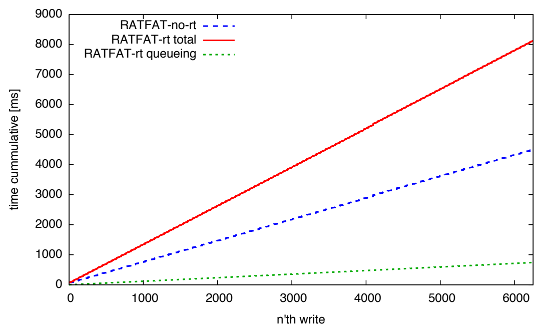


Figure 5. Cumulative write time comparision

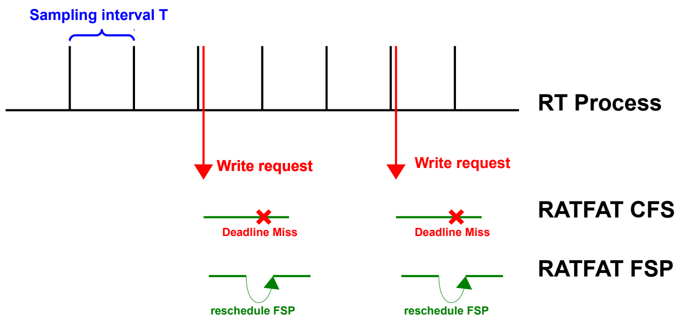


Figure 6. RATFAT's real-time managments splits long running file system operations into multiple atomic operations

intervals the process will trigger a write to a file. In a real application, the write operation might be used to log some data or results calculated using the sampled data. First, we perform this experiments using RATFAT's blocking CFS API. In this cases, if an operation on the file system takes longer than the sampling interval $T$, the real-time process will miss a deadline. In the second experiment, we use RATFAT's real-time management component. This should split the file system operations into smaller parts, allowing the real-time process to meet its deadline.

In this experiment we set $n$ to 5 and write 80 bytes with each

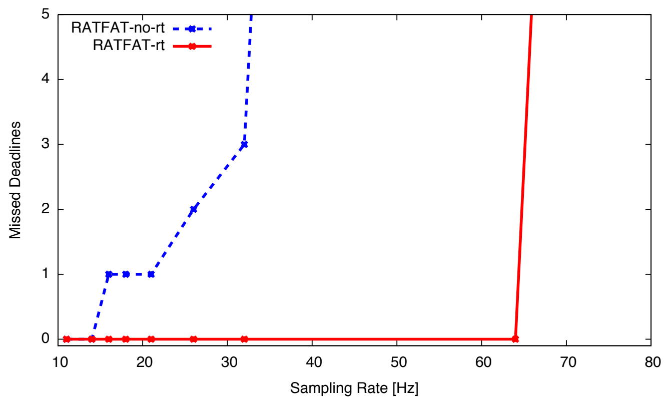


Figure 7. Number of missed real-time deadline for RATFAT with and without real-time management

write request. We vary the sampling interval $T$. The process runs for 2500 sample periods, writing a total of 40,000 kiB of data to a file. Figure 7 shows how many deadlines the sampling process is missing using either the blocking file system API or RATFAT's concurrent API. It can be seen that for sampling rates higher than 16 Hz ($T \approx$ 62ms), the blocking API leads to an increase in missed deadlines. In contrast the concurrent API, queueing request to the *FSP* process, allows the sample process to meet all deadlines up to a sampling interval of 64 Hz ($T \approx$ 16ms). This represents a 4-fold increase for the safe real-time operating area. With $T$ below 16 ms, the interval is too short compared to the duration of the SPI transfers to the microSD card for RATFAT to guarantee the deadlines. In such a case, software tricks will not help anymore, instead the application demands more powerful hardware.

### C. Memory Footprint

To give an impression about RATFAT resource usage, we measured RAM and ROM footprint on the INGA node. RATFAT does not use any dynamic RAM, so it can also be used on platforms where dynamic RAM allocation is not available. Table I shows RATFAT's and COFFEE's memory footprint in their respective default configurations. For RATFAT, we also show the resource usage without the real-time management.

RATFAT uses significantly more RAM than COFFEE. This is mainly due to the sector cache, which is also responsible for RATFAT's high throughput. The real-time variant uses even more RAM to manage the queue and additionally write buffers. When the real-time features are enabled RATFAT's codesize is almost twice as large as COFFEE's. For nodes such as INGA, whose processor includes a relatively large onboard flash of 128 kiB this should not be an issue, however for more constrained devices such as the older T-Mote Sky platform sporting only 48 kiB of flash this might prevent the usage of RATFAT.

## V. Conclusions

We presented RATFAT, an implementation of the FAT file system for WSN nodes. In combination with a node that supports cheap SD cards using the FAT file system eases data exchange between sensor nodes and PCs, as FAT is supported by virtually all PC operating systems. As RATFAT also supports the Contiki CFS API it can be used as a drop-in replacement for applications that have been using COFFEE or other CFS compliant file systems so far. RATFAT is available as open source in the public INGA Git[6].

In contrast to existing FAT implementations RATFAT includes a real-time management component, that can split file system requests into smaller operations that are scheduled based on the maximum allotted time for the file system. Using RATFAT, real-time constraints for a process can be fulfilled even when using a cooperative multitasking system like Contiki.

In the evaluation we have shown that the real-time management allows to implement systems with higher polling rates for real-time processes as longer running file system operations are broken down into shorter atomic operations. In fact on the evaluated INGA node, real-time poll rates of up to 64 Hz can be used. This leaves 15.6 ms between two process invocations for atomic file system operations which take up to 12 ms. Therefore, applications demanding even higher polling rates would need more powerful WSN hardware.

Table I
RATFAT MEMORY FOOTPRINT

| | COFFEE | RATFAT-non-RT | RATFAT-RT |
|---|---|---|---|
| ROM (.text) | 10748 | 14772 | 19786 |
| RAM (.data & .bss) | 233 | 907 | 1556 |

## References

[1] F. Büsching, U. Kulau, and L. Wolf. Architecture and evaluation of INGA an inexpensive node for general applications. In *2012 IEEE Sensors*, pages 1–4. IEEE, Oct. 2012.

[2] M. Marschollek, K.-H. Wolf, M. Gietzelt, G. Nemitz, H. Meyer zu Schwabedissen, and R. Haux. Assessing elderly persons' fall risk using spectral analysis on accelerometric data - a clinical evaluation study. In *Engineering in Medicine and Biology Society, 2008. EMBS 2008. 30th Annual International Conference of the IEEE*, pages 3682–3685, Jan. 2008.

[3] T. O'Donovan, J. Brown, F. Büsching, A. Cardoso, J. Cecelio, J. do O, P. Furtado, P. Gil, A. Jugel, W.-B. Pöttner, U. Roedig, J. sa Silva, R. Silva, C. Sreenan, V. Vassiliou, L. W. T. Voig and, and Z. Zinonos. The GINSENG System for Wireless Monitoring and Control: Design and Deployment Experiences. *ACM Transactions on Sensor Networks (TOSN)*, 10(1), February 2014. accepted for publication.

[4] T. O'Donovan, C. J. Sreenan, N. Tsiftes, Z. He, and T. Voigt. Poster abstract: Storage-centric debugging of performance problems in sensor networks. *7th European Conference on Wireless Sensor Networks, 17-19 Feb 2010, Coimbra, Portugal.*, 2010.

[5] S. Schildt, W.-B. Pottner, and L. Wolf. Contiki Ring File System for Real-Time Applications. In *2012 IEEE 8th International Conference on Distributed Computing in Sensor Systems*, pages 364–371. IEEE, May 2012.

[6] R. C. Shah, S. Roy, S. Jain, and W. Brunette. Data MULEs: modeling and analysis of a three-tier architecture for sparse sensor networks. *Ad Hoc Networks*, 1(2-3):215–233, Sept. 2003.

[7] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. Enabling Large-Scale Storage in Sensor Networks with the Coffee File System. In *Proceedings of the 8th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN 2009)*, San Francisco, USA, Apr. 2009.

[6]http://git.ibr.cs.tu-bs.de/?p=project-cm-2012-inga-contiki.git