CORA Cork Open Research Archive
Cartlann Taighde Oscailte Chorcaí

Title	CacheL: A cache algorithm using leases for node data in the Internet of Things (Best Paper Award)
Authors	Tracey, David;Sreenan, Cormac J.
Publication date	2016-08
Original Citation	Tracey, D. and Sreenan, C. (2016) 'CacheL - A Cache Algorithm Using Leases for Node Data in the Internet of Things', IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud), Vienna, Austria 22-24 August. doi: 10.1109/FiCloud.2016.9
Type of publication	Conference item
Link to publisher's version	https://ieeexplore.ieee.org/document/7575837 - 10.1109/FiCloud.2016.9
Rights	© 2016 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works.
Download date	2024-03-29 06:41:22
Item downloaded from	https://hdl.handle.net/10468/9652

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

CacheL - A Cache Algorithm using Leases for Node Data in the Internet of Things

David Tracey
Dept. Of Computer Science,
University College Cork,
Cork, Ireland.

Cormac Sreenan
Dept. Of Computer Science,
University College Cork,
Cork, Ireland,

***Abstract*—Wireless Sensor Networks (WSNs) allow applications to interact with the physical world using sensing nodes deployed in an Internet of Things (IoT). Many WSN sensing nodes have constrained computing and memory capabilities. This paper details a new cache algorithm suitable for use on constrained nodes and its use in an architecture incorporating caching and the flow of data from sensors to services, possibly Cloud-based. This cache algorithm is influenced by the Clock paging algorithm and manages the leases of cached data in its replacement policy, removing the need for a separate process for this. This paper presents implementations of the algorithm in C on the Contiki OS and Java, compares its performance to LRU and considers its suitability for use on constrained WSN nodes.**

Keywords—Wireless Sensor Networks, Contiki, Cache, Paging Algorithms, Lease, LRU

I. INTRODUCTION

Cheaper processing power and the use of IP is allowing more data to be gathered, stored and analysed in Wireless Sensor Networks (WSNs). Micro IP stacks and IPv6 over low power wireless (6LowPAN) [1] allow nodes to form an "Internet of Things" integrating the physical world with the Internet [2] in a distributed system of devices and applications comprising sensing, computation, actuation and analysis. WSN nodes have constrained processing power, memory and energy consumption, but their wireless capability allows nodes to be deployed close to the sensed phenomenon. Nodes may forward data to remote data centres or collect data from other nodes. Nodes may perform actions based on events or external input or possibly aggregate the sensor data, as aggregation and analysis may be required close to the source, e.g. a sensor testing for hazardous gases must react to major events based on timely local analysis of its own and neighbouring sensor readings (current and for a past period) and also forward data to a centralised, probably Cloud, system for longer term storage and more detailed analysis.

A key requirement in the design of WSNs is to reduce the number of messages sent (often over many hops) in order to extend battery life, while still being able to provide data at low latency. This is often addressed by managing node duty cycles, but another approach to reducing transmission and supporting local data analysis is to store data on the nodes. The limited available storage on nodes and the fact that local analysis of node data is often most useful for a recent time period suggests that caching is appropriate in WSNs. Furthermore, returning cached data from a node closer to the requesting node than the source node will reduce the number of transmissions required, also reducing interference effects. It will also reduce the response time, particularly when requests from multiple nodes are answered using the same cached data. Caching on WSN nodes is, however, problematic due to their limited memory and processing power, and the lack of a system architecture that easily incorporates cached data.

This paper presents our contribution of the CacheL algorithm and an implementation on constrained nodes using the Contiki OS. Its novelty lies in its intrinsic use of leases in a cache replacement policy for WSN data inspired by the Clock algorithm, so not requiring the use of sort (like LRU) and not requiring additional communication between WSN nodes, as in several cooperative caches. It incorporates lease management into the replacement process to remove the need for a separate periodic process to manage leases. We consider that the use of a lease for cached data is important in WSNs as it provides self-management, i.e. cached data for nodes that have left the network will be removed from the cache on lease expiry. A lease can also represent the time sensitive nature of node data, e.g. set/renew the lease based on the time of the next sensor reading. It also allows recent data to be retained as it is more likely to be of interest.

The remainder of this paper is organised as follows. We present prior work in section II and the CacheL Algorithm in Section III. Section IV presents a prototype implementation of the CacheL Algorithm and its integration into our architecture [3]. Section V presents performance results using the Yahoo Cloud Server Benchmark (YCSB) [4]. The paper concludes in Section VI.

II. EXISTING CACHE ALGORITHMS

The use of caching in WSNs has the potential to reduce energy use across nodes by reducing transmissions, to support local data analysis and to lower latency. A cache replacement policy aims to maximize the use of resources, e.g. memory or network bandwidth and can be based on [5]:

1. Application provided future access hints, e.g. based on a query to be performed.

2. Explicit detection of access patterns unfriendly to LRU and a switch to other replacement strategies.
3. Tracing and history of accesses (only useful if they reflect future use).

Least Recently Used (LRU) replaces the item which has not been accessed for the longest time from a list ordered by last access time. It is simple, but has the cost of maintaining order. It suits workloads showing locality, where an item is accessed shortly after a previous access. Most Recently Used (MRU) is useful when older items are more likely to be accessed. LRU does not distinguish a recently added, never accessed, entry from one accessed frequently but not recently, so a scan accessing a series of items once only flushes pages which may be accessed again. LRU/k [6] improves this by prioritising items based on their kth most recent access, but is log(n) to manage a priority queue. 2Q [7] provides constant overhead per access as in LRU with a similar page replacement performance to LRU/k, by using an LRU main queue and a FIFO queue for "hot" items. MQ [8] uses LRU queues based on page frequency. The page frequency is incremented on a hit and that item becomes the MRU in that queue. On an access, a constant (expireTime) of the LRU item per queue is checked and if expired that items becomes the next queue's MRU.

Least Frequently Used (LFU) removes items with the lowest count first using a linked list for O(n) removal, insertion, but it allows a previously frequently accessed item to remain cached and not be replaced by more recent items. LFU-Aging includes the recency of last access, while LFU with Dynamic Aging (LFUDA) adds a cache age factor (less than or equal to minimum value in the cache) to the reference count [9]. Greedy-Dual Size (GDS) considers the item's size and a cost function associated with fetch.

A TimeToLive (TTL) is often used where weak consistency is acceptable. Redis 2.x [10] actively removes timed-out keys on access and it also periodically tests a number of random keys and deletes the expired ones.

The CLOCK algorithm [11] uses a circular list of fixed sized pages. A "clock" hand points to the oldest page in the circular list. On a page fault, the reference bit (set on page access) of the page at the clock hand is checked. If it is not set, the page is replaced by the faulting page, otherwise it is reset and the hand moves through the list clearing page reference bits until it finds one not set and replaces that page. This algorithm approximates LRU and avoids its ordering of the list, but shares its lack of scan resistance.

The WSClock Algorithm [12] uses the task's virtual time and a page's last reference time to determine if the page should be replaced. GCLOCK [13] increments a page's counter on a hit and the clock hand sweeps the pages decrementing the counters until it finds and replaces a page with a zero count. CLOCK-Pro [14] keeps track of a limited number of replaced pages to overcome the LRU problems with scan and loop, using the LIRS (Low Inter-reference Recency Set) [15] policy of replacing a page with a high reuse distance even if recent. A single list of pages is ordered with small recencies at the head and large ones at the tail. Cold pages stay in the list for a test period (set to the largest recency of the hot pages, becoming hot if accessed in that period. Three hands sweep the list: pointing to the last hot page, the last cold page and the last cold page in a test period. Clock with Adaptive Replacement (CAR) [16] is self-tuning and uses two clock lists: T1 for pages with "recency" and T2 for pages with "frequency". New pages are added to T1 and move to T2 based on a test of long-term utility or frequency. A list with history of recently evicted pages from T1 and T2 is used to adaptively set the list sizes.

ARC uses two variably-sized lists (combined size of twice the number of pages) to hold the history access information for referenced pages [17]. One list holds cold pages (touched once recently) and the other holds hot pages (touched at least twice recently). The memory for each list is managed based on which list had the most recent misses using a ratio of cold/hot page accesses. It does not handle the locality of pages in the two lists, so a page that is regularly accessed with a reuse distance a little more than the memory size may get no hits.

In general terms, LFU and MQ are more expensive than LRU, while LIRS and ARC have a cost similar to LRU. CAR has a cost close to CLOCK, but with similar performance to ARC and better scan resistance than LRU.

III. A Cache Algorithm For WSN Nodes

A. *Cache Approaches in WSNs*

Caching has been investigated for WSNs in the context of co-operative caching to serve data with low latency and reduce energy consumption. Each node constructs responses to queries by cooperating with its neighbours. A key aspect is identifying which nodes will implement the co-operative caching decisions, e.g. which node makes forwarding decisions or which nodes get the requests for data [18], with little focus on the cache replacement algorithm itself. Some approaches calculate a Node Importance Index, requiring nodes to hold their neighbours' connectivity state and lack robustness [19]. Data replication and caching strategies have been considered in Mobile Ad-hoc Networks (MANETs) [20], but some schemes [21] require knowledge of network topology and involve periodically moving data, both of which reduce their effectiveness in a WSN, especially if data access patterns vary or nodes join/leave. Static approaches to cache placement [22] are similarly limited in the WSN scenario. COOP [23] keeps a table of previous requests and the nearest relevant cache, using flooding to find the data only in the case of a miss. The hybrid cache for cooperative caching in MANETs [24] does not require the selection of special nodes and shares data only on the path between source and requester. When forwarding data, a node may cache either the data or the path according to the data size, TTL and number of hops to be saved.

Directed Diffusion [25] is a data-centric approach, where node data is named by attribute-value pairs. A node sends interests with a duration to request named data and the concept of a gradient (a value and direction) moves data according to these interests. Intermediate nodes may cache recently sent data messages or aggregate data, although the emphasis is on the routing and filtering of data and the matching of interests rather than cache implementations.

The Constrained Application Protocol (CoAP) [26] uses Internet approaches, e.g. the RESTFul architectural style [27]. It is designed to be easy to proxy to/from HTTP and uses a small, simple header of less than 10 bytes and a UDP binding with reliability and multicast support. CoAP supports a simple cache in an endpoint or an intermediary, using freshness and validity information in the CoAP responses. The cache allows an earlier response message or a stored response for the current request. The origin server provides an expiration time using the Max-AgeOption and an ETag Option in the GET request allows an origin server select a stored response to use and to update its freshness.

B. System Model

We assume a WSN that consists of sensor nodes using bi-directional links in a multi-hop manner. No assumptions are made about the size of the network. We require that the cache algorithm imposes little communication overhead, e.g. does not require flooding and can handle the dynamic nature of the WSN in terms of the source and destination of requests. We do not assume a static topology or require data relocation or recalculation to update topology related data when nodes join/leave. We do not assume that all nodes have the same capability; some nodes will be able to cache data and other less capable nodes will only act as sources (or sinks) or forwarders of data. Data will be retrieved from the source node in the absence of a cache. We do not make any assumptions about the routing algorithm used and assume only that data may be cached as it is forwarded, preferably peer to peer. While such forwarding may be as a response to a request, data may also be pushed from nodes. Unlike the cooperative caching approaches above, no cache management protocol across nodes is required.

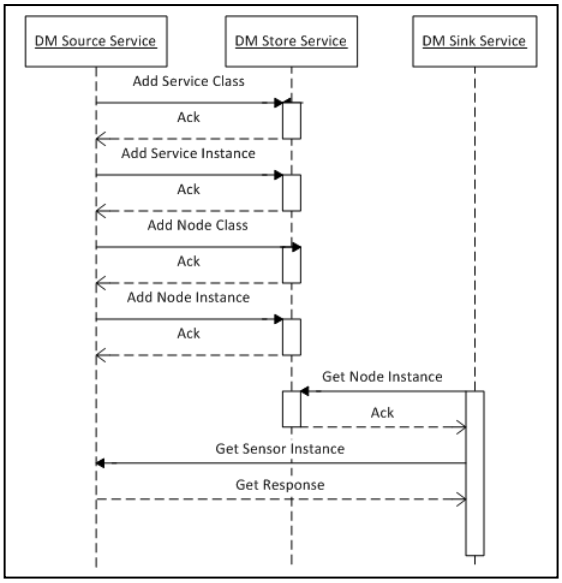


Fig. 1. Example of HPP Data Model Service Layer Interaction

This system model is supported by the Holistic Peer to Peer (HPP) Architecture [3] which has a Data Model (DM) Service layer to represent nodes and services (on node or Cloud) using defined roles based on capabilities. A node uses a data store modelled as a tuple space, with a simple protocol for the data flow between sensor and service(s). It includes commands for nodes and services to add/remove instances to/from the tuple space for their capabilities, interests and data, including setting leases on the data. Figure 1 shows a DM_SOURCE_SRV adding its service and node classes (or templates) and instances to a DM_STORE_SRV (holding local and peer data) on a node that is able to cache data. A DM_SINK_SRV queries this DM_STORE_SRV for its capabilities and then its node data, which may be returned from the DM_SOURCE_SRV or an intermediate DM_STORE_SRV (if cached there). HPP also includes roles for DM_FORWARDER_SRV (forwards to peers), DM_MATCHER_SRV (for advanced matching queries) and DM_AGGREGATOR_SRV (aggregates peer data).

C. Rationale for the CacheL Algorithm

The successful use of a cache enables reduced communication and so extends the battery life of WSN nodes, but also reduces interference effects. It should also reduce the response time to queries. We consider the key requirements for a cache on a WSN node to be:

1. Simple to implement and efficient in CPU and memory use.
2. Use of a single abstraction for data storage. There should not be separate stores for the node's own data, remote node data and lease management.
3. A lease associated with cache data. The lease is set and renewed by the source node as the data may only be useful for a limited period, e.g. only the latest copy needs to be held by a node. Permanently stored data will have an infinite lease.

The use of leases provides a self-managing means to handle WSN failures, e.g. if a node fails then leases on its data will not be renewed so the data will be removed from the cache on expiry, freeing valuable storage space.

Our contribution regarding leases is a low overhead replacement algorithm that performs both lease and cache management. This removes the need for a separate periodic lease management process to expire items. e.g. using time buckets to hold items according to lease times and updating these when the bucket period has passed. CacheL's commonality of code and metadata for cache and lease management reduces code size and memory use, as well as making the code simpler. Furthermore, allowing the periodic update of leases and expiry of items fits well with the WSN use case of running only when the node is awake.

A source node requests a lease, but the lease is granted by the caching node and it can use the size of its cache, the item's size and the lease requested to determine the lease granted, e.g. it may not cache items above a certain size. The lease is granted in milliseconds and is not tied to the actual time on the source or caching node. The lease is decremented based on time differences using the time on a node and is renewed by the source node, based on when it received the granted lease response. Hence, there is no need for a global time across the WSN to support the cache lease.

D. The CacheL Algorithm

We propose CacheL for resource constrained nodes to manage items in a node's datastore as a cache. CacheL extends the Clock algorithm by adding a lease to each item cached. The Clock algorithm was chosen as a base, because of its simplicity compared to the cache algorithms outlined above and the ease of extending it. CacheL combines application hints (leases set by the source node give a hint when the item can be removed) and history based on access. The lease has priority over access count in determining item replacement, because the source expects an item to be cached for the period granted and treating access count equally would lead to removing entries with long leases. A lease can be renewed by the data source, so that it is not just a TimeToLive on the data. Unlike paging algorithms, it does not assume a fixed cache size with fixed size buffers and it updates item metadata instead of setting the reference bit.

CacheL does not keep lists sorted by access count or time, but replaces the first one or more items it finds with expired leases or which have not been accessed a set number of times (similar to CLOCK-PRO's use of access counts). It manages leases by iterating through items on a specific event, such as adding an item or a time epoch passing. CacheL can also handle items without a lease and evicts them simply based on access count, similarly to the CLOCK algorithm. Items that are accessed frequently, but whose leases have expired can be handled according to a policy, i.e. whether to always remove on expiry or to retain them in the cache if accessed.

1) Algorithm Operation

The algorithm holds the entries to be managed in a list, which does not have to be held in any specific order (unlike LRU) and has at least one other list/queue, a pending queue, of item references whose lease is due to expire and to check the access count, i.e. not just based on count like 2Q. This also has similarities to MQ, but is simpler and done per access as with MQ, but also on a periodic basis. The algorithm sweeps through a main queue until it has found enough items to evict (using count or memory size). This sweep is done when an attempt is made to add to the cache or using the fronthand and backhand time periods, where the fronthand is less than the backhand.

The backhand sweep is primarily for lease management, but also moves items according to count. This sweep iterates through all the main queue or until it finds one or a specified number of items to remove. It puts items on the pending queue (or a count queue) based on their remaining lease or deletes them if they were not accessed since the previous sweep. This sweep through the main queue decrements the access count, but does not clear it to give some precedence to items with multiple accesses. The previous accessed time is not stored as that is implicitly handled by the periodic sweeps. It holds a reference to the last item checked on a sweep to start the next sweep.

A sweep at the fronthand frequency iterates through the pending queue. Items on this queue are due to expire soon, so this sweep removes them, unless they have been accessed or had their lease renewed, and decrements the access count (effectively setting the time for an item to be accessed before it will be deleted). This pending queue sweep will be called on each add and removes the specified number of entries with expired leases. Items are added to the tail of this queue, but it is searched from the head increasing the time an item can remain on the queue if items expire ahead of it. Items are not held in order of their remaining lease or recency of access as with LRU. The pending queue means that items with short or near expiry leases will be removed first, which is reasonable as the source node setting a lease should know how long this data should remain. Items may stay in the cache longer than the lease, however, as the lists are only checked on a sweep to save processing overhead.

A lease threshold is used to select items to put onto the pending queue and by default is set to the time before the next backhand sweep - increasing it may remove items that still have some lease remaining. The period between fronthand sweeps and a backhand sweep could be tuned using the size of the pending queue, as the time between being put on the pending queue on a backhand and removal on the next fronthand allows for lease renewal (or access).

The number of items (or size of memory) to free can be set rather than just inserting the new page at the selected location on a page fault. It returns the number of entries removed, so the caller can decide whether to allow a new item to be added or to explicitly delete an item, rather than wait for leases to expire. Deleting an item may occur outside cache management, so its reference will either be removed immediately from the pending queue(s) or lazily rely on the lease to expire as it will not be renewed. Similarly, items are either immediately moved from the pending queue on lease renewal (or on access) or lazily on the next sweep.

Simplified pseudo-code for the sweep methods follows. Other code stores the times of the last fronthand and backhand (since_fhand, since_bhand) to determine which sweep to run, even if the node had been asleep.

```
doSweep() {
 if (since_fhand) //based on time period or counts
   if pending Queue not empty
      deleted = pendQSweep(toDelete)
 // Optimisation added for lots of long leases
 if (min_lease – lease_threshold > since backhand)
    return
 if (since_bhand || // based on time period or counts
    deleted < toDelete))
    deleted = mainQSweep(toDelete)
}
pendQSweep() {
 for each item in pendQ
  decrement access_cnt
  if lease set {
     if (lease expired)
        remove
     else if lease > lease_threshold // was renewed
        move to mainQ behind hand pointer
        // has full sweep to be accessed.
     else if (lastAccessTime == 0)
        if access_cnt <= 0,
           remove // not accessed while on pending queue
```

```
      else // (access_cnt > 0 so was accessed)
         leave in pendQ // allows lease renewal
   } // end of lease case
 }
mainQSweep() {
 iterate mainQ starting from last_checked item
 last_checked = this item
 decrement access_cnt
 if (lease set) {
    decrement lease
   adjustLeaseByCnt
   if (lease expired)
     if (access_cnt == 0)
        remove
     else
        move to pendQ
     else if lease < threshold
        move to pendQ // else stays on mainQ
    update min_lease if lease < min_lease
    return
  } // end of lease set case
  if access_cnt <= 0
     remove
   else if access_cnt == 1
     move to pendQ          // or a separate cntQ
   else // if access_cnt > 1
     leave on mainQ, // gives it more precedence than Clock
 }
```

Unlike LRU, it is valid for CacheL to not cache an item if no leases have expired, so the mainQSweep() could move through the entire cache, giving an O(N) worst case. We reduce this cost by holding the minimum lease of an item in the main queue and not doing a sweep if that lease has not expired. Also unlike LRU, a sweep is done on an add or a time period rather updating ordered lists per access.

Large, infrequently accessed objects with a long lease may remain cached ahead of those accessed more often with short leases. That should be handled by not granting a long lease to large objects when they are added and by increasing the lease based on count in adjustLeaseByCnt().

The algorithm acts like CLOCK (but with a pending queue) to handle items without a lease; the mainQSweep sets last_checked and decrements access count, while the pendQSweep removes items if not accessed. Indeed, a separate count-based list could be implemented for items without leases or the sweep approach could be easily extended to check other parameters, e.g. the number of hops data has taken. In the no lease case, the algorithm resists scan patterns once the cache is populated, as it adds to the end of the pending queue and the main queue is not sorted.

IV. Implementation

This section discusses prototype 'C' and Java implementations of CacheL. The implementations included specific metadata in the cached items to avoid having to update all entries in the cache on every sweep, e.g. the last time it was visited on a sweep and its access count. For updating access count, the number of times it missed being decremented (due to the periodic nature of sweeps) is handled by holding a count which is incremented on each sweep and using that to decrement the count on next sweep.

A. 'C' Implementation

The 'C' prototype was implemented initially on Linux and ported to the popular Contiki OS using the erbium REST implementation [28] on a Sky WSN node using an MSP-430 Microprocessor with 10K RAM and 48K Flash. This node was emulated in Contiki's Cooja simulation environment. CoAP "resources" were created which integrated the cache into our HPP architecture, e.g. a DM_SOURCE_SRV was created and key value pair objects were sent to a DM_STORE_SRV node.

B. Java Implementation

CacheL was also implemented in Java so that it could be compared to a Java LRU implementation using the Yahoo Cloud Server Benchmark (YCSB) [4]. The Java LRU implementation used HashMap's removeEldestEntry() method. The CacheL implementation also used a HashMap to be comparable, although this needed extra code for ConcurrentModificationExceptions to mark items for deletion in a sweep and to remove them later. It also had to hold a separate list of index-object reference pairs so a sweep could continue from the previous position (the C implementation did not need as it used its own circular list).

C. YCSB Test Environment Implementation

While not representative of real WSN scenarios, the YCSB tests compare the effectiveness of CacheL to LRU. YCSB testing was performed on a single dual core PC with 8GB RAM, using the default YCSB data size of 1000, 1 KB records and 1000 operations. YCSB uses the configurable workload options shown in Table I, where WA indicates Workload A. A workload with a Uniform distribution (WB-Uni) was added to represent periodically reporting sensors.

Load	Ratio of Operations	Dist	Nature	Example
WA	Read/update : 50/50	Zipf	Update Heavy	Session store
WB	Read/update : 95/5	Zipf	Read Heavy	Photo tags
WC	Read/update : 100/0	Zipf	Read Only	User Profile
WD	Read/update/insert: 95/0/5	Latest	Read Latest	Status updates
WE	Scan/insert: 95/5	Zipf	Short Ranges	Posts in thread
WF	Read/read-modify-write: 50/50	Zipf	Read-Modify-Write	User Database

TABLE I. YCSB Workloads

YCSB uses separate commands to load a database and then a run phase test on that data, but these had to be combined to test an in-memory cache. The update implementation was not a simple insert as with databases, as the item to be updated may not be cached and so had to be read and inserted. The tests were run for LRU and CacheL (with and without leases) over a range of cache sizes with CacheL doing fronthand/backhand

based on the number of calls to put() and not the time since last sweep (which suits CacheL better and is likely in a WSN scenario).

V. EVALUATION

This section considers CacheL in terms of hit/miss ratio compared to LRU, the effectiveness of a lease and its suitability for implementation on a WSN node. It also outlines optimisations made based on the values of counts added at key points in the code (shown in Figure 2).

A. Implementation Complexity

The pseudo-code extract shows CacheL to be straightforward to implement, including its use of our object space and data model service abstractions. The C implementation was about 150 lines of C code and used about 1KB of memory with a supporting object abstraction using a circular list library. CPU use was not an issue during the test runs. The CacheL implementation was about 200 lines of Java whereas the LRU code was 30. This is reasonable as CacheL includes code for lease handling and interfacing to the object store, as well as the extra code due to the use of HashMap. The YCSB driver was modelled as a DM_STORE_SRV cache with our object API methods of new() read(), remove() and put() integrating easily with the YCSB API methods of init(), read(), delete(), update(), insert().

B. Optimisations to CacheL

Testing resulted in several optimisations to make better use of access counts and reduce unnecessary sweeps. Firstly, when a sweep does not find an expired lease, a reference to the item with the minimum lease is held to avoid unnecessary backhand sweeps looking for expired leases. This improved the latencies relative to LRU, especially for lease range 0-1000ms, e.g the backhand sweep was skipped over 900 times for WorkloadA at all sizes and for other workloads according to cache size, e.g. 944, 795 for Workload D (Uniform) for sizes 100, 250. The second optimisation was adjustLeaseByCnt(), which increased the lease by the backhand period for items with an access count above zero. This extends the time frequently accessed items stay in the cache while still giving lease priority and allowing smaller leases to be granted. This worked for small leases, but had little effect for long leases, e.g. cache size 100 and workloadb-uniform had 892 failed puts out of 1000 as no cached item had expired. This optimisation is expected to be useful in longer-lived tests. The third optimisation was adding another sweep after the backhand if it did not delete the required number of items, as the first sweep may have moved items to the pendingQ or decremented access counts. The fourth one was to hold the last_lease_check time at the end of the main sweep to reduce system time-related calls.

C. Performance Comparison of LRU and CacheL

1) LRU vs CacheL without leases

CacheL works like CLOCK in this case. Table II shows the hit ratios were comparable for LRU and CacheL, with LRU slightly higher on Workload B for Uniform distribution, but CacheL was equal or better on Zipf tests. Changing the time or number of puts (default 3) between fronthand and backhand sweeps did not affect the hit ratio, due to the short duration of tests.

Cache Size	**100**		**250**		**500**		**750**		**1000**	
	lru	ChL	lru	ChL	lru	ChL	lru	ChL	lru	ChL
WA-Zipf	56.4	55.6	58.4	59.9	59.2	59.5	61.7	61.8	63.9	62.6
WB-Uni	15.6	12.9	29.5	29.1	52.6	48.7	73.5	77.0	100	100
WB-Zipf	19.5	23.1	32.0	30.8	56.7	56.6	78.7	78.8	100	100
WC-Uni	10.2	11.0	28.9	24.9	52.7	49.5	72.3	77.3	100	100
WC-Zipf	9.4	19.2	23.8	26.1	47.8	50.6	71.9	73.0	100	100
WD-Lat	69.7	62.4	83	82.2	91.5	91.8	95.6	96.7	99.4	99.9
WD-Uni	10.7	11.5	27.5	27.3	50.3	50.7	62.3	74.9	96.7	98.2
WF-Uni	9.0	9.0	28.1	25.2	51.4	52.7	63.3	73.0	100	100
WF-Zipf	19.1	20.4	35.9	36.1	44.7	58.2	68.1	79.1	100	100

TABLE II. HIT RATIO PER WORKLOAD FOR CACHE SIZES (100 TO 1000) FOR LRU AND CACHEL (WITH NO LEASES)

2) LRU vs CacheL with leases

These tests use leases uniformly distributed over a time range of 0-100ms or 0-1000ms, which is smaller than expected in a real WSN, but was used as the tests completed in approximately 300 ms. These ranges show how effectively CacheL manages leases, although not taking advantage of CacheL being able to use a 'hint' provided by the application/sensor. Comparing the results in Table II to Table III, the use of CacheL with the 0-100ms lease distribution generally has a hit ratio higher than, or comparable to, LRU for cache size 100 and is comparable at other cache sizes, although there is a notable reduction for the Workload D-latest distribution (WD-Lat). Table III shows a hit ratio reduction for CacheL in all 100 sized caches when using the 1000ms compared to 100ms lease range as residency in the cache is dominated by the priority given to lease. It also suggests the granting of leases should be managed actively to ensure appropriate expiry, e.g. based on the leases remaining in the cache.

Cache Size	**100**		**250**		**500**		**750**		**1000**	
Lease Range	100	1000	100	1000	100	1000	100	1000	100	1000
WB-Uni	20.0	13.6	28.1	27.5	52.9	54.0	75.4	75.7	99.9	99.8
WB-Zipf	16.3	13.4	27.5	29.5	51.2	54.0	74.3	75.5	100	100
WC-Uni	17.6	11.3	25.5	25.0	50.7	52.3	76.7	74.8	100	100
WC-Zipf	15.7	9.8	22.6	26.7	48.7	48.5	73.1	75.3	100	100
WD-Lat	48.6	5.6	77.4	9.9	90.7	54.2	97.4	22.2	99.7	57.0
WD-Uni	28.3	13.7	25.8	30.0	51.0	51.0	73.6	76.7	96.4	99.7
WF-Uni	21.3	10.0	26.1	24.1	49.4	50.6	73.9	75.3	100	99.9
WF-Zipf	21.5	10.0	23.1	26.3	48.5	52.0	73.5	75.6	100	100

TABLE III. HIT RATIOS FOR CACHEL (WITH UNIFORMLY DISTRIBUTED LEASES 0-100MS AND 0-1000MS)

D. *Performance Characteristics of CacheL*

Figure 2 shows the values of the counts added to the code for cache size 100 and a lease distribution of 0-100ms, where CacheL was most effective, i.e., expiring data quickly in a small cache. fhDeleteCount and bhDeleteCount are the number of deletes done on a fronthand or backhand sweep as leases expire, with fronthand having an effect on WA-ZIPF, WD-Uni and WF-ZIPF (although fhDeleteCount drops to less than 10 for other cache sizes). The maximum number of entries on the pending queue (pendQMax) and fhDeleteCount show it used for cache size 100 and Workload A. SweepDeleteCount counts the number of items deleted after a backhand sweep and shows the impact of the third optimisation on bhDeleteCount. bh_skipped counts the backhand sweeps not run based on the minimum lease in the cache, showing the value of the first optimisation above.

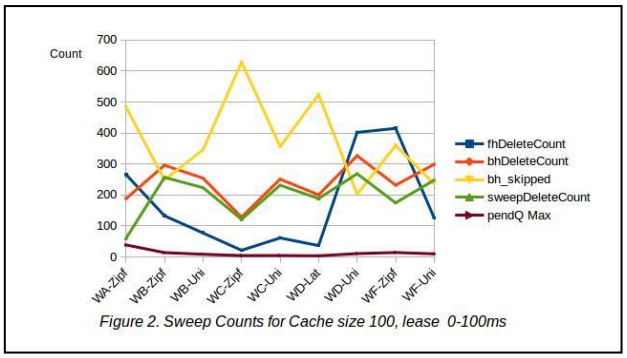


Figure 2. Sweep Counts for Cache size 100, lease 0-100ms

Other counters showed that long leases limited the value of both access count in cache replacement and the fronthand sweep, e.g. the 0-1000ms distribution had few leases near expiry to populate the pending queue.

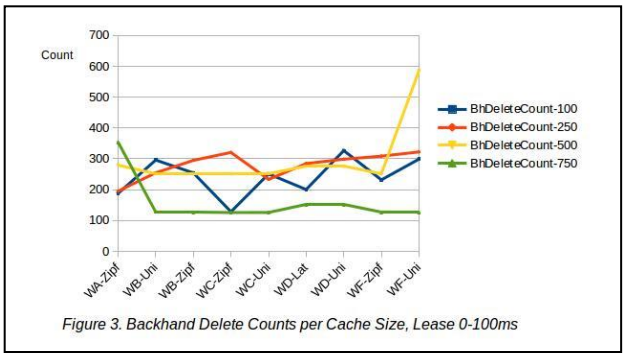


Figure 3. Backhand Delete Counts per Cache Size, Lease 0-100ms

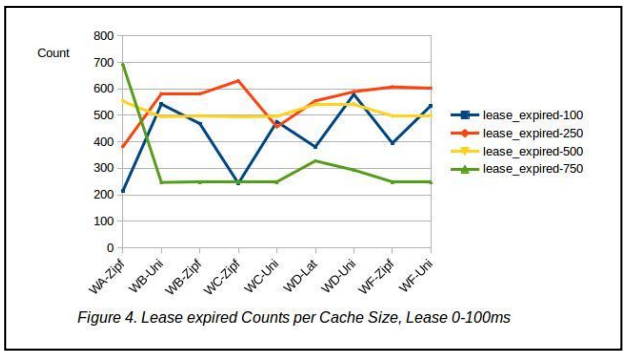


Figure 4. Lease expired Counts per Cache Size, Lease 0-100ms

Figures 3 and 4 show the bhDeleteCount and lease_expired counters across cache sizes using the 0-100ms lease distribution. The lease_expired count is incremented when deleting an item in the backhand sweep if the lease and access count are both less than or equal to 0. The lease_expired counter in Figure 4 and bhDeleteCount in Figure 3 show that the larger cache sizes are dominated by lease expiry in the backhand sweep. There is more variation at cache sizes 100 and 250 where the lease and access counts are used. Figure 5 details this for size 100.

In Figure 5, min_lease counts the number of times the item with the minimum lease was deleted in the backhand sweep; lease_thrshld counts items deleted by a fronthand sweep; lease_pending_expiry counts items moved to the pending queue by a backhand sweep; lease_expiry_removed counts the items removed from the pending queue by a fronthand sweep. For cache size 100 the lease_expired counter in the backhand sweep is the main way of expiring leases, but lease_expiry_removed indicates items were moved to the pendingQ and lease_threshold shows they were then removed by a fronthand sweep.

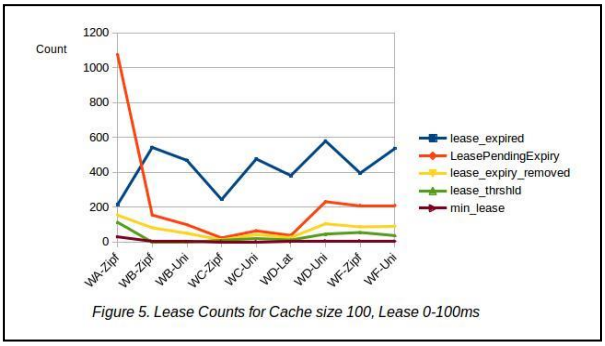


Figure 5. Lease Counts for Cache size 100, Lease 0-100ms

In summary, it can be seen that CacheL is comparable to LRU and when using short leases it expires data quickly in a small cache, as required on WSN nodes. It can also be seen that the algorithm manages leases effectively and could take advantage of a lease provided by source nodes as a hint.

VI. CONCLUSION

We have outlined the rationale for the use of a cache algorithm in WSN nodes and the value of an associated lease. We have proposed the CacheL algorithm with its inherent management of leases for cached data. This algorithm has been shown to be flexible enough to handle the limited node memory for cached data and simple enough to implement and run on a constrained node. We have also shown how CacheL fits into our architecture [3].

The results show that the algorithm is comparable to LRU, even without leases, and that it successfully manages the cache using leases and access count. This shows the value of sources using a lease to provide hints for their data and also the importance of managing the granting of leases. This will be investigated in further work.

We have also shown the use of YCSB to test, understand and optimise the performance of the CacheL algorithm. Future work will use a YCSB load generator to run tests against WSN nodes to investigate the effectiveness of CacheL in reducing power consumption and supporting self-management as nodes move or leave the network.

REFERENCES

[1] N. Kushalnagar, "IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs): Overview, Assumptions, Problem Statement, and Goals", RFC 4919

[2] D.A Reed, D.D. Ganno., J.R. Larus., "Imagining the Future: Thoughts on Computing," Computer, vol. 45, no. 1, pp. 25-30, Jan. 2012.

[3] D. Tracey, C. J. Sreenan, "A Holistic Architecture for the Internet of Things, Sensing Services and Big Data", Data-intensive Process Management in Large-Scale Sensor Systems held with CCGrid 2013

[4] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan and R. Sears, "Benchmarking Cloud Serving Systems with YCSB", ACM Symposium on Cloud Computing (SoCC), 2010

[5] D. Lee et al, "On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies", Proceeding of ACMSIGMETRICS, May 1999.

[6] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K Page Replacement Algorithm for Database Disk Buffering", Proceedings of the 1993 ACM SIGMOD Conference, 1993, pp 297-306.

[7] T. Johnson, D. Shasha "2Q: A Low Overhead High-Performance Buffer Management Replacement Algorithm", Proc. VLDB Conf., Morgan Kaufmann, 1994, pp 297-306

[8] Y. Zhou, J.F. Philbin, "The Multi-Queue Replacement Algorithm for Second-Level Buffer Caches," Proc. Usenix Ann. Tech. Conf. (Usenix 2001), Usenix, 2001, pp. 91-104.

[9] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms", USENIX Symposium on Internet Technologies and Systems, Monterey, CA, pp. 193-206, Dec. 1997

[10] www.redis.io

[11] F. J. Corbato, "A Paging Experiment with the Multics System", MIT Project MAC Report MAC-M-384, May, 1968.

[12] W.R. Carr and J.L. Hennessy, "WSClock—A Simple and Effective Algorithm for Virtual Memory Man- agement," Proc. 8th Symp. Operating System Principles, ACM Press, 1981, pp. 87-95.

[13] A. J. Smith, "Sequentiality and Prefetching in Database Systems", ACM Trans. on Database Systems, Vol. 3, No. 3, 1978,pp. 223-247.

[14] S. Jiang, F. Chen, X. Zhang, "CLOCK-Pro: An Effective Improvement of Clock Replacement", Usenix 2005.

[15] S. Jiang, X. Zhang, "LIRS: An Efficient Low InterreferenceRecency Set Replacement Policy to Improve BufferCache Performance", In Proceeding of 2002 ACM SIGMETRICS, June 2002, pp. 31-42

[16] S. Bansal and D.Modha, "CAR: Clock with Adaptive Replacement", Proceedings 3nd USENIX Symposium on File and Storage Technologies, March, 2004.

[17] N. Megiddo, D. Modha, "ARC: a Self-tuning, Low Overhead Replacement Cache", Proceedings of the 2nd USENIX Symposium on File and Storage Technologies, March, 2003

[18] N. Dimokas, D. Katsaros, L. Tassiulas, Y. Manolopoulos, "High performance, low complexity cooperative caching for wireless sensor networks", Wireless Networks, Springer, 2010

[19] N. Dimokas., D. Katsaros, Y Manolopoulos, "Cooperative caching in wireless multimedia sensor networks". ACM Mobile Networks and Applications, 13(3–4), 337–356., 2008

[20] L. Yin, G.Cao, "Supporting Cooperative Caching in Ad Hoc Networks", IEEE Transactions on Mobile Computing, Jan. 2006

[21] T. Hara, S. Madria, "Data replication for improving data accessibility in ad hoc networks". IEEE Transactions on Mobile Computing, 2006

[22] K. S. Prabh, T. F. Abdelzaher, "Energy-conserving data cache placement in sensor networks. ACM Transactions On Sensor Networks, 2005

[23] Y. Du, K.S. Gupta, "COOP: A cooperative caching service in MANETs", Proceedings of ICAS-ICNS (pp. 58–63), 2005

[24] S. Lim,W. C. Lee, G.Cao, C.R. Das, " A novel caching scheme for improving internet-based mobile ad hoc networks performance. Ad Hoc Networks, 2006

[25] C. Intanagonwiwat, R. Govinden, D. Estrin, J. Heidemann, F. Silva, "Directed Diffusion for Wireless Sensor Networking", IEEE/ACM Transactions on Networking, Vol 11, No. 1, February 2003

[26] Z. Shelby, K. Hartke, C. Bormann, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-18

[27] R. Fielding "Architectural Styles and the Design of Network-based Software Architectures", Doctoral dissertation, 2000

[28] M. Kovatsch, S. Duquennoy, A. Dunkels, "A Low Power CoAP for Contiki", IEEE Conference on Mobile Adhoc and Sensor Systems (MASS), 2011