The Cache Mechanism

-for NICTA Open Sensor Web Architecture

Supervisors: Tom Kobialka and Rajkumar Buyya

Students: Huan Xie, Hong Xue

Department of Computer Science and Software Engineering

The University of Melbourne

National ICT Australia (NICTA)

October 2007

Contents

Abstract

Along with sensor network application development, the trend of combining a sensor network with other techniques is growing. NICTA has implemented an Open Sensor Web Architecture (NOSA) [1] that as a combination of Web Services and Grid technology working with sensor networks to overcome the obstacles of connecting and sharing heterogeneous sensor resources. Although the current NOSA middleware can handle multiple queries, its processing ability is limited. The Core services of the NOSA middleware compose of three specifications of Sensor Web Enablement (SWE) [4]: Sensor Collection Service (SCS) [6], Sensor Planning Service (SPS) [7], Web Notification Service (WNS) [8] and Sensor Repository Service (SRS). Among those four services, the SCS has the responsibility to communicate with the sensor networks and it has implemented for handling multiple queries concurrently. The bottleneck of processing simultaneously requests is that the physical sensor network can only process one query at a time so consecutive queries must wait in a queue until the current query processing completes. This project implements a cache mechanism for overcome this limitation of NOSA in such way that the cache system can group and schedule all the query requests, store the results of the historical queries and analyze incoming query requests based on the relationships between current and cached queries. The entire or part of the results of historical queries can be reused to answer current queries. The aim of these enhancements is to reduce the workload of the physical sensor nodes and improve the multiple queries processing ability of the system. In this report we illustrate the cache system architecture and how it integrates with the existing NOSA system. We describe the design issues related to the components of the cache mechanism and we evaluate our contribution with several test cases, demonstrating that the cache mechanism improves the concurrent queries processing speed of the NOSA middleware by 10~20 times faster without reducing the accuracy.

Acknowledgement

We would like to express our appreciation to Tom Kobialka, the supervisor of this project, for supporting our work, sharing his knowledge and experience with us, and guiding us in the right direction. We also wish to thank Rajkumar Buyya, Associate Professor and Leader of Computer Science and Software Engineering; and Director of the Grid Computing and Distributed Systems (GRIDS) Laboratory at the University of Melbourne and Christopher Leckie, Senior Lecturer at the Department of Computer Science and Software Engineering and Program Leader of Network Information Processing Program at NICTA, for giving us suggestions of how to improve the software mechanisms. Furthermore, we would like to thank Egemen Tanin, lecturer of Department of Computer Science and Software Engineering, for providing us with research references and directions.

1 Introduction

Wireless sensor networks provide a way to observe and monitor the physical environment. Sensor networks have wide applicability in industry, military and scientific applications. Along with the development of sensor networks, pervasive sensors are becoming a reality, providing opportunities for new sensor-based services. The NICTA Open Sensor Web Architecture (NOSA) middleware provides such a sensor-based service by integrating the sensor network with Service Oriented Architecture (SOA) and Grid Technique [1]. The SOA allows middleware to discover, describe and invoke stateful Web Services (WSRF) from a heterogeneous software platform using XML and SOAP standards. And with the contribution of Globus Middleware platform, the services could be highly geographically distributed. NOSA takes a major step forward to achieving the vision of a Sensor Grid [2]. Following such integrated architecture, the sensor network as a resource could be discovered, accessed and controlled over the World Wide Web through the Web Services which define the basic data processing operations including data query, retrieval and aggregation, resource scheduling, allocation and discovery. [3] In section 1.1 below, we will demonstrate the limitation of the NOSA middleware and the motivation of our cache mechanism.

1.1 Background and Motivation

Kobialka T et al explain that the NOSA is a suite of middleware services for sensor network applications which are built upon the Sensor Web Enablement (SWE) [4], which is a set of operations and standard sensor data representations defined by the Open Geospatial Consortium (OGC) [5].

The overall structure of NOSA is outlined in Figure 1-1. Four layers have been defined, namely Fabric, Services, Development and Application. Fundamental services are provided by low-level components whereas higher-level components provide tools for creating applications and management of the lifecycle of data captured through sensor networks.

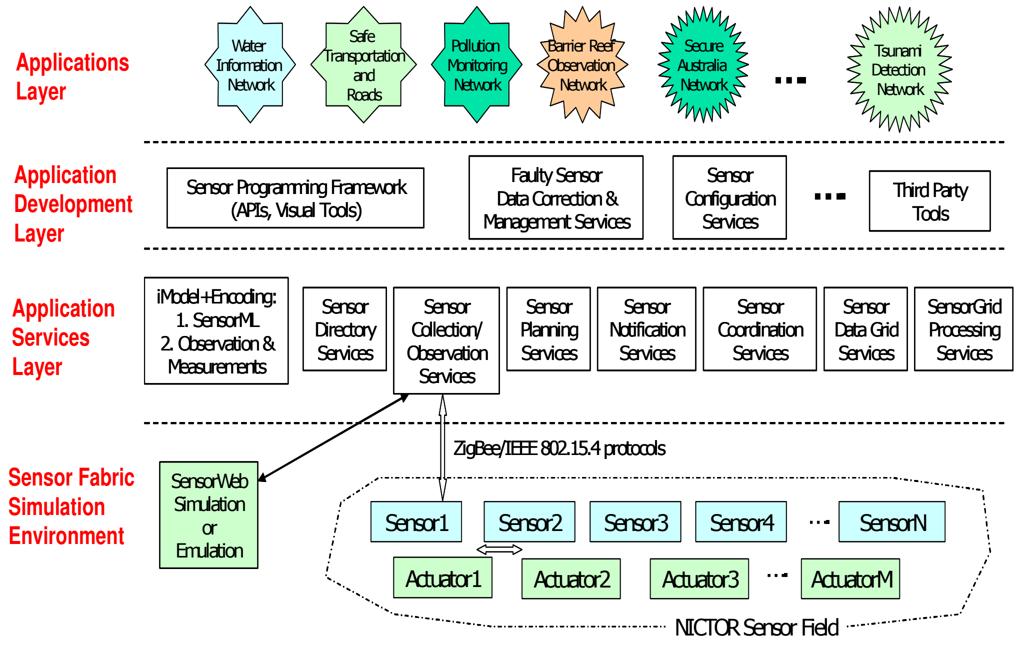

Applications Layer
Water Information Network
Safe Transportation and Roads
Pollution Monitoring Network
Barrier Reef Observation Network
Secure Australia Network
...
Tsunami Detection Network
Application Development Layer
Sensor Programming Framework (APIs, Visual Tools)
Faulty Sensor Data Correction & Management Services
Sensor Configuration Services
...
Third Party Tools
Application Services Layer
iModel+Encoding: 1. SensorML 2. Observation & Measurements
Sensor Directory Services
Sensor Collection/ Observation Services
Sensor Planning Services
Sensor Notification Services
Sensor Coordination Services
Sensor Data Grid Services
SensorGrid Processing Services
ZigBee/IEEE 802.15.4 protocols
Sensor Fabric Simulation Environment
SensorWeb Simulation or Emulation
Sensor1
Sensor2
Sensor3
Sensor4
...
SensorN
Actuator1
Actuator2
Actuator3
...
ActuatorM
NICTOR Sensor Field

Figure 1-1: High-level view of NICTA Open Sensor Web Architecture [3]

According to [3], the four specifications of SWE implemented in the NOSA, Sensor Collection Service (SCS) [6], Sensor Planning Service (SPS) [7], Web Notification Service (WNS) [8] and Sensor Repository Service compose the core services set of NOSA. Figure 1-2 below shows the typical invocation for Sensor Web Client using those services.

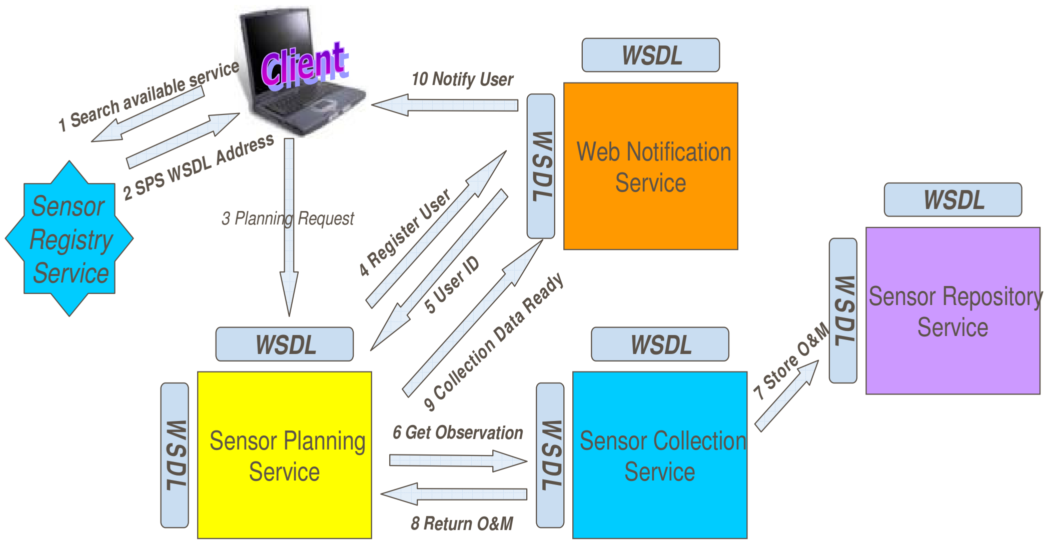


Figure 1-2: A typical invocation for Sensor Web client [3]

Among those four core services, the SCS is the largest and most important service in the NOSA architecture. The responsibility of the SCS is communicating directly with the sensor networks. It receives incoming SOAP requests and then passed the query to the sensor network via a proxy. The SCS provides the proxy interface to both streaming data and query based sensor applications that are built on top of TinyOS [10] and TinyDB [11]. The proxy worked with streaming data, which are collected directly from sensor nodes, is the Techfest proxy, and the one worked with TinyOS and TinyDB that fetching data from TinyVIZ simulation software is TinyDB proxy. After sensing, the proxy collects the resulting queries information and translates the raw observational data into a XML Observation and Measurement (O&M) [9] encoding and then returns the encoded observation data to the connecting client.

According to [3], the SCS could handle simultaneous queries to heterogeneous Sensor Networks, but the experiment results of the [3] shows that as the number of simultaneous clients steadily increases so does the response time. Degradation in the performance is an outcome of the limited concurrency available at the sensor network level of the service because only one query can be processed by the sensor network at a time. In this case, if the SCS get multiple requests at the same time, the queries have to be placed in a queue at the sensor network accessing point until the result of the current executing query obtained from the sensor network.

The solution to this problem could be eliminating or reducing duplicate data in the queries through implementing a cache in memory and in a database. In this approach, if a new query is received that is similar to one which has recently been executed, then the result can be anticipated as being the same and resulting observational data could then be pulled from the cache. This would reduce the number of duplicate queries sent to the sensor network and improve the scalability. [3] To develop this solution, our group has implemented a cache mechanism as part of the NOSA project.

1.2 Cache Mechanism

The cache mechanism is implemented as part of the NOSA middleware, as mentioned before, the NOSA middleware consist of four layers: application layer, service layer, sensor layer and physical layer. Basically, we focus on the lower three levels to build up the cache mechanism to analyze the query requests, which come from SCS. With the cache mechanism, the SCS will return an observation result back to the client either directly from cache or it will forward the query to the sensor network. For processing, this is determined by the environment parameters, user defined setting or by the cache system and intelligently analysis of historical data sets. The layer structure of the NOSA middleware and how the cache mechanism fits in the existing architecture are shown in Figure 1-3 below.

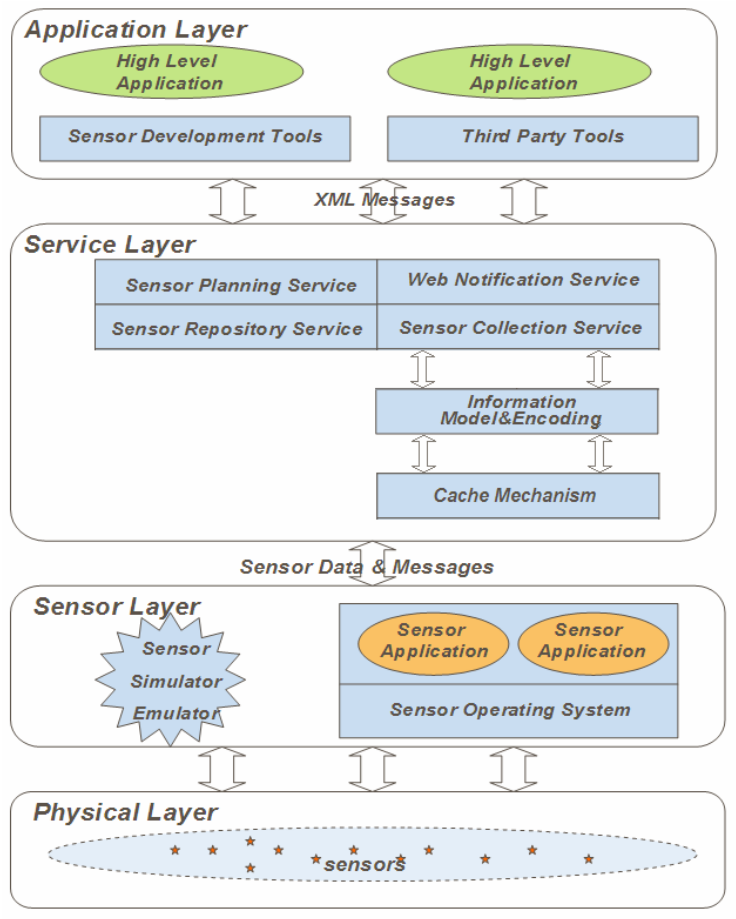


Figure 1-3: Layer architecture of the NOSA middleware [3]

The cache mechanism sits between the SCS and the physical sensor networks or TinyVIZ simulation environment, working with the corresponding proxy to processing the query requests and observational results data sets.

The cache mechanism includes several novel properties, as indicated below:

1. The cache mechanism currently composes of two level caches: memory cache and database cache. Generally, memory cache is fast, but the size of memory is largely restricted by the runtime environment. By contrast, a database cache tends to have much more space, while its speed is lower than the memory. So combining them together is good way to achieve high speed and large space. More cache level can be made to satisfy specially needs (like tape media accessing).
2. The cache mechanism has adjustable parameters to control the physical characters of the cache and the logic of insert, delete or analyze the cached queries and results. This includes the cache size, lifetime of the cached entries, different discard policies for deleting the cached entries, estimation for evaluation of the effect of historical results, threshold to decide whether the cached entries are reusable or not. All of these parameters will be discussed in detail in the section 2.2.
3. The cache mechanism is designed for working with types of proxies connected, including streaming data and query based sensor applications that are built on top of TinyOS and TinyDB.
4. The mechanism includes a Rule Engine that can exploit the similarity between queries, analyze the historical data stored in the cache and calculate the trend of environment changing. According to the analysis result, the mechanism could produce a prediction of the result of further incoming queries. The estimation and threshold mentioned in the second property are implemented in the rule engine.

1.3 Related Works

Most sensor applications are used to monitor changes of physical phenomena, for example sound, light or temperature. The sensor network could be considered as a distributed database, which comprises of the sensing data and the description of characteristics of the sensor nodes, such as the location and the type of the sensor. [12]

Since the sensing data collected by one sensor at one time could only reflect the physical phenomena at a small region and the changes during a short period of time, the sensor database requires processing the information in a different way with traditional database system. The sensing service should map the raw sensor readings with the sensor characters onto the physical reality and, in this case, a model of that reality is required to complement the readings. On another hand, *adaptive sampling* is sufficient for the model, which means if sensor nodes only send data when the observed physical phenomena change. This allows answering queries directly at the gateway rather than fetching every tuple from the network. The idea has been proposed in [13] where a statistical model is run at the gateway. Whenever the prediction of the model does not reach the confidentiality specified in the query, the gateway actively requests additional data from the sensors in order to update the model parameters and thus increase the quality of the prediction. In our cache mechanism, the rule engine plays a similar role as such a model that the rule engine will update the parameters for the cache system to use the historical results in a more intelligent and efficient way. As an extension of this idea, one can consider running a less complex model at the sensor nodes that allows the node to decide on its own when the model at the gateway is outdated and requires new data to update its parameters. By moving from a pull-based to a push-based mode of operation, the number of messages can further be reduced since no explicit requests have to be sent.

Besides the [13], many other query aggregation architectures and approximate querying models in sensor networks have been proposed to solve how to collaboratively process the sensor reading and how to build a model to analyze historical data to predict the current environment status. The [14] proposes a Similarity Aware Query Processing scheme, which can answer queries by exploiting the similarities among different queries issued to data-centric storage sensor networks. The basic idea is to replicate results for previously issued queries as materialized views in the network and utilize the materialized views to answer similar queries. In our cache mechanism, we have implemented a Comparer, whose responsibility is comparing the current executing query with cached queries. The algorithm, which we utilized to split a query and fetch the useful parts from the result, is a simplified version of the one discussed in the [14], because the query split algorithm in [14] is for multi-dimensional range query, which is a different case with our proposal.

The query aggregation frame proposed in [15] used for application that have the features that the query rate is high due to a large number of users sending queries, while the response data to the query is simple. This might be the case of the NOSA applications, since the response data only containing temperature, sound or light values. One disadvantage of the query aggregation frame is that it does not consider the topology of the existing sensor network and this also might be a future work for our cache mechanism. One novel feature of the cache system in [16] is partial match caching, which ensures that even partial matches on cached data can be exploited and that correct answers are returned. The cache mechanism that we have implemented integrates all the valuable ideas of the approaches or models referred above, combines with the existing NOSA middleware to achieve a more flexible and practical system. It considers not only the query processing model, which could exploit the similarity of queries and parse the historical result in the cache to estimate the physical phenomena, but also a configurable two level cache system to implement those models.

The remainder of this paper is organized as follow. Section 2 shows the architecture and design strategy of our cache mechanism and describes the software and hardware developing environment. A discussion concerning the limitation of the simulation and test environment will also be included in this section. The experiment results are in the section 3, and conclusion and future work are in the section 4.

2 Architecture and Design

In this section, we will illustrate the system architecture of the cache mechanism and the key components of each mechanism. This includes the locking scheme for handling concurrent queries, the two level cache chain, rule engine working principles and the query aggregation rules. They are used for parsing the cached queries and predicting the results for current queries.

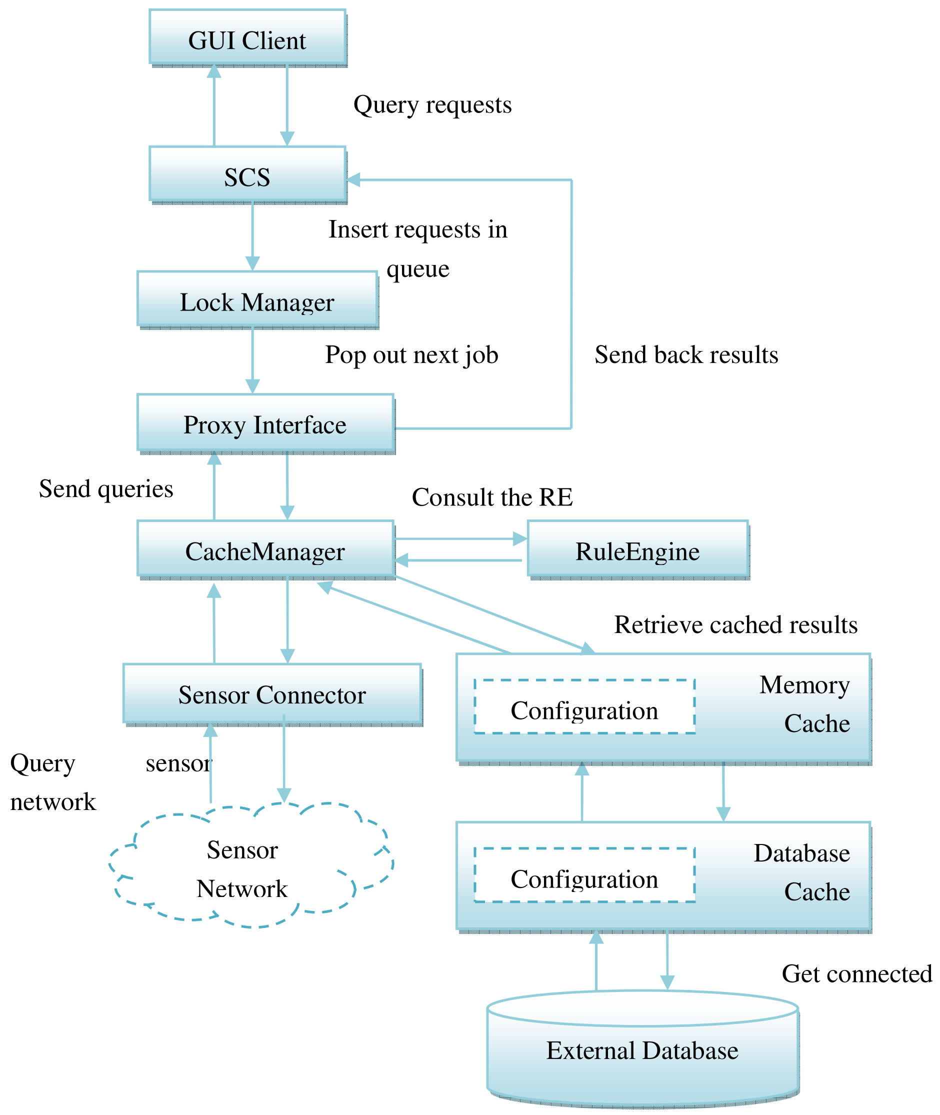


Figure 2-1 Architecture of the cache mechanism

2.1 Architecture Design

System Structure

The cache mechanism is implemented with the Sensor Collection Service. It interacts with the proxy interface, storing the results retrieved from the physical sensor nodes and utilizing the cached result to analyze and predict the observation time for current queries. Figure 2-1 shows the architecture of the cache mechanism. As indicated in the figure, the system consists of several components: LockManager, CacheManager, RuleEngine and the Cache, connecting with the existing NOSA components: SCS, GUI client, proxy interface and sensor connector.

- **LockManager** sits between SCS and Proxy Interface. It aims at providing concurrent access control to the sensor network. Before connecting to a sensor connector, the sensor proxy has first get a lock from LockManager. The operation for getting a lock is a blocking operation, which means control cannot be returned to the sensor proxy until the lock is granted. When query is finished and result is fetched from sensor connector, the lock should be returned. By doing this, another request in queue can acquire the lock and continue processing.
- **CacheManager** is an entrance to the cache(s) from a SensorProxy's point of view. Inside a CacheManager, there is a RuleEngine and a pointer which points to the first cache of the cache chain. Upon receiving a request, the CacheManager first asks the RuleEngine whether is time to access sensor network. If yes, CacheManager returns null immediately so that the sensor proxy will query the sensor network. If no, CacheManger check each level of the cache and return a suitable cache value or null if no suitable cache value found. Whenever a sensor proxy queries the sensor network, it should also feedback the result to CacheManager when the query is done. With the feedback function invoked, the CacheManager will store the result into every level of cache as well as feedback the result to the RuleEngine.
- **RuleEngine** is an advisor for CacheManager to tell whether to access the sensor

network or not. The parameters inside the RuleEngine are used to make decisions. These parameters are dynamically changed by the surrounding environment via the feedback function. If the environment is changing fast, the RuleEngine will tell system to access sensor network (instead of cache) more frequently.

- **Cache** provides caching facilities. Cache acts like a map. Providing a key, there is at most one value corresponding to that key. In the project, key is defined as the query string, while value is the result from sensor network. Cache is actually an interface, so that it can be implemented in many ways flexibly to meets different systems' needs.

System Interaction

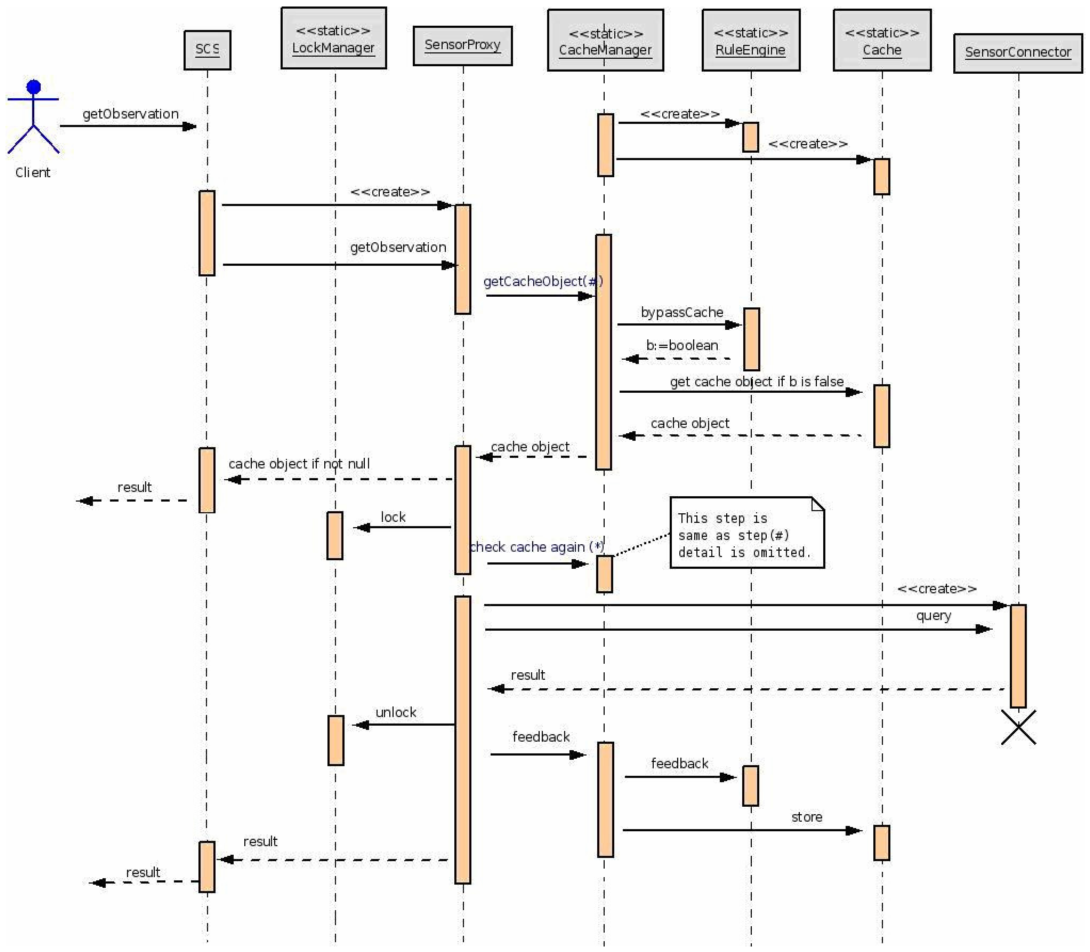


Figure 2-2 Work flow of the cache mechanism

The Figure 2-2 is a sequence diagram which shows the work flow of the system. The clients send out the query request initially, after the SCS receiving the request, the services will create a SensorProxy to handle the request. SensorProxy will ask the CacheManager to check whether there are cached results available (This is first check). If the cache is available, the corresponding cache entry will be fetched From Cache and send back to client. If no cache entry is available, the cache manager will insert the request into the queue. When a request comes out from the queue, it will check again whether there's cache available (This is second check). The second check is the same as first check, but omitted in Figure 2-2. This double check makes sure that cache is reused if another request gets a value back while this request is waiting in the queue. This might happen. Let's assume there's no cache at the beginning. Now request A comes. Obviously A cannot find suitable cache entry. So it gets the locks and access sensor network. Now B comes with same query. Since A has not return yet from sensor network yet, B will not find any cache result, so it has to enter the queue and wait for the lock. If there's no double check after B get the lock later, B will access the sensor network immediately, which is not efficient enough because A has come back with a reusable result. In order to reuse A's result, the double check is necessary. Next, if a cache is available after the double check, it is sent back to the client. If there's still no cache available after double check, SensorProxy will connect to the sensor network via SensorConnector. After value is read from sensor, it will be sent back to client and stored in all cache levels.

When we talk about getting cache entry from Cache, there are two steps.

First, we have to discover, whether there are cache entry keys similar to the request one. Key is actually the query string. The key is stored as a string likes "SELECT * FROM SENSOR WHERE light > 90", other substring like "features …" from the original query string is removed. Two keys are "similar" if they are exactly the same or criteria of if the queries are within some threshold. The threshold is changed dynamically by the RuleEngine to reflect the environment, while the decision of whether they are similar enough is given by Comparer. For examples, let's say the initial threshold is 3. If the first query is "SELECT * FROM SENSOR WHERE light > 30" and the second query is "SELECT * FROM SENSOR WHERE light > 33". We

define these two query is "similar" and the result from the first query is reusable because "33-30 <= 3". Both RuleEngine and Comparer will be discussed later on Section 2.2
Second, if no cached queries are "similar" to the request query, we then use Comparer to refine the cached Object by removing, adding or combining some of them together to form a new result. For examples, if the one of the cached query is "light > 90" and the request query is "light >100". Then all values between 90 and 100 of the cached entry will be removed. The values left becomes the result for query "light>100".

On the other hand, storing new result back to cache is also not in a single step. First, we have to determine where to put the result. If the cache already has one entry with exactly same query string, then we can safely replace it. If no, a new entry will be allocated and put into the cache. This step cannot process if the cache is already full, which means the size of entries is equal to the configured parameter "maxCacheSize". When cache is full, we have to first remove some of them. There's a lot of replacing policies in other sophisticated cache system. Here only two are implemented: least rank and oldest. The first one keen to remove the one which is used least by using rank in descending order, while the latter one keen to remove the oldest one by using lifetime in ascending order. After being stored in the cache, the result is also feedback to the RuleEngine to be analyzed. In RuleEngine, the result along with received time (of that result) is analyzed to make changes to "estimate" and "threshold".

2.2 Components Design

Both SensorLockManager and CacheManager are used and only used in the sensor proxy. Because both SensorLockManager and CacheManager imply single instance pattern, there's no code in the sensor proxy class to create or destroy these classes. Just as shown in Figure 2-3, the relation among sensor proxy and these classes are association, which means sensor proxy just knows where they are but not has them. For example, the sensor proxy can lock the sensor network simply by invoking "SensorLockManager.lock()". Similarly, the instance of CacheManager can be retrieve by invoke "CacheManager.getInstance()".

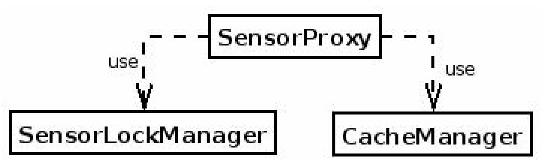


Figure 2-3 Relation between Sensor Proxy and new Components

The Figure 2-4 shows the relations between each component. Please refer to Appendix II to see the full classes diagram. There's only one CacheManager across the system. Within the CacheManager, there's one RuleEngine and a pointer pointed to the Cache on first level. Each Cache has a pointer to the next level Cache, unless there's no more lower level cache. Besides the pointer, there's a configuration class named CacheConfig, holding all configuration parameters. Two classes is also hold in the CacheConfig. One is Comparer for doing query key comparing quest. The other one is DiscardParser, which is responsible for ensuring replacement policy. Both of these two class are also configurable from the configuration file.

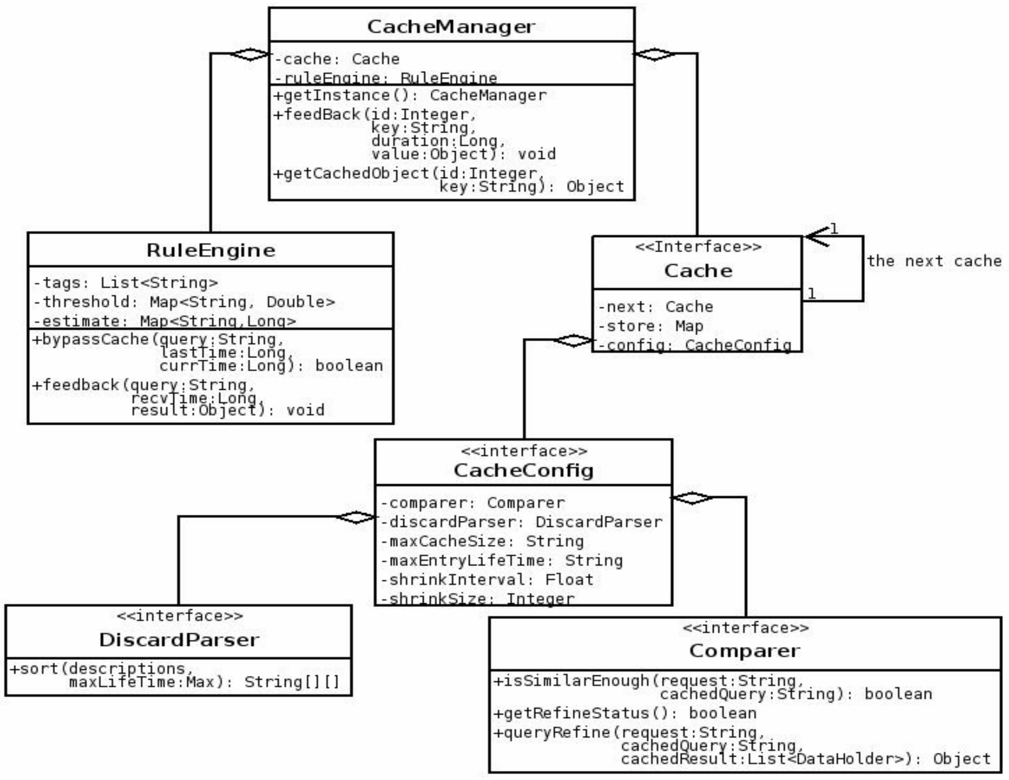
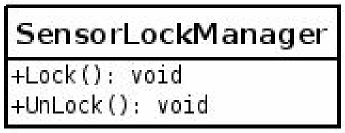
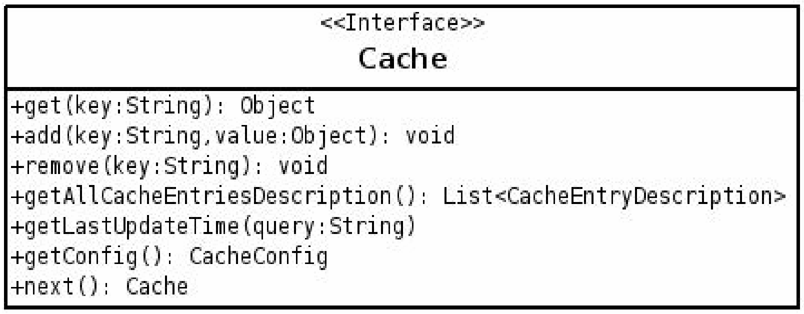

CacheManager
-cache: Cache
-ruleEngine: RuleEngine
+getInstance(): CacheManager
+feedBack(id:Integer, key:String, duration:Long, value:Object): void
+getCachedObject(id:Integer, key:String): Object
RuleEngine
-tags: List<String>
-threshold: Map<String, Double>
-estimate: Map<String,Long>
+bypassCache(query:String, lastTime:Long, currTime:Long): boolean
+feedback(query:String, recvTime:Long, result:Object): void
<<Interface>>
Cache
-next: Cache
-store: Map
-config: CacheConfig
1
1
the next cache
<<interface>>
CacheConfig
-comparer: Comparer
-discardParser: DiscardParser
-maxCacheSize: String
-maxEntryLifeTime: String
-shrinkInterval: Float
-shrinkSize: Integer
<<interface>>
DiscardParser
+sort(descriptions, maxLifeTime:Max): String[][]
<<interface>>
Comparer
+isSimilarEnough(request:String, cachedQuery:String): boolean
+getRefineStatus(): boolean
+queryRefine(request:String, cachedQuery:String, cachedResult:List<DataHolder>): Object

Figure 2-4 Cache Related Classes Diagram

2.2.1 Locking scheme



The lock manager provides a method to queue the incoming requests. Although it looks like a separate part in the system structure, it provides the cache system the fundamental function of handling concurrent requests. LockManager is the only one class in this part. When lock() is invoked, it appends the request to the queue and pops it out when unlock() is called by the other.


<<Interface>>
Cache
+get(key:String): Object
+add(key:String,value:Object): void
+remove(key:String): void
+getAllCacheEntriesDescription(): List<CacheEntryDescription>
+getLastUpdateTime(query:String)
+getConfig(): CacheConfig
+next(): Cache

2.2.2 Cache

Cache is linked. Each Cache has a pointer pointing to the next Cache (we can use next() the get the next Cache). Different levels of caches work together. On the other hand, Cache behaves like a map. Each cache entry has a corresponding key, which is the query string. Therefore, objects can be stored/retrieved to/from Cache by normal Map functions add()/get().

Like many other cache system, there are many parameters that control the runtime behavior of the cache. These parameters are packed in the configuration class. So that cache can focus on dealing with how to access and manager storage media, while all configuration stuff is handle by the configuration class.

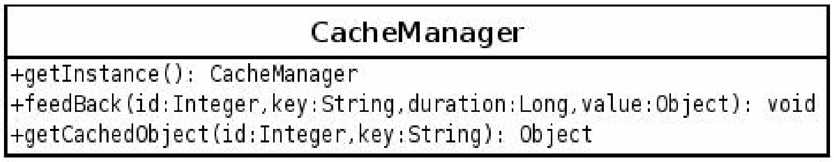
Parameters are listed below:

1. ***Max cache size:*** This parameter determines at most how many cache entries can be stored in the cache. It is critical for memory cache because memory is limited on most of the system. Cache size will not be reduced by a certain amount until a new entry comes in.
2. ***Max entry life time:*** This restricts how long a entry can reside in the cache. Together with max cache size, these two parameters decide how the cache grow and shrink.
3. ***Shrink interval:*** Cache is shrunk at a certain interval time. Expired entries are removed at each shrink process. Too small shrink interval time makes system very busy, while a large one makes cache might have lots of expired entries.
4. ***Shrink size:*** When the cache is full, some entries have to be removed. Shrink size determines at most how many cache entries will be removed at a time.
5. ***Comparer:*** Comparer is used to provide information of how related those cache entries are with current request and how can they be reused. It will be discussed later in this section as a separated module.
6. ***Discard Parser:*** The discard policy is fairly flexible. Basically, the policy is just a string that can be in any format. What we need is a parser to parser that string and provide accordingly function. Discard Parser is what we need. The only one function in this class is used to sort all cache entries in a certain order. With this ordered cache entries, a cache can remove its entries one by one to ensure the discard policy. Currently, only “rank; lifetime”

and "lifetime; rank", which mean "least used first" and "oldest first" respectively, is implemented.

2.2.3 Cache Manager

CacheManager is the gateway to all function except the lock facility. This is the only class that a client should inter face to. Cache manager uses singleton pattern, which means there is only on instance of CacheManager at any time across the system. Method getInstace() is used to get the instance reference. This ensures different clients use the same cache.

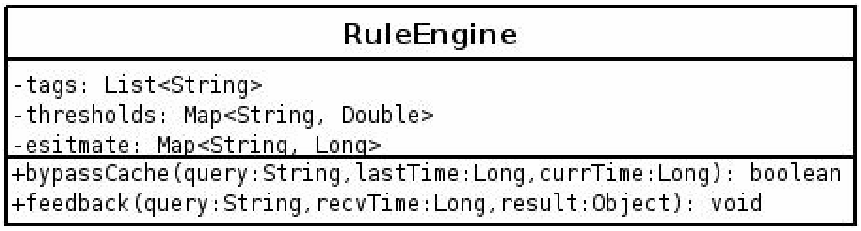


```
CacheManager
+getInstance(): CacheManager
+feedBack(id:Integer,key:String,duration:Long,value:Object): void
+getCachedObject(id:Integer,key:String): Object
```

One of the major tasks for CacheManager is to deal with the cache along the link. Let say we have memory cache and database cache (which is the current settings of the project). When acquiring cache object, if it resides in the database but not memory, it will be fetched from the database and stored in memory as well as sent back to client. When storing a cache entry, it will be stored to both cache level (memory and database). CacheManager also communicate with the RuleEngine. Whenever it is asked for cache object, CacheManager asks RuleEngine whether to query the cache. If the answer is yes, then look up the cache object in each level of cache by the order defined in configuration file (commonly first memory second database makes sense). If no, it will send back null object to tell the client to query the sensor network. CacheManager also pass the result object to RuleEngine so that it can analyse the result.

2.2.4 Rule Engine

The cache



```
RuleEngine
-tags: List<String>
-thresholds: Map<String, Double>
-esitmate: Map<String, Long>
+bypassCache(query:String,lastTime:Long,currTime:Long): boolean
+feedback(query:String,recvTime:Long,result:Object): void
```

interface provides higher performance to the system, however, there’s a gap between the cache result and the current query being processed by the sensor network. Clearly, we need some mechanism to fill the gap, and RuleEngine does this.

Here we define the gap as the different between the latest available cached value and the current value from sensor.

Since there’s only two ways to obtain a value, either from the cache or from the sensor, we have to make a decision when to access cache and when to access the sensor. It is controlled by a parameter named “estimate”. Estimate is the estimate of how fast the environment is changing. If it is small, it means environment is changing fast. If it is larger, the environment tends to be stable. For examples, the light environment of a room with a lamp flashing is changing faster then the one in an open field under the sun. On receiving a request, the system will check the current time of that request and pass it to the RuleEngine. When CacheManager asks the RuleEngine for help, whether to access sensor network instead of cache, the RuleEngine uses this estimate to make the decision. If the request time exceeds the last update time of cached result plus the estimate, then the request is redirect to the sensor network, because “the environment is changing fast”.

Before going down to further explain how estimate works, a description of each functions are shown here:

bypassCache():

Client uses this function to determine whether to use the cache or not(not considering whether there’s cache value available yet, which is determine by other classes named Cache and Comparer)

feedback():

After receiving the result from sensor network, client uses this functions to give the RuleEngine feedback. Then the RuleEngine uses the result and time to change its estimate and threshold.

Now we are going to describe how estimate works. The estimate is initialized in the configuration file and dynamically changed by the RuleEngine at runtime. When a client (Actually, the "client" is a SensorProxy. We name it "client" because from CacheManager's point of view, it is the client) feedbacks the result read from sensor network to RuleEngine via CacheManager, the result is analyzed. As described above, CacheManger will first store the result to Cache, then pass it to the RuleEngine. How much the result changes compared to last cached result will affects the estimate. By a formula, we calculate the difference and applied it to the estimate. So estimate will increase if difference is small and decrease if difference is large.

In the following, a formula for the estimate will be presented and discussed.

Assumption**:**

We have i sensors:

$$S_1, S_2, S_3, \ldots, S_i \quad (i \in I)$$

Last reading for each sensor is recorded as well as the time. The record format is:

$$(V_1, T_1), (V_2, T_2), (V_3, T_3), \ldots, (V_i, T_i) \quad (i \in I)$$

*where V stands for value and T stands for time.

Scenario**:** At T time, we have k $(k \in K)$ new readings for some of the sensors:

$$(V_{k_1}^{'}, T), (V_{k_2}^{'}, T), (V_{k_3}^{'}, T), \ldots, (V_{k_k}^{'}, T) \quad (k_j \in K)$$

Goal**:** calculate the new estimate.

Formula**:**

```
$for \quad all \quad V_i, i \in I$
    $if \quad i \in K$
        $\Delta V_i = \left(V_i' - V_i\right)$
        $\Delta T_i = 1/(T - T_i)$
    $else$
        $\Delta V_i = 0$
        $\Delta T_i = 0$
$\Delta T = \sum_{i \in I}^{i} \Delta T_i$
$n_i = sizeof \ (I)$
$n_k = sizeof \ (K)$
$dev' = \sqrt{\frac{1}{n_k} \sum_{i \in I}^{i} \Delta V_i^2 (\Delta T_i / \Delta T)}$
$dev = last \ time \ dev$
$portion = n_k / n_i$
$sign = dev' > dev \ ? -1 : +1$
$estimate' = (1 - portion) * estimate$
            $+ sign * portion * estimate * dev' / dev$
```

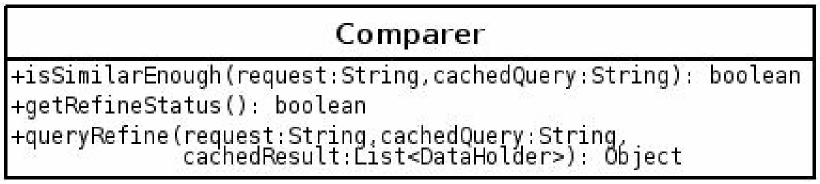
***Explanation*:**

Firstly, we have to calculate the differences of each new value, $\Delta V_i, i \in I$. Then we calculate the Standard Deviation dev' with weight $\Delta T_i / \Delta T$. The weight comes from the reciprocal of time difference because the larger the time difference is, the less the environment changed.

Secondly, we need to determine the sign and portion. Sign is important to control the estimate whether to grow or decrease. When current dev' is larger than the last dev, which means environment is change faster, the sign is negative, so that the estimate will decrease. Portion is the ratio of change of estimate. Since we only received n_k results out of n_i, thus the portion of change is n_k / n_i.

At last, every thing is ready to calculate the new estimate. (1-portion) of the old estimate remains the same. The rest of the old estimate increase/decrease by dev' / dev.

2.2.5 Comparer



```
Comparer
+isSimilarEnough(request:String,cachedQuery:String): boolean
+getRefineStatus(): boolean
+queryRefine(request:String,cachedQuery:String,
            cachedResult:List<DataHolder>): Object
```

Comparer is the components for comparing the current queries with the historical queries stored in the cache, after analysis, the comparer will make a decision of whether the cached result is reusable and how to reuse the cached result, either directly copy the old value or refine the cached result based on the current queries.

The comparison rules are shown in the table 2-1 below:

Table 2-1 Query comparing rules

Compare Property	Compare Operator			Compare Value	Return Value
Different	-----			----	False
Same	Different			-----	False
	Same	-----	-----	Same	True
		S: >	C: >	V1>V2	True, if V1-V2<threshold
		S: >	C: >	V1<V2	True, refine required when V2-V1>threshold
		S: >	C: >	V1>V2	False, if V1-V2>threshold
		S: <	C: <	V1>V2	True. Refine, when V1-V2>threshold
		S: <	C: <	V1<V2	True if V2-V1<threshold
		S: <	C: <	V1<V2	False, if V2-V1>threshold
		S: =	C: =	-----	False
		S: <>	C: <>	-----	False

* S = Storage Query, C = Current Query, V1 = Storage Value, V2 = Current Value
False = the cached results cannot be reused, True = the cached result could be reused

There are two cases that the results need to be refined to fit the current query. The refining method is deleting the useless part of the results. For example, the cached query is

SELECT light

FROM all sensors

WHERE light>30

And the current query is

SELECT light

FROM all sensors

WHERE light>37

Assuming the threshold is 5, which means the cached result cannot be reused directly, because 37-30=7 is bigger than 5. Result is required to be refined in this case. Considering the cached result of the historical query that consists of several data elements:

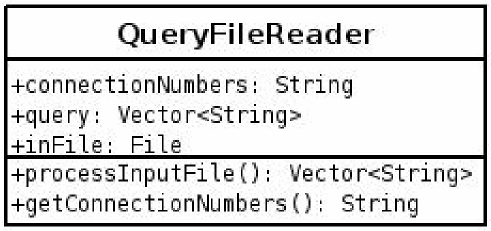
Table 2-2 Result

Node id	Reading Value
2	40
3	32

As shown in Table 2-2, the first element still meets the requirement of the second query, so it will be kept in the result, and the second reading, which is smaller than 37, will be deleted. The refined results could be retrieved back to the client directly without visiting the physical sensor nodes.

2.2.6 GUI client

User friendly is an important criterion for a system. The existing NOSA includes a client GUI interface, which could send queries to the services but cannot specify the queries. For improving the system usability, we modify the input part of the client by adding features that user could choose different methods to produce queries:


QueryFileReader
+connectionNumbers: String
+query: Vector<String>
+inFile: File
+processInputFile(): Vector<String>
+getConnectionNumbers(): String

1. Manual input some particular query
2. Input from a file that including the specification of queries. File is read and processed by QueryFileReader as shown on the right.
3. Random produce queries

This makes the experiment of the cache mechanism much easier and more efficient. Another improvement we made is changing the queries produced time. In this case, the client program could be used as a test case producer, all the queries producing processes are under control.

2.3 Limitations

At this stage, the NOSA use centralized storage architecture to process the queries and results. The storage capabilities of sensor nodes have been ignored. Since communication is more energy-consuming compared to local computation, in-network aggregation could reduce energy consumption. Combining the in-network aggregation with the query aggregation in the cache system will be a better solution.

The TinyViz simulation environment is not stable, which is an obstacle when implementing the cache mechanism. In the developing and experiment processes, we try to use the physical sensor instead of the simulator.

3. Implementation

The cache mechanism has to work with the existing NOSA system, so that we try to follow the same programming pattern in the developing process, such as:

Object Factory: Many classes are initialized within the object factory using parameters from "application.properties" file. This unified instantiation of object can make system highly configurable.

Interface and Abstract: Besides using the object factory, we also create many interfaces and abstract class to represent the system.

Configurability: The parameters for the Cache Mechanism are configurable via the "cache.conf" file. Like the other part of the project, by doing this, we can make changes to the system without recompiling and redeploying the software. For detail of how to configure, please see the Appendix I.

The development environment and software tools for the cache mechanism are[1]:

1. Java

   JavaSE5 is the runtime and develop kit for the project.

2. Eclipse3.2

   Eclipse is the IDE for this project, combined with Tomcat Plugin and subclipse for subversion controlling.

3. TinyOS 1.x

   TinyOS runs on a different Java runtime, which is Java1.4 inside cgywin. We have tried to run it on a outside Java runtime, like Java5 described above, but it doesn't work. Java1.4 comes with the installation of TinyOS.

4. Tomcat 5.5

   Tomcat 5.5 is the server for hosting the web services, including SCS. The directory "$TOMCAT_HOME/common/class" is the place we place our configuration files.

5. TinyDB

TinyDB combined with the GUI (TinyViz) is used to simulate the sensor environment. During testing, 10 sensors are used for the simulation.

4. Experiment and Results

In this section, we measure the performance of the cache mechanism through two different aspects: system processing ability and system accuracy with physical sensor networks, which consists one base station and six MICAz sensing nodes. The system processing ability is the major improvement comparing with the original system, as mentioned before, in the original version of NOSA system, the ability to handle concurrent queries is quite limited and all the queries will be passed to the sensor network directly. Considering the properties of sensor network, the original system could consume excessive resources, including power of physical sensor nodes, system processing time and the bandwidth of the network connection. In the experiment, we use different data sets to test the processing ability of the system with cache mechanism and compare the result with original system without the cache. System accuracy is another point to evaluate the cache mechanism, because the cache mechanism will utilize the historical data to predict the results of current queries, it is unavoidable to introduce some errors. We will show in the experiment that these errors are well controlled by adjusting the parameters of the cache system. Both two tests are focus on querying the light property of the environment because of limitation of the current NOSA system.

4.1 Processing Ability Test

4.1.1 Data Sets

System processing ability, in another words, it is how the system could handle queries simultaneously in our cache mechanism. This test is based on three groups of test data sets, as shown below:

Case one:

```
SELECT light
FROM all sensors
```

```
WHERE light>30
```

Case two:

```
SELECT light              SELECT light
FROM all sensors          FROM all sensors
WHERE light>30            WHERE light>31

SELECT light              SELECT light
FROM all sensors          FROM all sensors
WHERE light>37            WHERE light<40
```

Case three:

```
SELECT light
FROM all sensors
WHERE light > or < a value between 20 and 40
```

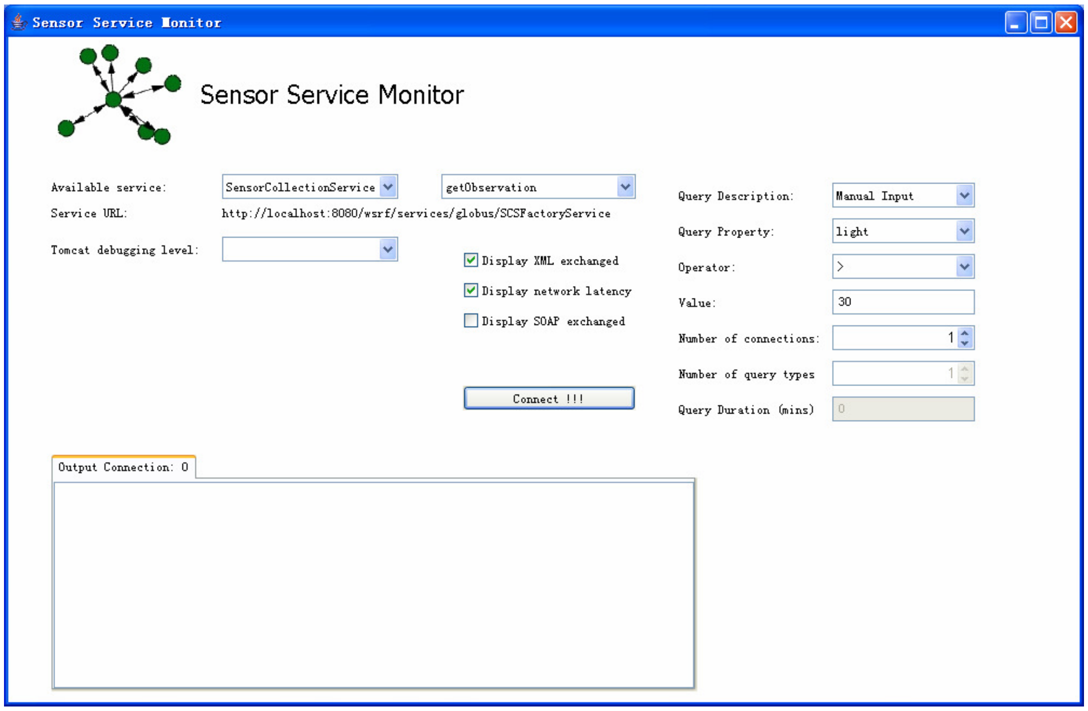
All the query values are in a range of 20~40 lux, because we assume the environment is relatively stable, which is the most case in reality also. The first two test cases are pressure test cases, where test case one is the simplest case and case two contains all the possibilities of variety of queries which are:

1. Same operator and value difference smaller than the threshold
2. Same operator and value difference bigger than the threshold
3. Different operator

The third case is simulating the real situation. We send different number of such three kinds of queries to the system and measure the total processing time.

4.1.2 Environment Setting

For the first and the third cases, we use number of 1, 2, 4, 8, 16, 32 and 64 queries sent to the sensor network and for the second case, we use number of 1, 2, 4, 8 and 16 for each of the four queries: light>30, light>31, light>37 and light<40 so the total number are 4, 8, 16, 32 and 64. The first two queries are initialed at a time, with the time interval randomly produced in the range of 0~5 seconds, one after another. For example, in the case of test case one with four queries, the queries might be initialed at time: 0s, 2s, 7s, and 10s. The third query are produced in such way that all the queries equally divided into three parts, the interval of initial time of the queries of the first and the third parts using the value randomly produced in the range of 0~5s while the second parts using the value randomly produced in the range of 0~10s, because we aimed to build such a situation that the frequency of queries changed from time to time. In this case, the first and the third queries of the test case three are dense, but the second part is sparse.



Sensor Service Monitor

Available service: SensorCollectionService getObservation

Service URL: http://localhost:8080/wsrf/services/globus/SCSFactoryService

Tomcat debugging level:

☑ Display XML exchanged

☑ Display network latency

☐ Display SOAP exchanged

Query Description: Manual Input

Query Property: light

Operator: >

Value: 30

Number of connections: 1

Number of query types 1

Query Duration (mins) 0

Output Connection: 0

Figure 4-1

As shown in Fig 4-1, we choose the manual input, and setup the value then click connect. The rest of the test input are similar.

All the three cases will be tested under three conditions: without the cache, with the cache and with the cache working with the rule engine. As indicated in the design and implementation sections, the rule engine will make the cache system working in a more intelligent way that some of the performance controlled parameters of the cache mechanism could be modified according to the environment change. In this test, we just want to show that using the rule engine with the cache system will not change the ability of processing the queries and the advantage of the rule engine will be presented in a much obvious way in the accuracy test later.

The parameters of the cache mechanism, which read from the configuration files, are set as follow:

1. Number of Entries in the cache: 10
2. Lifetime of the cache entries: 60s
3. Threshold: 5, if the difference of query values between the current query and historical query which stored in the cache is less than five, the result will be retrieved from the cache directly
4. Estimate: 3, for the cases with cache and rule engine, first query will visit the physical sensor nodes and the next three will retrieve results from cache. This parameter is dynamically adjusted.
5. Discard policy: lifetime discard and ranking discard policies.

4.1.3 Test Results

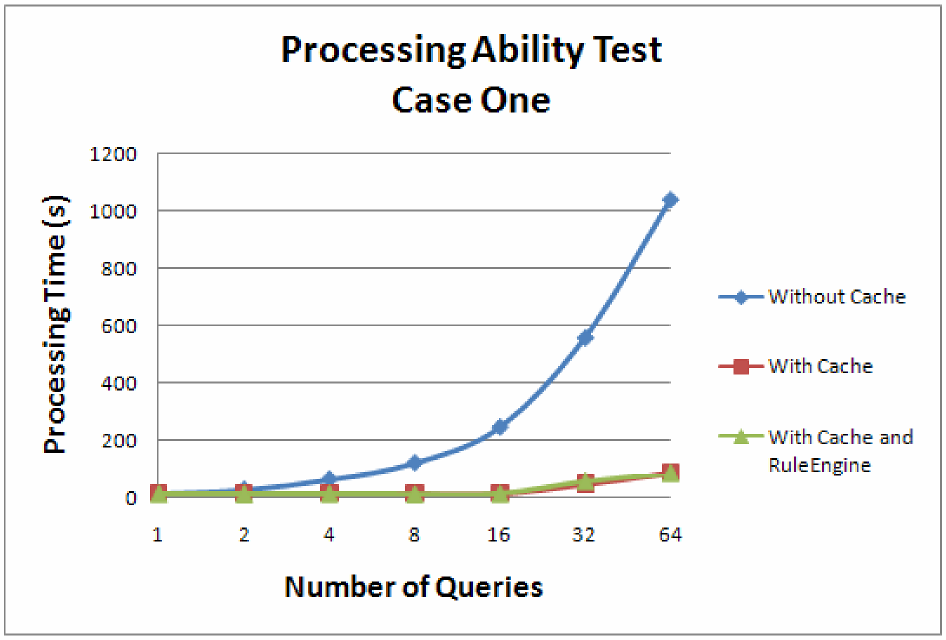


Figure 4-2 Results of test case one

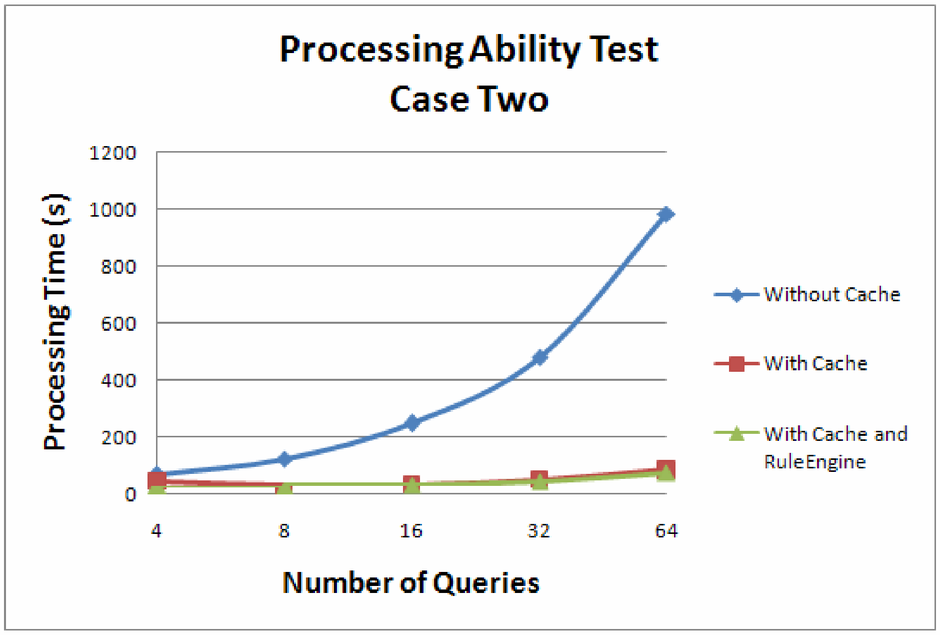


Figure 4-3 Result of test case two

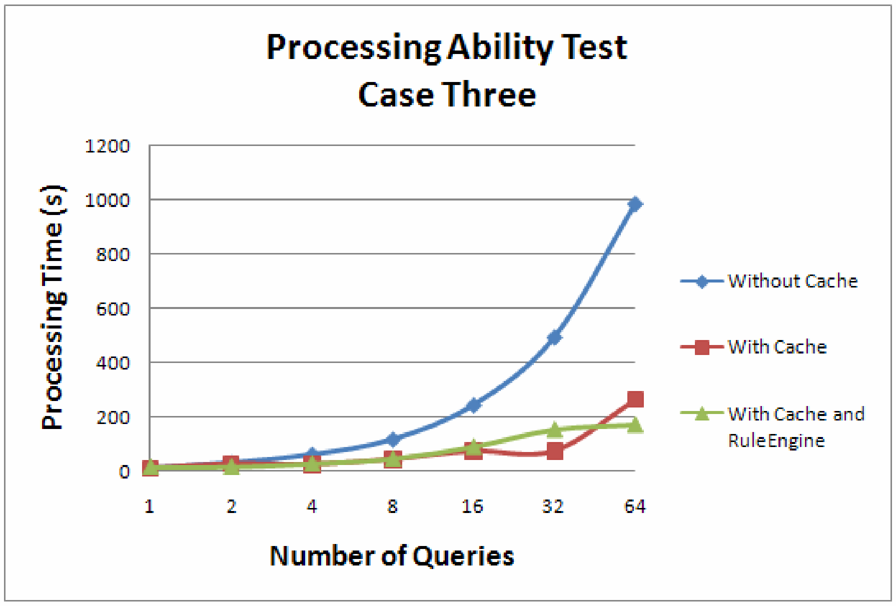


Figure 4-4 Results of test case three

The Figures 4-2 to 4-4 illustrate the experiment results of the system processing ability test above. All the three graphs show that the processing time of the system without cache mechanism is linear with the number of the queries. In certain circumstance like case one and case two, the cache mechanism increase the speed of the query processing by 20 times when there are 64 queries, and as the number of queries increasing, the times will increase too. In the third case, because the queries are generated randomly, the enhancement of the processing time is 5.7 times. The difference of performance between the first two cases and the last case shows that the probabilities of similarity of queries affects the times of cache hits. The results prove that the cache mechanism providing a sufficient improvement in the query processing ability of the system.

4.2 Accuracy Test

For testing the accuracy of the results given by cache mechanism, we deigned two different test models simulating a continuous changing environment.

4.2.1 Test Model

The test model utilizes a three levels light source:

1. Level one: deploying the sensors in a dark room and under a persistent reading lamp
2. Level two: in the same environment with level one, cover the reading lamp using two pieces of paper
3. Level three: deploying the sensors in a total sealed metal box

We switch the three levels of the light conditions following a predefined frequency, which is changing the level every minute for test case one and every 30 seconds for the test case two.

Assuming the human factors like covering the lamp, covering the sensors by the metal box will not influence the experiment results by executing these operations at the time slot between the actual sensing processes, the reading of the system without cache should reflect the environment condition, so that, comparing the results from the system with cache or the system with cache and rule engine with the results from the system without cache will show the accuracy of the cache mechanism.

The query sending to the service is defined as:

```
SELECT light
FROM all sensors
WHERE light>0
```

In all three light levels, the sensor readings are bigger than 0, so we choose the light>0 for the test cases, in such way, all the sensor readings will be saved and we can treat the reading from the system without cache as the measurement of the environment and use them for comparison as discussed before.

4.2.2 Environment Setting

In test case one, we set the related parameters as follow:

1. Number of queries: 25
2. Query frequency: 5 per minute
3. Light source level: 1-2-3-2-1 each stage lasted for one minute
4. Number of Entries in the cache: 10
5. Lifetime of the cache entries: 40s
6. Threshold: 5
7. Estimate: 3 (*threshold and estimate are same with the processing ability test)
8. Discard policy: lifetime discard and ranking discard policies.

For the test case two, we set the parameters as:

1. Number of queries: 21
2. Query frequency: 6 per minute
3. Light source level: 1-2-3-2-1-2-3 each stage lasted for 30 seconds
4. Number of Entries in the cache: 10
5. Lifetime of the cache entries: 20s

The others are same with the test case one.

4.2.3 Test Results

The test results 4-5, 4-6 below show that the cache mechanism could reflect the light condition changes in most situations, although it makes the system more environment changing insensitive, especially when the light level change from level three, which is total dark condition, to level two. The reason is when the light condition is changed, the system can not realize at the right time until the cache entry is delete from memory and it has to visit the physical sensor nodes. One possible improvement is dynamically adjusting the lifetime of the cache entries.

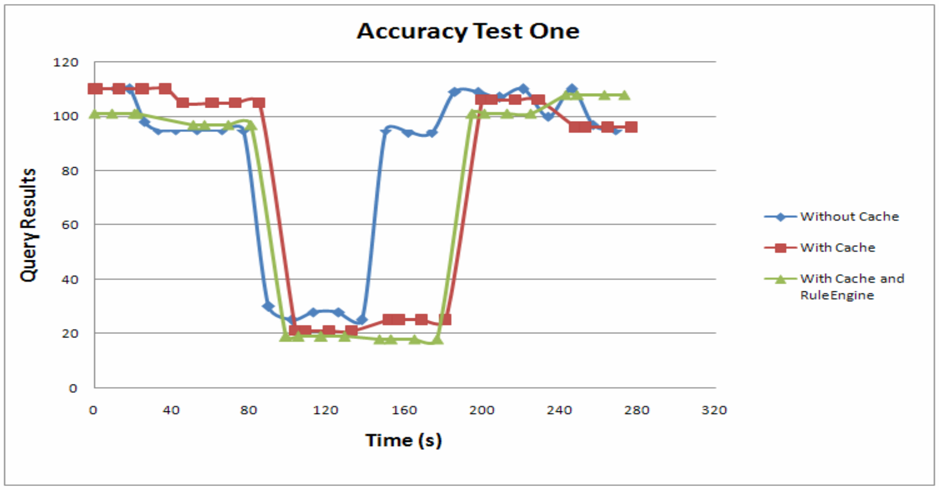


Figure 4-5 Result of accuracy test one

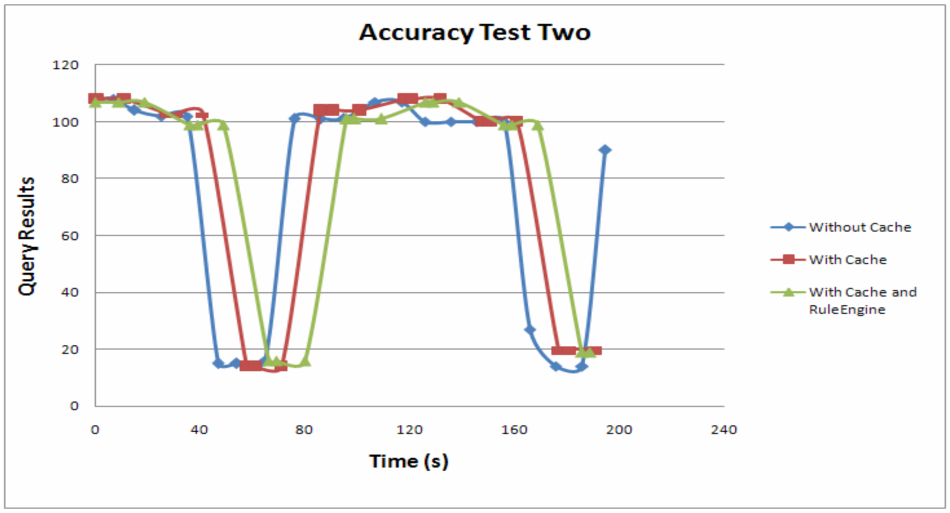


Figure 4-6 Result of accuracy test two

In the second test case, because of the more frequently changing environment, the rule engine updates the estimate from 3 to 1 after reading from the sensors at time 36s. This is one of the most important properties of the cache mechanism to make the system more intelligent and it is also one of the contributions to overcome the environment changing insensitive.

The parameters setting are the major factor that could affect the performance of the cache mechanism. As shown in the results, once we change the lifetime of the cache entries from 40s in test case one to 20s in the second test case, the delay time of discovering the environment changing is reduced from nearly 30s to 10s. This proves again that making the lifetime adjusting dynamically is an effective way to improve the system.

The test results could not reflect the accuracy in a highly precise way, because of the following reasons. Firstly, the test environment is controlled by ourselves, which means the assumption we made in the test model that human factors do not affect the experiment could not always be achieved, so that the fluctuation of the reading values is reasonable.

Secondly, the queries were initialed by the client with the same time interval and what we expected is that they will be processed with the same interval too, which means queries 0~20 were sent out at time 0s, 10s, 20s, etc and they should be processing at time 0s, 10s, 20s, etc, if we consider the query 0 was processed at time 0s. Actually, the queries might be processed at time 0s, 12s, 15s, etc, which caused by different waiting time when the queries are waiting for execution. For such reason, the query with same ID will have different execution time under the three conditions: without cache, with cache and with cache and rule engine. For example, in the accuracy test case two, query 4 was executed at time 25s, 27s and 19s under those conditions respectively. The figures present that in some cases the accuracy of the cache with rule engine is worse than the one without rule engine. Due to the reason discussed above, this proposition cannot be proven. In contrast, when the environment changing frequency reduce to switch level every 30s, which is the test case two, the number of fetching results from cache directly is reduced from 3 (initial estimation setting) to 1, which is a evidence of improved accuracy since the reading from sensor network is more accuracy.

5. Conclusions and Future Work

The cache mechanism is working with the existing NOSA project, providing an enhanced ability to handle multiple concurrent queries by storing the results of historical queries, analyzing the relationship between current queries with cached queries, reusing entire or part of the historical results to reduce the query traffic. The cache mechanism includes many configurable parameters and some of them could automatically adjust referring to the changes of the physical environment. These features make the system more intelligent. The experiment results prove the tremendous improvement of query processing ability of the system, and the self-update functions.

As discussed in the report, the lifetime of the cache entries should be updated dynamically, which will improving the accuracy of the cache mechanism. Another approach might be including more data mining related algorithms to analyze the historical result, so that the cached results could be utilized in a more efficient way. The sensor network part of the current NOSA system has many disadvantages considering the consumption of sensor nodes power and sensor connection bandwidth, in-network aggregation is a possible solution that the raw data sensing by the nodes should be processed before passing to the base station [12] And as demonstrated in the related work section, the query analysis model at the sensor nodes, which allows the node to decide when the query stored in the cache is expired and the cache mechanism requires new data to update its parameters, is also a possible solution to further reduce the number of messages passing over the sensor network.

References

[1]. Chu X, Kobialka T, Durnota B, and Buyya R. "Open Sensor Web Architecture: Core Services". In Proceedings of the 4th International Conference on Intelligent Sensing and Information Processing (ICISIP 2006, IEEE Press, Piscataway, New Jersey, USA, ISBN 1-4244-0611-0) pp. 98-103, Dec. 15-18, 2006, Bangalore, India.

[2]. Tham C, and Buyya R, "SensorGrid: Integrating Sensor Networks and grid Computing", Technical Report, GRIDS-TR_2005-10, Grid Computing and Distributed Systems Laboratory, University of Melbourne, Australia, June 24, 2005

[3]. Kobialka T, Buyya R, and Leckie C, Open Sensor Web Architecture: Stateful Web Services, Technical Report, GRIDS-TR-2007-13, Grid Computing and Distributed Systems Laboratory, The University of Melbourne, Australia, July 13, 2007.

[4]. Botts M, Percivall G, Reed C, Davidson J, OGC® Sensor Web Enablement: Overview and High Level Architecture, OpenGIS Consortium Inc, 2006

[5]. Open Geospatial Consortium, Inc
URL: http://www.opengeospatial.org/
Accessed [4, Nov]

[6]. The Globus Alliance
URL: http://www.globus.org/
Accessed [4, Nov]

[7]. Simonis I, Sensor Planning Service OGC 05-089r1, Open GIS Consortium Inc, 2005

[8]. Simonis I, Wytzisk A, Web Notification Service OGC 03-008r2, Open GIS Consortium Inc, 2003

[9]. OGC Implementation Specification Download Agreement
URL:
http://portal.opengeospatial.org/modules/admin/license_agreement.php?suppressHeaders=0&access_license_id=3&target=http://portal.opengeospatial.org/files/index.php?artifact_id=14034
Accessed [5,Nov]

[10].TinyOS Alliance
URL: http://www.tinyos.net
Accessed [5,Nov]

[11] Sam Madden, TinyDB
URL: http://telegraph.cs.berkeley.edu/tinydb
Accessed [5, Nov]

[12] Feng Z and Leonidas G, Wireless Sensor Network, Morgan Kaufmann Publishers, San Francisco, CA, 2004

[13] Deshpande A, Guestrin C, Madden S, Hellerstein J, and Hong W, Model-based Approximate Querying in Sensor Networks, Proceedings of VLDB Journal , 2005

[14] Ping X., Chrysanthis P.K. and Labrinidis A., Similarity-Aware Query Processing in Sensor Networks, Parallel and Distributed Processing Symposium, IPDPS 2006, April 25-29, 2006

[15] Wei Y, Thang N L, Jangwon L, Dong X, Effective query aggregation for data services in sensor networks, Computer Communications archive Volume 29, Issue 18, November 2006

[16] Amol D, Suman N, Phillip B. G, Srinivasan S, Cache-and-query for wide area sensor databases

International Conference on Management of Data, Proceedings of the 2003 ACM SIGMOD international conference on Management of data, San Diego, California 2003

[17] David C, Amol D, Joseph M. H and Hong W, Approximate Data Collection in Sensor Networks using Probabilistic Models, Proceedings of the 22nd International Conference on Data Engineering (ICDE'06), Volume 00, page 48, 2006

Appendix I. NOSA Cache Project Configuration Manual

Packages Organization

The hierarchy is organized as follows:

org.sensorweb.core.scs.cache contains all important classes like CacheManager and RuleEngine and all interfaces.

org.sensorweb.core.scs.cache.impl contains the concrete classes for those interfaces and abstract class.

There are some other classes lying around other packages, which are listed below

org.sensorweb.demo.report.StopWatch: A class to calculate the time cost.

org.sensorweb.demo.summer: This package also contains some new added utilities class.

org.sensorweb.service.SensorLockManager: A class to handle the locking mechanism.

Configurations

Two files are needed in order to make service runs. The first one is "application.properties", which is presented in the original project. Some place needs to be changed slightly. The second one is "cache.conf", which is new added in this project. Details of two files are shown below:

application.properties:

We need to add following lines to "application.properties":

```
##########cache###########
cache.comparer.EnhancedComparer=org.sensorweb.co
re.scs.cache.impl.ComparerImpl
cache.discard.OrderParser=org.sensorweb.core.scs
.cache.impl.DiscardParserImpl
cache.Memory=org.sensorweb.core.scs.cache.impl.M
emoryCache
cache.DB=org.sensorweb.core.scs.cache.impl.DBCac
he
```

Lines start with "#" are comments, just add whatever you want. Other lines are the configuration line. String on the left of “=” is the reference name of class. String on the right of “=” is the QName of the class. Reference name of class is used in the “cache.conf” file. Just use whatever you like to name the class. QName is the full path which leads to the class, including the class’s name.

Please make sure that all the classes’ names used in the “cache.conf” file is presented and well specified here.

cache.conf:

The following are the default settings of "cache.conf"

```
########## RuleEngine Configuration ############
#it indicates whether to use RuleEngine
#available values are "true" and "false"
useRuleEngine=true

#estimate. estimate controls whether to access sensor network or not.
#this is the intialized estimate for RuleEngine
#it is in seconds, for examples, 3 means 3 seconds
#estimate=3

#similar threshold controls the gap between request query and cache
query
#please refer to the documentation
#threshold=1
########## Memory Cache Configuration ##########
#cache size is the maximum size of a cache
#default value is 50, uncomment to specify your value
#mem.maxCacheSize=50

#how long can an entry life in a cache
#in seconds
#default is 60 seconds
#mem.maxEntryLifeTime=60

#discard policy tells in what order the entries are replaced
#currently two are available, "rank;lifetime" means least used will
be replaced first
#while "lifetime;rank" means oldest will be replaced first
#please notice that there is not default value, make sure you specify
exactly one
#mem.discardPolicy=rank;lifetime
mem.discardPolicy=lifetime;rank
#implemention: implemention class to ensure the policy by using the
above string
mem.discardParserClass=cache.discard.OrderParser
```

```
#It defines how frequently the CacheMainter to shrink the cache
#default is 5 seconds (they are in seconds)
#mem.shrinkInterval=5

#how many entries are shrinked at a time if no expired entries found
#default is 3
#mem.shrinkSize=3

#the compare class for memory cache
mem.comparerClass=cache.comparer.EnhancedComparer

########## Database Cache Configuration ##########

#most of the database cache parameters are the same as the memory
cache
#please refer to the memory cache comments

#db.maxCacheSize=50

#db.maxEntryLifeTime=60

#policy
#db.discardPolicy=rank;lifetime

#policy implemention
db.discardParserClass=cache.discard.OrderParser

#shrink interval
#db.shrinkInterval=5

#how many entries are shrinked
#db.shrinkSize=3

#the interval time between refresh used to fectch cached queries
#database cache need to refresh its entry list. this is the interval
time.
refreshInteval=0.5
```

```
#db.comparerClass=cache.comparer.SimpleComparer
db.comparerClass=cache.comparer.EnhancedComparer

#this is the connection string. it should be standard jdbc connection
string
db.connectionString=jdbc:postgresql://localhost:5432/NOSA
#driver name for the database
db.driverName=org.postgresql.Driver
#name of the table
db.tableName=cache
#user name
db.userName=your_name
#password
db.password=your_password

########## CacheManager Configuration ##########

#use cache chain to chain to memory and database cache
#when not cache.chain is specified here, no cache is used
#for example, cache.Memory;cache.DB means memory cache is the first
level
#while database cache is second level.
#cache.chain=cache.Memory;cache.DB
#cache.chain=cache.Memory
#cache.chain=cache.DB

########## Performance Capture Configuration ###########
#this is the filename where performance of query is output
#filename should in such format where "\" should be written as "\\"
#filename=e:\\result3-cache
#surfix is the surfix of the file
#surfix=txt
```

most of the parameters are explained in the comments of the file. There's only one thing to be careful. Parameters start with prefix. For examples, memory cache starts with "mem" while database cache start with "db". Prefix is defined inside the class. Please refer to the class documentation.

There are several parts to config, but most of them are very easy.

1. Cache:

   Cache has to specify in the “cache.conf” file. First, set the cache chain under field “cache.chain”. Second, setup the parameters with prefix of that cache.

2. RuleEngine:

   There are three parameters to set. “useRuleEngine” determine whether to use RuleEngine or not. “estimate” is the initial value of estimate. Similarly, “threshold” is the initial threshold.

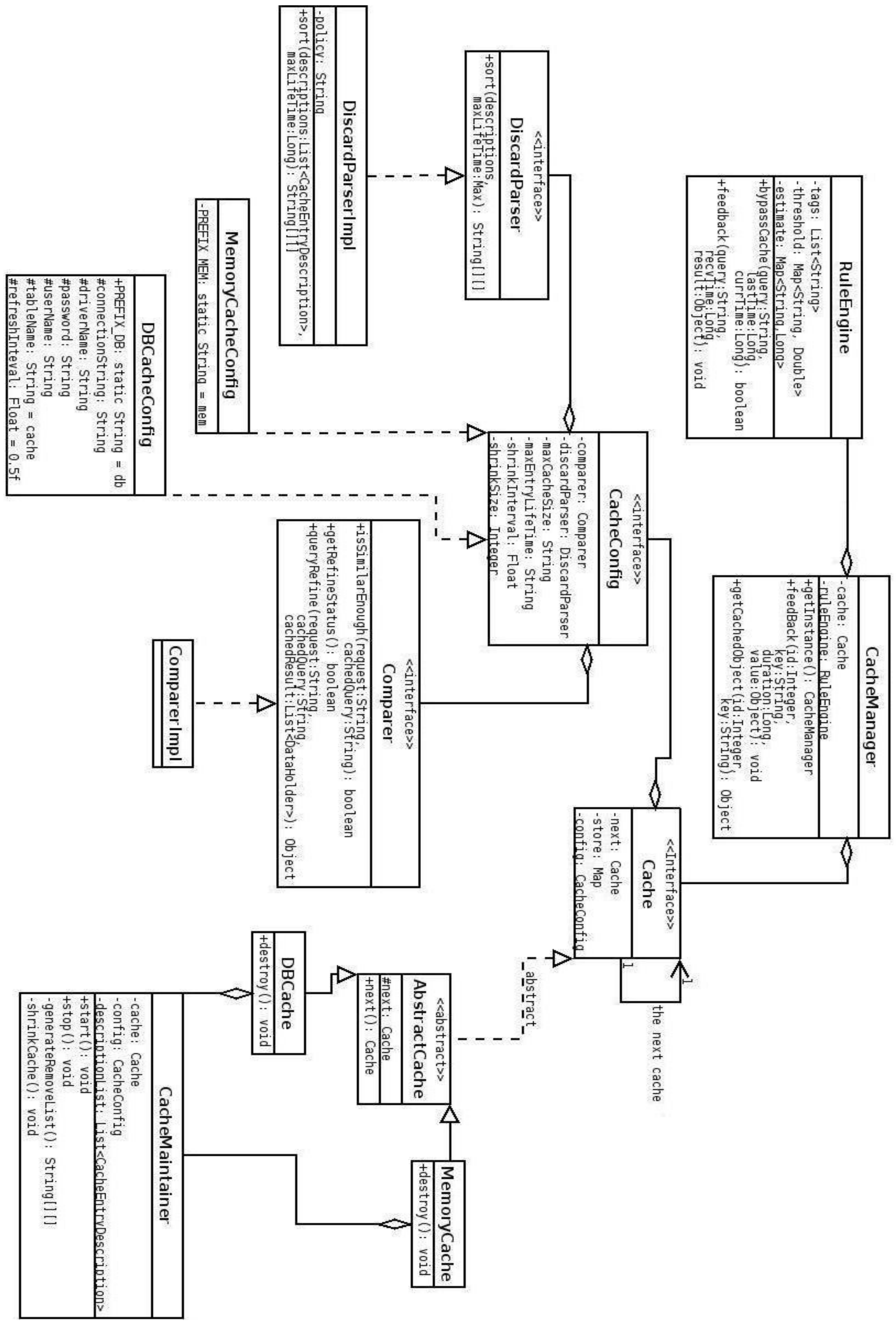
3. CacheManager:

   A parameter name “cache.chain” specifies what does the cache chain looks like.

4. Performance:

   This part set the output file’s name for performance of cache.

Appendix II. Full Classes Diagram


RuleEngine
-tags: List<String>
-threshold: Map<String, Double>
-estimate: Map<String,Long>
+bypassCache(query:String, lastTime:Long, currTime:Long): boolean
+feedback(query:String, recvTime:Long, result:Object): void
<<interface>>
DiscardParser
+sort(descriptions, maxLifeTime:Max): String[][]
DiscardParserImpl
-policy: String
+sort(descriptions:List<CacheEntryDescription>, maxLifeTime:Long): String[][]
MemoryCacheConfig
-PREFIX_MEM: static String = mem
DBCacheConfig
+PREFIX_DB: static String = db
#connectionString: String
#driverName: String
#password: String
#userName: String
#tableName: String = cache
#refreshInteval: Float = 0.5f
<<interface>>
CacheConfig
-comparer: Comparer
-discardParser: DiscardParser
-maxCacheSize: String
-maxEntryLifeTime: String
-shrinkInterval: Float
-shrinkSize: Integer
CacheManager
-cache: Cache
-ruleEngine: RuleEngine
+getInstance(): CacheManager
+feedBack(id:Integer, key:String, duration:Long, value:Object): void
+getCachedObject(id:Integer, key:String): Object
<<interface>>
Comparer
+isSimilarEnough(request:String, cachedQuery:String): boolean
+getRefineStatus(): boolean
+queryRefine(request:String, cachedQuery:String, cachedResult:List<DataHolder>): Object
ComparerImpl
<<Interface>>
Cache
-next: Cache
-store: Map
-config: CacheConfig
1
1
the next cache
abstract
<<abstract>>
AbstractCache
#next: Cache
+next(): Cache
DBCache
+destroy(): void
MemoryCache
+destroy(): void
CacheMaintainer
-cache: Cache
-config: CacheConfig
-descriptionList: List<CacheEntryDescription>
+start(): void
+stop(): void
-generateRemoveList(): String[][]
-shrinkCache(): void

This diagram shows all the classes that are newly created by this project. From beginning, we can first focus on the interface. Cache is a cache holder, which contains a CacheConfig. CacheConfig holds DiscardParser and Comparer. These three classes makes up the major part of cache facility. RuleEngine and CacheManager are singleton, while CacheMaintainer resides in each Cache for cleaning up the expired cache entries.