A Reusable Component-Based Model for WSN Storage Simulation

Marcos Aurélio Carrero
Universidade Federal do Paraná, Brazil
macarrero@inf.ufpr.br

Martin Alejandro Musicante
Universidade Federal do Rio Grande do Norte, Brazil
mam@dimap.ufrn.br

Aldri Luiz dos Santos
Universidade Federal do Paraná, Brazil
aldri@inf.ufpr.br

Carmem Satie Hara
Universidade Federal do Paraná, Brazil
carmem@inf.ufpr.br

ABSTRACT

Sensor networks are a fast-evolving technology used for a variety of applications, ranging from environmental monitoring to cyber-physical systems (CPS) and IoT, including applications designed to support smart cities. The widespread use of sensor networks rises new challenges to data management and storage. The development of data storage systems is a hard task due to the specific nature of wireless sensor networks (WSNs) and the lack of a common general purpose development framework. Software component models provide an appropriate level of system abstraction, reducing the development complexity and improving productivity. In this paper we propose RCBM, a Reusable Component-based Model for wireless sensor network storage simulation. RCBM promotes software reuse from existing components to improve the efficiency of system development and evaluation. RCBM has been implemented on the NS2 simulator and experimental results show that RCBM is more flexible than previous component-based models for WSNs. Due to its general-purpose approach, RCBM can be applied to develop simulation code for a wide range of WSN storage models, reducing the development effort.

KEYWORDS

WSN Storage; Component Library; Code Reuse

1 INTRODUCTION

Wireless sensor networks (WSNs) have evolved from small scale deployments to a wide range of large-scale networks. They have become the technological infrastructure for supporting a variety of applications that demand sensing data, from environmental monitoring to cyber-physical systems (CPS) and IoT, designed to support smart cities [15, 18]. Due to the heterogeneous nature of environmental information found at several urban areas, sensing such environments requires the deployment of dense WSNs [22]. As a consequence, the increase on the volume of sensing data requires autonomous and scalable WSNs, rising new challenges to data management and storage models development.

Distributed data repositories and clustering are common approaches to tackle the scalability of data storage problem. Although recent efforts have been made to build efficient data storage systems, the specific nature of WSNs and the lack of a common general purpose development framework make the design of these applications a hard task [21]. In general, systems are implemented from scratch, to meet the requirements of specific domains, or integrated in complex applications, which make separation of concepts and code reuse very limited [19]. High-level application modeling allows developers of WSN systems, such as in-network query processing[5, 13], to abstract from lower-level details, reducing the development complexity and improving the productivity.

System software modeling approaches have provided an appropriate level of abstraction in the development of distributed systems. In the context of WSN storage systems, software component models allow developers to build the system from reusable existing components based on functional specifications i.e. *interfaces* and interaction patterns techniques [8]. Some component models, such as OSGi and OpenCOM, describe the *interfaces* in terms of a set of operations (functions). Other models follow a port-based approach interface, providing the components with entries for receiving and sending data. Common interaction approaches used for remote component communication include pipe and filter, request-response, publish-subscribe and broadcast. Component modeling has been used in different application domains, such as automotive software [17], CPS [20], WSNs simulation [7] and IoT [25]. In WSNs simulation environments, component-based models are a promising approach to manage large-scale application scenarios.

Inspired by these challenges, some works have emerged in order to improve system development productivity and code reuse. Most of them stress the importance of identifying common concepts on the target application, such as data processing, network [26, 27], and IoT [24]. To the best of our knowledge, the only component-based model specific for WSN in-network storage has been proposed by CBCWSN [2].

In this paper we propose the *Reusable Component-Based Model* for WSN Storage Simulation (RCBM). RCBM focuses on storage-level entities, defining a set of common concepts and functionalities that represent various instances of WSN storage systems. RCBM allows developers to reuse components across system implementations, reducing the complexity to design and implement new systems. It differs from CBCWSN on how the components interact

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Q2SWinet'17, November 21–25, 2017, Miami, FL, USA
© 2017 ACM. ISBN 978-1-4503-5165-2/17/11…$15.00.
DOI: http://dx.doi.org/10.1145/3132114.3132118

with each other. While CBCWSN has a rigid flow of execution, RCBM adopts a flexible execution model, based on a coordinator, which is responsible for message exchanges. This model is in accordance with network simulator NS2, on which RCBM has been implemented. We have conducted three case studies, implementing the LCA [4], LEACH [16] and MAX-MIN[3] systems with RCBM. The results show that for these models RCBM provided at least 66% of code reuse. Our studies show that for MAX-MIN, the reuse with RCBM reaches 66% of the code, while with CBCWSN it is as low as 23%.

The remainder of this paper is organized as follows. Section 2 discusses the related work. Section 3 presents RCBM. The case studies are detailed in Section 4. Our experimental study is reported in Section 5, and we conclude in Section 6.

2 RELATED WORK

The use of software components to facilitate the implementation of families of systems that share a common structure is a recurring idea in computer science. The idea of building systems by using *off-the-shelf* control and data objects is at the core of many modern programming concepts and paradigms [14] and even broad areas inside software engineering such as Software Product Lines [6], Aspect-Oriented Software Development [12] or Service-Oriented Architectures [11].

Owing to the nature of distributed systems and networks, there is a large number of successful attempts to use components in distributed applications development. A seminal work in this area is [23], where a *Software Component Architecture* is more formally defined and some applications and examples are given. The use of components is usually associated to the existence of a *library* of component templates containing (possibly incomplete) code. The contents of the library can be transformed into components, by the addition of user-defined portions of code, such as data declarations and commands.

Network simulation tools are useful to test and compare systems before real-world deployment, providing insights about systems behavior. In WSNs simulation environments, components models are a promising approach to manage the increase demanding for scalability and autonomy requirements. The RCBM modular design, inspired in the object-oriented programming encapsulation and design patterns principles for creating reusable components, can be implemented in any of well known simulators, such as NS2, NS3 and OMNeT++.

SenNet [26], MDDWSN [27], CBCWSN [2] and IoTSuite [24] apply separation of design concepts, focusing on a specific aspect of the system under construction. MDDWSN and SenNet propose a software development process to address data processing-related and network-related concepts. These tools, however, do not support the concept of distributed data repositories and are implemented under TinyOS operating system, which limits their applicability to small WSN networks. IoTSuite, on the other hand, proposes a common understanding of concepts that constitute the IoT. It provides a centralized storage-related concept, which increases communication overhead and is less scalable than decentralized approaches. CBCWSN identified the key components to be reused in the majority of clustering algorithms for WSNs. However, as opposed to RCBM, CBCWSN adopts a predefined flow of execution. As a result, it is harder to extend CBCWSN to other storage models that do not adhere to its interaction pattern.

3 THE DATA STORAGE COMPONENT MODEL

There is a plethora of storage models for wireless sensor networks proposed in the literature [10]. Although they differ on the target application and details to tailor the model to specificities of the application, some concepts (or entities) are common to a vast majority of them. In this section, we propose the *Reusable Component-Based Model* for WSN Storage Simulation (RCBM). It is a general storage metamodel for WSNs, described by entities, properties, and functions. The model is devised to include a library of “instances” of the metamodel. As a result, new models that are specific to certain applications can be created by developing a few functions instead of considering the problem of storing sensing data as a whole. In this way, RCBM promotes *code reusability* and supports the *agile development* of new models.

In order to determine the common entities among existing models, we first identified three storage classes, based on the location where sensing data are stored. The classes, as shown in Figure 1(a) are: *Local*, referring to models that store data locally in the sensors' flash memory; *External*, consisting of models that store all data collected by sensor devices at an external storage; and *Repositories*, which include models where some sensors are responsible for collecting the data of a set of sensors. In each of the classes it is possible to identify common components to create an RCBM metamodel. In dense WSNs, keeping the data grouped into repositories reduces the number of data transmissions for querying the data [9], an important requirement to reach scalability. Among the storage models described in our taxonomy, the *repositories* class has attracted a lot of attention, given that it is more scalable than other approaches. Thus, this work focuses on developing components to models in the *repositories* class, which is the most appropriate for large-scale networks. It is important to point out that the proposed component-based approach can be extended to consider other storage classes, as long as their entities are properly modeled.

In RCBM, each entity is associated with a set of components that implement common functionalities. For models in the repository class, these entities include: *(i)* sensor devices; *(ii)* clusters, which consist of a set of sensors; and *(iii)* cluster-heads (CHs) that are sensors responsible for storing the information of all cluster members. These entities define a hierarchical storage model, where each cluster designates a sensor as cluster-head for storing the readings of its group members. This approach share some common tasks, as described in [2]. The first one is related to the cluster-head election strategy, that can be a random choice or based on a sensor attribute. The second task refers to cluster membership; that is, how nodes that were not elected as cluster-heads decide to which cluster they should join. Common criteria include random choice, location-based and attribute-based decisions.

In a component-based design, components are reusable units, and the development of components is kept separate from the system development [8]. This idea is illustrated in Figure 1, showing that existing components can be combined and restructured in order to develop new models, depicted in the figure as model instances.

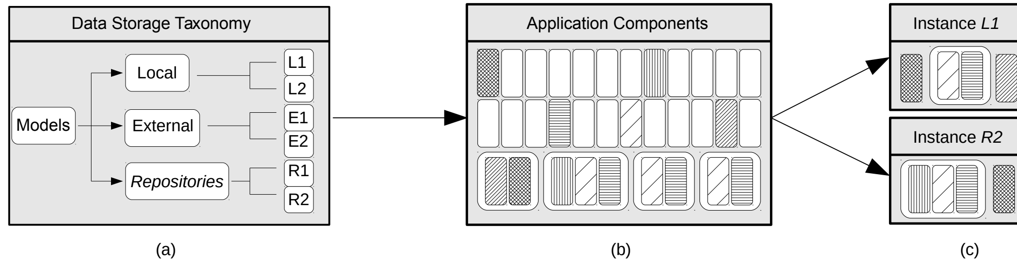

Figure 1: Component-based Development Process for WSNs Storage Models.

Besides the *Application* components, that are directly related to WSN storage models, in RCBM we have two other types of components: *Coordination* and *Library*. The first one is responsible for coordinating the execution flow and interaction between WSN components. The second type provides additional functionality, common to several models. Thus, three types of components can be distinguished in RCBM:

- *Application Components* refer to entities that compose the storage model. Each of them provides a set of functions that are invoked by other components and the coordinator in order to compose the overall system functionality.
- A *Coordinator Component* controls the execution flow of the system.
- *Library Components* implement functions that are orthogonal to storage models, but provide a toolbox that is useful for developing application components. Examples of library components include aggregation and timer functions.

Syntactically, each component is defined by two documents: component template and component implementation. Component templates consist of a set of operation descriptions and their input/output parameters, thus, defining the interface of the component. Particular implementations may be defined for each template. Separation of interface and implementation is of major importance in order to create a pluggable set of components, which can promote reusability in several ways. It allows library components to be used in the development of any application component. Moreover, the coordinator can be reused to apply the same execution flow to different implementations of application components. In addition, the same application components can be reused in a different execution flow by a new implementation of the coordinator.

3.1 The RCBM Platform

RCBM is designed to promote reusability by considering an architecture composed of three layers: the specification layer, the implementation layer, and the communication platform. The architecture of RCBM is depicted in Figure 2. The specification layer consists of components that can be used "as-is" (such as library components) as well as the templates (interfaces) of the application and coordinator components. The implementation layer consists of actual code for the application and coordinator components, in order to develop storage model instances. The separation between specification and implementation facilitates the development, promoting the composition of new components with previously developed ones. The communication platform provides the basic infrastructure for implementing communication among components.

In the development process, the programmer starts with a set of function libraries and component templates that can be used to build the desired model. During the implementation, which comprises the combination of pluggable pieces and the coordinator, templates are transformed into components, by the addition of user-defined portions of code (component implementations). The application components associate functionalities to WSN entities, such as clusters and CHs, while the coordinator controls the execution flow among them. At the lower level, the communication platform provides the simulation runtime support. Furthermore, RCBM provides a code skeleton that integrates the coordinator with the communication platform. Thus, the user does not have to take care of communication platform details.

In the following sections we present details of library components, application templates and application components.

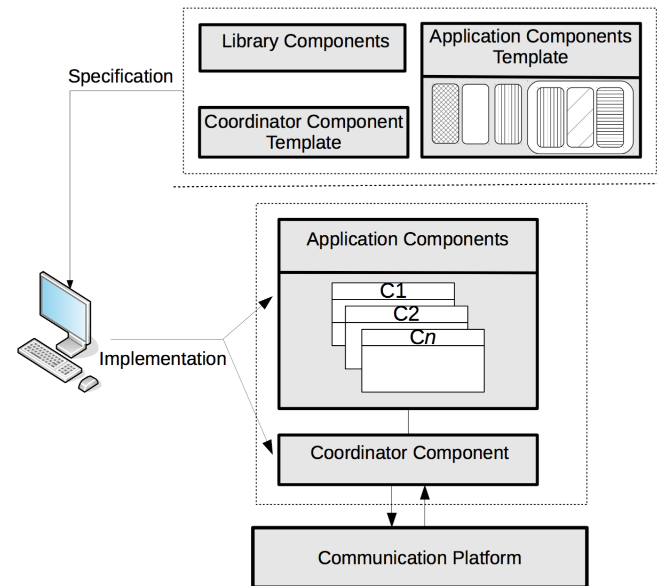

Figure 2: RCBM Architecture.

3.2 Library Components

Library components are not directly related to application entities, but provide useful functions that can be used in the development of other components. In this section, we present two of them: aggregate functions and timer components.

Aggregate functions are executed on a set of values, in order to obtain a single, representative one such as the maximum value (MAX), minimum value (MIN), average (AVG), summation (SUM), and set cardinality (COUNT). Aggregate functions are useful in a number of operations for WSNs. As an example, in cluster-based storage models, cluster-heads (CHs) are the only sensors contacted to process in-network queries. Usually, CHs do not report the individual readings of all cluster members, but provide a single representative value, which can be obtained by an aggregate function. The goal of this approach is to minimize the volume of traffic and number of transmissions in the network. Listing 1 illustrates the aggregate functions component specification that defines the MIN, MAX, AVG and SUM aggregation operations.

```
1  class Library_Aggregation {
2  public:
3    template <typename K, typename V>
4      map<K, V> MIN(map<K, V>);
5
6    template <typename K, typename V>
7      map<K, V> MAX(map<K, V>);
8
9    template <typename K, typename V>
10     V AVG(map<K, V>);
11
12   template <typename K, typename V>
13     V SUM(map<K, V>);
14 }
```

Listing 1: The library component aggregation template.

Notice that function declarations use C++ template syntax, and thus they can handle different types of data. In our approach, all aggregate functions operate on a set of key-value pairs "map<K, V>". This representation is quite general. For instance, suppose that a CH receives the reading 10.2 from a cluster member s_2. Then, this information can be represented as the key-value pair $(s_2, 10.2)$, where s_2 is the key K, that is, the sensor unique identifier, and 10.2 is its associated value V. Given a set of key-value pairs of cluster members readings, $\{(s_2, 10.2), (s_3, 10.5), (s_4, 10.7)\}$, the CH can obtain their average applying the AVG function, which returns the value 10.47. MIN and MAX functions return a key-value pair instead of a value. The result for MIN is $(s_2, 10.2)$, while the result for MAX is $(s_4, 10.7)$.

```
1  class Timer : public TimerHandler {
2  public:
3    virtual void expire(Event *e);
4    template <class T> void initTimer(T);
5    template <class T> void restartTimer(T);
6  }
```

Listing 2: The library component timer template.

Another useful component for WSNs is the timer. It provides functions for synchronization purposes, including message transmissions and sensoring data tasks. Listing 2 illustrates the Timer component template specification.

The timer library provides functions to initialize the timer "initTimer(T)", restart the timer "restartTimer(T)", and expire at T delay seconds. When the timer expires, function "expire(Event *e)" is automatically invoked. As a consequence, the user does not need to check manually for the timer's expiration.

3.3 The Application Components Template

In contrast to library components, application components are related to functionalities of entities in the WSN. Moreover, library functions provide useful operations that programmers can use without knowing any of the internal details. On the other hand, different WSN storage models may involve specific steps, and thus the programmer has to modify or implement some of the application components, associated with entities of the model. Since our focus is on models in the repository class, these entities are: the sensor device, clusters, and CHs. Due to space limitations, only some of the components are discussed.

For sensor devices, the main characteristic presented in autonomous and scalable systems relies on following a distributed solution instead of a centralized approach. In distributed architectures, sensors have only a local knowledge about the network. For example, some solutions require that each sensor s_i maintains information about its neighbors $N(i)$, such as, their identifier, geographic coordinates, remaining energy, CH election announcements and join cluster announcements. Listing 3 illustrates the Sensor component specification that provides some of the functionalities for sensor devices.

```
1  class Sensor {
2  public:
3    template <typename K, typename V>
4    void setNeighborReadings(map<K, V>);
5
6    template <typename K, typename V>
7    map<K, V> getNeighborReadings();
8
9    template <typename K, typename V>
10   map<K, V> getCandidateCHs();
11 }
```

Listing 3: The sensor component template.

```
1  class CH_Election {
2  public:
3    template <typename K, typename V>
4    void selectCH(map<K, V>);
```

Listing 4: The CH_Election component template.

For the cluster entity, we can identify two main phases in clustering algorithms: CH election and cluster formation. Thus, we have two application components: CH_Election and Cluster_Formation, which are illustrated in Listings 4 and 5, respectively. CH election algorithms strategies can be classified into three categories: preset, random and attribute-based [1]. The template provided by the

CH_Election component is generic enough to implement all of these CH selection criteria.

The CH_Election component defines the function "selectCH(map<K, V>)" that users must implement according to the target model. Note that the function operates on a set of key-value pairs (map<K, V>). As an example, consider the probabilistic model where a sensor becomes a CH with probability p. Intuitively, each sensor s_i executes $selectCH(s_i, p_i)$, where $K = s_i$ and $V = p_i$. The template provides useful information to programmers, along with the argument specification. The programmer has to extend the CH_Election component template to implement the CH selection task required by the storage model.

```
1  class Cluster_Formation {
2  public:
3    template <typename K, typename V>
4    void join(map<K, V> );
```

Listing 5: The Cluster_Formation component template.

Once CHs are elected, they announce the decision to the network. A node decides to join a cluster based on a classification criterion. The Cluster_Formation component contains the "join(map<K, V>)" function, which takes as input a set of key-value pairs (map<K, V>) of CH announcements. Suppose that the join criterion is that every sensor joins the cluster which has a CH with the smallest identifier (ID). Then, in each sensor the function "join" takes as input the set of its candidate cluster-heads, for example $\{(s_j, 3), (s_k, 5)\}$, where s_j, s_k are sensors with IDs 3 and 5, respectively. The implementation of the function should choose to join the cluster with CH s_j, given that it has the lowest ID among the CH candidates.

The communication among components in the system is defined by the Coordinator component. It is responsible for the overall execution flow. Listing 6 illustrates the simplified coordinator component template declaration. Recall that our focus is on simulation environment tools. In RCBM, every program has a function "startSimulation()", which is called when the simulation starts. During the execution of the simulation, communication between remote components (and sensors) is achieved through asynchronous message passing. Thus, functions "sendPkt" and "recv" defined in the coordinator component provide such functionality.

```
1  class Coordinator {
2  public:
3    void startSimulation();
4    virtual void recv (Packet*, Handler*);
5    void sendPkt(MsgID, WSN_Components_Message*);
6  }
```

Listing 6: The Coordinator component template.

4 RCBM IMPLEMENTATION LAYER

This section presents implementation details showing how RCBM was used to develop two storage models in the repositories class: LCA and LEACH. These case studies show that reusing library components and templates improve the development productivity.

4.1 LCA Components Implementation

LCA (Linked Cluster Algorithm) [4] is a hierarchical storage model intended to be used for small networks. The criterion for clustering is based on the unique identifier (ID) associated with each sensor node. During the CH election phase, LCA elects as CH a node with the lowest ID among its neighbors that not received a CH announcement. After the election phase, remaining sensors join the cluster of the closest CH. Our component model can be used to implement LCA by extending the "CH_Election" and "Cluster_Formation" templates.

The CH_Election Component. Listing 7 illustrates the simplified implementation of the "selectCH" function for LCA.

```
1  template <class K, class V>
2  void CH_Election::selectCH(map<K, V> Neighbors) {
3   int minNeighbors = compLib->MIN(Neighbors);
4
5   if ( getSensorId() < minNeighbors ) {
6    role = CH;
7    WSN_Components_Message param();
8    param.setId(getSensorId());
9    param.setDestination(Broadcast);
10   sendPkt(CH_ANNOUNCE, &param);
11  }
12  else
13   role = CM;
14 }
```

Listing 7: The CH_Election component implementation.

The developer has to provide the implementation of the "selectCH" function according to the model criterion. The "selectCH" function operates on a set of key-value pairs "Neighbors", that is, the s_i's neighbors that have not been assigned to a CH *(l.2)*. Initially, MIN(Neighbors) *(l. 3)* library function computes the minimum *ID* among s_i's neighbors. A sensor is selected as CH when its ID, given by "getSensorId()", is less than the calculated "minNeighbors" *(l. 5)*. It then broadcasts its role as CH to the network *(l. 7-10)*. Otherwise, the sensor role is set as a cluster member (*CM*) *(l. 13)*.

The Cluster_Formation Component. In the next step, nodes that were not selected as CHs must join a cluster, as illustrated in Listing 8. A node chooses as CH the sensor with the minimum ID among its neighbors that are CHs *(l.3)*. Then, it sends an ACK message to the chosen one *(l.5-9)*.

```
1  template <class K, class V>
2  void Cluster_Formation::join(map<K, V> knownCHs) {
3      int minCH = MIN(knownCHs);
4
5      WSN_Components_Message param();
6      param.setId(getSensorId());
7      param.setDestination(minCH);
8
9      sendPkt(ACK_CH_ANNOUNCE, &param);
10 }
```

Listing 8: Cluster_Formation component implementation.

The Coordinator Component. The coordinator is responsible to glue components together, coordinating the interactions among them. The main tasks involve control of the input/output message

buffer and coordination of the execution flow. Listing 9 depicts the implementation of the main functions.

```
void Coordinator::startSimulation(double T) {
     setCurrentRound(SELECT_CH);
     initTimer(T);
}

void Coordinator::recv(Packet* pkt, Handler *) {
   switch(param.getMsgId()) {
      case (CH_ANNOUNCE):
           manageCHAnnounce(&param);  break;
      case (ACK_CH_ANNOUNCE):
           manageACKCHAnnounce(&param);  break;
}

void Coordinator::sendPkt(MsgID ID, MsgParam* P) {
   switch (ID) {
      case (CH_ANNOUNCE):
      case (ACK_CH_ANNOUNCE):
            send(P->getPkt(), 0);  break;
}

void Timer::expire(Event*) {
      if (getCurrentRound() == SELECT_CH) {
           param=getCHParams();
           compCHElection->selectCH(&param);
           setCurrentRound(JOIN_CLUSTER);
      } else if (getCurrentRound() == JOIN_CLUSTER) {
           param=getJoinParams();
           compClusterFormation->join(&param);
           setCurrentRound(FINISH);
      }
}
```

Listing 9: Coordination component implementation.

In LCA, data transmissions follow a round-robin schedule, from the lowest ID to the highest ID. Thus, each node configures its time schedule and the current round at the beginning of LCA simulation *(lines 2-3)*. When the timer expires *(line 21)*, each node executes the tasks for the current round. During the first round, each node calls the "selectCH()" function, which implements the election strategy, and waits for some time units to start the next round *(lines 23-25)*. When the timer expires, each node decides to join the cluster, executing the "join" function of component Cluster_Formation *(lines 27-29)*.

4.2 LEACH Components Implementation

LEACH (Low-Energy Adaptive Clustering Hierarchy) [16] is a probabilistic model that forms one-hop clusters. LEACH assumes that all nodes are within the communication range of each other. Sensors elect themselves as cluster-heads with a probability p. The remaining sensors join the cluster of the CH that requires the lowest energy consumption to communicate.

The CH_Election Component. Listing 10 depicts the simplified implementation of the "CH_Election" Component.

```
template <typename T>
void CH_Election::selectCH(map<K, V> P) {
   double r = ((double) rand() / (RAND_MAX));
   prob = getProbability(P);
   double threshold = prob / ((1.0 - prob) * fmod(1.0, (1 / prob)));

   WSN_Components_Message param();
   param.setId(getSensorId());

   if (r < threshold)
     sendPkt(CH_ANNOUNCE, &param);
}
```

Listing 10: CH_Election component implementation.

The CH component implements function "selectCH" that takes a probability P as an input parameter *(line 2)*. First, each node selects a random floating point number between 0 and 1 *(line 3)* and calculates a threshold based on the given probability *(line 5)*. A node selects itself as a CH when the calculated number is less than the threshold, and broadcasts the decision to the network *(lines 10-11)*.

The Cluster_Formation Component. In the next step, nodes that were not selected as CHs must join a cluster. Each node chooses as its CH the sensor that requires the minimum communication energy, based on the received signal strength (RSS) of the CH advertisements. Listing 11 illustrates the simplified Cluster_Formation component implementation.

```
void Cluster_Formation::Join(map<K, V> knownRSS){
     double maxRSS = MAX(knownRSS);

     WSN_Components_Message param();
     param.setId(getSensorId());
     param.setDestination(maxRSS);
     if (role == CM)
       sendPkt(ACK_CH_ANNOUNCE, &param);
}
```

Listing 11: Cluster_Formation component implementation.

During the cluster formation phase, sensor s_i executes the "Join" function, taking as input a key-value pair "knownRSS", that represents a set of CHs announcements received by s_i. First, each node computes the maximum received signal strength (RSS) from "knownRSS". The largest RSS is the CH that requires the minimum amount of energy to communicate with. Thus, s_i sets as CH the sensor with the maximum RSS recorded by (knownRSS) *(line 2)* and sends an ACK message *(lines 4-8)*. The LEACH and LCA coordinator components share many similarities. Thus, due to space limitations, the details of the LEACH coordinator will be omitted. The two case studies presented in this section show that the model instances share the same components templates. This approach promotes reusability, and the programmer develops a few lines of code with the specificities of each model.

We have also implemented a third model, MAX-MIN [3], which is used in our experimental study[1].

5 EXPERIMENTAL STUDY

This section shows an empirical performance evaluation of RCBM. We have applied our approach to support the implementation of LCA, LEACH and MAX-MIN protocols on NS2 network simulator version 2.35. LCA and MAX-MIN clustering algorithms follow an attribute-based clustering criterion. LEACH follows a probabilistic

[1]The implementation is available at http://www.inf.ufpr.br/macarrero/Q2SWinet.

model. We have conducted two experiments. In the first one, we analyzed the amount of code reuse across the systems and compare these results with the amount reported by the CBCWSN system, reported in [2]. In the second experiment, we validate the correctness of our MAX-MIN implementation by applying it to the same evaluation scenario and parameters reported in [3]. MAX-MIN is an attribute-based model that differs from the ones reported in Section 4 because cluster members can be multiple hops away from the cluster-head. Moreover, the clustering algorithm involves two communication steps among sensor nodes before the election of CHs. More specifically, the algorithm has four logical steps: *(i)* propagation of larger node IDs, *(ii)* propagation of smaller node IDs, *(iii)* determination of CHs, and *(iv)* linking of clusters. Although the execution flow of the MAX-MIN algorithm differs from LCA and LEACH, some of the steps closely resemble the steps in these algorithms. The separation of the coordinator from the application components proposed by RCBM allowed us to explore this similarity for code reuse. The results of our experiments are presented next.

5.1 Code Reuse

The goal of this experiment is to determine the impact of code reuse on the overall system implementation. We have evaluated RCBM component model over LCA, LEACH and MAX-MIN systems and compare the results with those reported by CBCWSN [2]. We used a synthetic scenario with 140 sensors statically placed on a 1400×1000 square meter monitored area. The distance between sensor nodes were around 90 meters, with symmetric links, and using MAC protocol (802.11). The radio range of every sensor on the field was set to 100 meters. Nodes were equipped with GPS devices allowing them to be aware of their geographical position over the monitored area. The simulation duration time was 40 minutes. The simulation parameters are presented in Table 1.

Table 1: Simulation parameters.

Parameter	synthetic scenario
Network devices	140 sensors
Monitored area	$1400mX1000m$
Sensor communication range	100 meters
Simulation duration	40 minutes

Table 2: Proportion of reused code in each system model.

System Model	Component Model	Lines of Code	Reused Code	Diff (%)
MAX-MIN	RCBM	725	480	66 %
	CBCWSN [2]	1140	259	23 %
LCA	RCBM	556	481	86 %
	CBCWSN [2]	671	655	98 %
LEACH	RCBM	488	366	75 %
	CBCWSN [2]	681	661	97 %

Table 2 lists the total number of lines of code, reused lines of code and the difference in percentage of RCBM and CBCWSN models. In both cases, the lines of code were obtained from well-formatted, human-readable programs. While RCBM and CBCWSN design principles result on some common components (such as

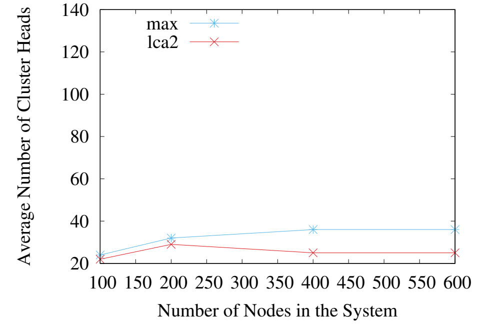

Figure 3: Impact of network density on the number of CHs.

CH election and cluster formation), RCBM components are more flexible because they do not assume any fixed execution flow or criteria. Instead, simple coordination operations are kept up to the developer. As a result, a larger number of models and designs can benefit from the *reusability* of code promoted by RCBM. This is shown by our experimental results. MAX-MIN adopts a distinct sequence of logical steps when compared with the LCA and LEACH approaches, which, in turn, have some similarities to each other. Our results show that RCBM has better code reuse capabilities than CBCWSN for the MAX-MIN implementation, while keeping good *reusability* numbers for the LCA and LEACH systems. These results follow from the fact that RCBM provides a *flexible* component library and execution control flow. In RCBM, we provide a generic code skeleton, which the developer has to fill the gaps in specific places of the code. Vast portions are reused from library components and the user has to code few lines mainly in the “selectCH” and “joinCluster” function skeletons.

CBCWSN follows a different approach. It does not provide a generic code skeleton with gaps. Instead, the developer starts with a predefined set of components implementation that share many similarities with LCA and LEACH. Thus, it is harder to extend CBCWSN to other storage models. As reported in Table 2, CBCWSN reduces significantly the effort to implement LCA and LEACH. However, CBCWSN’s predefined set of components have significant differences compared to the MAX-MIN system design, leading to a higher development overhead.

5.2 Correctness of the Implementation

In this experiment, the goal is to validate the system developed with the RCBM component model. In other words, we would like to check whether the code generated using our approach presents anomalies or if the resulting implementation has the same behavior as the ones previously reported in the literature. To do so, we analyzed the number of cluster heads generated with our implementation of the MAX-MIN and LCA model and compare the results with those reported by the original MAX-MIN paper [3]. The simulation considered a $200X200$ square meters monitored region, varying the network density. We considered networks with 100, 200, 400 and 600 sensors, and for each density we generated five different snapshots scenarios. The radio range of every sensor on the field was set to 20 meters and the maximum number of wireless hops between a node and its cluster-head was set to 2. The simulation settings parameters are shown in Table 3.

Table 3: Simulation parameters.

Parameter	Value
Network devices	100, 200, 400 and 600 sensors
Monitored area	$200mX200m$
Sensor communication range	20 meters
d-hop clusters	2

Figure 3 shows the impact of network density on the number of cluster-heads. Note that the experimental results reported in [3], presents a graph comparing MAX-MIN, LCA2, LCA and DEGREE systems. LCA2 is a modified version of the original LCA, described in Section 4.1, that generates 2-*hops* clusters. In our experiments we have compared MAX-MIN with LCA2, applying the same modification to our LCA implementation to properly compare the results. We noticed that due to the RCBM component model code reuse, we have successfully concluded the implementation, simulation and validation of LCA2 in 2 days, with minor changes in our LCA code reported in Section 4.1. The achieved results, considering MAX-MIN and LCA2, were consistent with the ones reported by the original work [3]. The original values are not plotted in the graph because the actual values are not reported in [3]. However the graph and values of the 2 systems we obtained are almost identical to the original ones.

6 CONCLUSION

In this paper we proposed a Reusable Component-based Model (RCBM) to support and facilitate the development of simulation programs for WSN storage models. RCBM addresses storage-level entities that share concepts and functionalities, which represent various instances of WSN storage systems. These shared functionalities are the components of the system. RCBM considers three types of components: library components, application components, and the coordinator. Library components provide a toolbox, that can be used to implement application components, associated with WSN entities. The coordinator is responsible for the execution flow. Components implementations fill the gaps of a program skeleton provided by RCBM. The skeleton takes into consideration some specificities of the network simulator. In our current implementation, we use the NS2 simulator.

As case studies we developed LCA, LEACH and MAX-MIN models following our proposed approach. Our experiments showed that RCBM provides good reuse capabilities when compared with CBCWSN [2]. The achieved results also show that the code generated for the MAX-MIN and LCA models is compatible in terms of the number of cluster heads with the original MAX-MIN implementation [3]. Furthermore, we showed that RCBM promotes code reuse and agile development during the development of the modified LCA model. These experiments showed that our proposal generates sound implementations. RCBM can be seen as a first step towards a WSN programming environment, in which components code can be generated from higher-level composition primitives.

ACKNOWLEDGMENTS

This work is partly supported by INES, grant CNPq/465614/2014-0 and grant CNPq/486393/2013-5.

REFERENCES

[1] M. M. Afsar and M. H. Tayarani-Najaran. 2014. Clustering in sensor networks: A literature survey. *Journal of Network and Computer Applications* 46 (2014), 198–226.

[2] D. Amaxilatis, I. Chatzigiannakis, C. Koninis, and A. Pyrgelis. 2011. Component based clustering in wireless sensor networks. *arXiv preprint arXiv:1105.3864* (2011).

[3] A. D. Amis, R. Prakash, T. H. P. Vuong, and D. T. Huynh. 2000. Max-min d-cluster formation in wireless ad hoc networks. In *Proc. of the IEEE International Conference on Computer Communications (INFOCOM).* 32–41.

[4] D. J. Baker and A. Ephremides. 1981. A Distributed Algorithm for Organizing Mobile Radio Telecommunication Networks.. In *Proc. of the IEEE International Conference on Distributed Systems (ICDCS).* 476–483.

[5] M. A. Carrero, R. I. da Silva, A. L. dos Santos, and C. S. Hara. 2015. An autonomic in-network query processing for urban sensor networks. In *Computers and Communication (ISCC), 2015 IEEE Symposium on.* IEEE, 968–973.

[6] P. Clements and L. Northrop. 2015. *Software Product Lines: Practices and Patterns: Practices and Patterns.* Addison-Wesley.

[7] G. Coulson, B. Porter, I. Chatzigiannakis, C. Koninis, S. Fischer, D. Pfisterer, D. Bimschas, T. Braun, P. Hurni, M. Anwander, and others. 2012. Flexible experimentation in wireless sensor networks. *Commun. ACM* 55, 1 (2012), 82–90.

[8] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M. R. V. Chaudron. 2011. A classification framework for software component models. *IEEE Transactions on Software Engineering* 37, 5 (2011), 593–615.

[9] G. D´ Angelo, D. Diodati, A. Navarra, and C. M. Pinotti. 2016. The Minimum k-Storage Problem: Complexity, Approximation, and Experimental Analysis. *IEEE Transactions on Mobile Computing* 15, 7 (2016), 1797–1811.

[10] O. Diallo, J. J. Rodrigues, M. Sene, and J. Lloret. 2015. Distributed database management techniques for wireless sensor networks. *IEEE Transactions on Parallel and Distributed Systems* 26, 2 (2015), 604–620.

[11] T. Erl. 2005. *Service-Oriented Architecture: Concepts, Technology, and Design.* Pearson Education. https://books.google.com.br/books?id=y2MALc9HOF8C

[12] R.E. Filman. 2005. *Aspect Oriented Software Development.* Addison-Wesley. https://books.google.com.br/books?id=_nYZAQAAIAAJ

[13] S. S. Furlaneto, A. L. Dos Santos, and C. S. Hara. 2012. An efficient data acquisition model for urban sensor networks. In *Network Operations and Management Symposium (NOMS), 2012 IEEE.* IEEE, 113–120.

[14] M. Gabbrielli and S. Martini. 2010. *Programming Languages: Principles and Paradigms.* Springer London.

[15] P. Gonizzi, G. Ferrari, V. Gay, and J. Leguay. 2015. Data dissemination scheme for distributed storage for IoT observation systems at large scale. *Information Fusion* 22 (2015), 16–25.

[16] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. 2000. Energy-efficient communication protocol for wireless microsensor networks. In *Proc. of the 33rd IEEE Hawaii International Conference on System Sciences (HICSS).*

[17] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo. 2015. Development of autonomous car, Part II: A case study on the implementation of an autonomous driving system based on distributed architecture. *IEEE Transactions on Industrial Electronics* 62, 8 (2015), 5119–5132.

[18] S. K. Khaitan and J. D. McCalley. 2015. Design techniques and applications of cyberphysical systems: A survey. *IEEE Systems Journal* 9, 2 (2015), 350–365.

[19] I. Malavolta and H. Muccini. 2014. A study on MDE approaches for engineering wireless sensor networks. In *Software engineering and advanced applications (SEAA), 2014 40th EUROMICRO conference on.* IEEE, 149–157.

[20] A. Masrur, M. Kit, V. Matěna, T. Bureš, and W. Hardt. 2016. Component-based design of cyber-physical applications with safety-critical requirements. *Microprocessors and Microsystems* 42 (2016), 70–86.

[21] I. Minakov, R. Passerone, A. Rizzardi, and S. Sicari. 2016. A comparative study of recent wireless sensor network simulators. *ACM Transactions on Sensor Networks (TOSN)* 12, 3 (2016), 20.

[22] C. L. Muller, L. Chapman, C. S. B. Grimmond, D. T. Young, and X. Cai. 2013. Sensors and the city: a review of urban meteorological networks. *International Journal of Climatology* 33, 7 (2013), 1585–1600.

[23] R. Niekamp. 2005. Software component architecture. In *Gestión de Congresos-CIMNE/Institute for Scientific Computing, TU Braunschweig.* 4.

[24] P. Patel and D. Cassou. 2015. Enabling high-level application development for the internet of things. *Journal of Systems and Software* 103 (2015), 62–84.

[25] P. Patel, A. Pathak, T. Teixeira, and V. Issarny. 2011. Towards application development for the internet of things. In *Proceedings of the 8th Middleware Doctoral Symposium.* ACM, 5.

[26] A. J. Salman and A. Al-Yasiri. 2016. SenNet: a programming toolkit to develop wireless sensor network applications. In *New Technologies, Mobility and Security (NTMS), 2016 8th IFIP International Conference on.* IEEE, 1–7.

[27] K. Tei, R. Shimizu, Y. Fukazawa, and S. Honiden. 2015. Model-driven-development-based stepwise software development process for wireless sensor networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 45, 4 (2015), 675–687.