Πανεπιστήμιο Θεσσαλίας
Τμήμα Μηχανικών Η/Υ Τηλεπικοινωνιών και Δικτύων

Ανάπτυξη ενός ομότιμου συστήματος για την κατανεμημένη πρόσβαση σε δίκτυα αισθητήρων

A peer to peer overlay network for distributed access to multiple sensor networks

Λάμπρος Παππάς

Διπλωματική Εργασία στα πλαίσια του Προπτυχιακού Προγράμματος Σπουδών του Τμήματος Μηχανικών Η/Υ, Τηλεπικοινωνιών και Δικτύων του Πανεπιστημίου Θεσσαλίας.

Επιβλέποντες Καθηγητές:
Σπυρίδων-Γεράσιμος Λάλης
Βασίλειος Βερύκιος

Ιούνιος 2008

Στους φίλους και την οικογένειά μου

Ευχαριστιες

Ευχαριστώ τον επιβέποντα καθηγητή κ. Σπύρο Λάλη για την πολυτιμη καθοδήγησή κατά την εκπόνηση αυτής της εργασίας. Επίσης θα ήθελα να ευχαριστήσω τους ανθρώπους του Εργαστηρίου Δ1 και ειδικά τον κ. Νίκο Φραγκογιάννη για τη βοήθειά του με την πλατφόρμα των Smart-Its. Τέλος, ένα μεγάλο ευχαριστώ στους φίλους και την οικογένειά μου που με στήριξαν σε όλη τη διάρκεια των σπουδών μου.

Acknowledgements

I thank my supervisor Spyros Lalis for his valuable guidance during all stages to the completion of this thesis. I would also like to thank all the people of Lab D1 and especially Nikos Fragogiannis for his help with the Smart-Its platform. Finally, special thanks go to my friends and family who supported me during my studies.

ΠΕΡΙΛΗΨΗ

Τα τελευταία χρόνια υπάρχει αρκετό ερευνητικό ενδιαφέρον στο πεδίο της εγκατάστασης δικτύων αισθητήρων και της διαχείρισης των παραγόμενων δεδομένων. Στα πλαίσια αυτής της εργασίας παρουσιάζουμε ένα ομότιμο σύστημα που παρέχει τη δυνατότητα υποβολής ερωτημάτων σε πολλαπλά δίκτυα αισθητήρων με βάση την περιοχή εγκατάστασής τους. Οι κόμβοι που εκπροσωπούν το κάθε δίκτυο αισθητήρων οργανώνονονται ιεραρχικά με βάση την τοποθεσία τους. Η ιεραρχία των κόμβων χρησιμοποιείται για την προώθηση ερωτημάτων στα δίκτυα αισθητήρων και για την παράδοση αποτελεσμάτων στους χρήστες που τα υπέβαλαν. Τα πιο σημαντικά στοιχεία του συστήματός μας είναι η αυτόματη κατασκευή της ιεραρχίας κόμβων, η ενημερότητα για τους διαθέσιμους τύπους αισθητήρων και η αποδοτική ομαδοποίηση ερωτημάτων για επαναχρησιμοποίηση αποτελεσμάτων. Δόθηκε επίσης ιδιαίτερη προσοχή στη δυναμική είσοδο και έξοδο δικτύων αισθητήρων στο, και από το, σύστημα και στην αντιμετώπιση βλαβών. Εκτελέσαμε διάφορα πειράματα ωστε να αξιολογήσουμε την προτεινόμενη αρχιτεκτονικη και τα αποτελέσματα έδειξαν οτι ο σχεδιασμός του συστήματος οδήγησε σε μειωμένη επικοινωνία μεταξύ των κόμβων και λιγότερο υπολογιστικό φόρτο στα δίκτυα αισθητήρων.

Το σύστημα που προτείνουμε έχει ως στόχους:

-Τη δυναμική προσθήκη και αφαιρεση υποσυστημάτων δικτύων αισθητήρων χωρίς να καταργείται η σχηματισμένη ιεραρχία.

-Την υποβολή ερωτημάτων από χρήστες μέσω εφαρμογών ή ιστοσελίδων σε πολλαπλά δίκτυα αισθητήρων, μέσω του Διαδικτύου. Οι χρήστες δε χρειάζεται να γνωρίζουν την εσωτερική δομή του συστήματος.

-Τη διαχείριση ερωτημάτων μακράς διαρκείας για μεγάλες περιοχές.

-Την επαναχρησιμοποίηση δεδομένων από παρόμοια ερωτήματα.

Κάθε δίκτυο αισθητήρων που συμμετέχει στο σύστημα εκπροσωπείται από έναν κόμβο (peer). Οι ιδιοκτήτες δικτύων αισθητήρων προσδιορίζουν την περιοχή εγκατάστασης τους δίνοντας δηλωτικά περιοχής στους κόμβους-εκπροσώπους που μοιάζουν με ονόματα DNS (π.χ.: *earth.europe.greece.thessaly.volos.port*). Τα δηλωτικά αυτά φανερώνουν σχέσεις πατέρα-παιδιού μεταξύ των κόμβων (π.χ.: ο κόμβος *earth.europe.greece.thessaly.volos.port. gate1*μπορεί να θεωρηθεί παιδί του προηγούμενου). Οι χρήστες του συστήματος έχουν πλήρη εικόνα των διαθέσιμων κόμβων και μπορούν να ζητήσουν δεδομένα από δίκτυα αισθητήρων με βάση την περιοχή που τους ενδιαφέρει. Στα ερωτήματα ορίζεται η περιοχή ενδιαφέροντος, ο τύπος αισθητήρα (ήχος, φως, επιτάχυνση κ.τ.λ.) η διάρκεια του ερωτήματος και οι περίοδοι δειγματοληψίας δεδομένων.

Οι πρώτες εργασίες στο πεδίο των δικύων αισθητήρων αφορούσαν κυρίως την κατασκευή αισθητήρων και στην ανάπτυξη πρωτοκόλλων για τη μεταξύ τους επικοινωνία αλλά και τη σύνδεση με Ηλεκτρονικούς Υπολογιστές. Σταδιακά προστίθενταν περισσότερες δυνατότητες τόσο στο υλικό όσο και στο λογισμικό τους με αποτέλεσμα να φτάσουμε στην ανάπτυξη μικρών λειτουργικών συστημάτων. Στη συνέχεια αναπτύχθηκαν αρκετά περιβάλλοντα για τη διαχείριση και την ανάκτηση δεδομένων, αρχικά από ένα μόνο δίκτυο και στη συνέχεια από πολλαπλά δίκτυα αισθητήρων.

Στο σύστημα εμφανίζονται 4 κύριες οντότητες το Application Client (AC) API, τα Peer Element (PE), η Peer Element Registry (PER) και τα Sensor Network Front End (SNFE). Για κάθε δίκτυο αισθητήρων που συμμετέχει στο σύστημα υπάρχει ένας ηλεκτρονικός υπολογιστής στον οποίο είναι εγκατεστημένο το PE ωστε να διαχειριστεί ερωτήματα και να ανακτήσει δεδομένα για το συγκεκριμένο δίκτυο. Για κάθε τύπο αισθητήρα υπάρχει ένα SNFE που δρα ως διεπαφή μεταξύ του PE και του λογισμικού που επικοινωνεί με το δίκτυο αισθητήρων (Sensor Network Gateway). Κάθε φορά που ένα PE μπαίνει ή βγαίνει από το σύστημα η PER ενημερώνεται ώστε να ανανεώσει τη σχηματισμένη ιεραρχία. Οι χρήστες υποβάλουν τα ερωτήματα μέσω εφαρμογών ή ιστοσελίδων που χρησιμοποιούν το AC API το οποίο επιτρέπει την αποστολή τους στο σύστημα. Η οντότητα AC κρατάει πληροφορίες για κάθε υποβληθέν ερώτημα και το προωθεί κατάλληλα αφού επικοινωνήσει με την PER.

Όπως αναφέραμε και προηγουμένως οι κόμβοι του συστήματος (PE) οργανώνονται ιεραρχικά σε μορφή δέντρου με βάση τα δηλωτικά περιοχής τους. Η ιεραρχία σχηματίζεται αυτόματα με την είσοδο και έξοδο των PE. Τη διαδικασία συντονίζει η PER ενώ ειδικός αλγόριθμος χρησιμοποιείται για την αποφυγή εμφάνισης κόμβων με πολλά παιδιά.

Τα ερωτήματα προωθούνται σύμφωνα με την ιεραρχία. Κάθε κόμβος (PE) που λαμβάνει ένα ερώτημα το καταγράφει εσωτερικά και το προωθεί στα κατάλληλα SNFE και σε όσα από τα παιδιά του μπορούν να το εξυπηρετήσουν. Αν ληφθεί ερώτημα παρόμοιο με κάποιο που ήδη εξυπηρετείται, επαναχρησιμοποιούνται τα δεδομένα που παράγονται για το παλαιότερο και δεν προωθείται εκ νέου σε SNFE. Δεν προωθούνται ερωτήματα σε υπο-δέντρα της ιεραρχίας που δεν μπορούν να υποστηρίξουν το ζητούμενο τύπο αισθητήρων.

Βλάβες σε κόμβους αντιμετωπίζονται αφού εντοπιστούν από κάποιον άλλο κόμβο χωρίς να διαταράσσεται η ιεραρχία.

Προσομοιώσαμε τη χρήση του συστήματός και διαπιστώσαμε οτι οι αλγόριθμοι και διαδικασίες που χρησιμοποιούνται οδηγούν σε μειωμένη χρήση μηνυμάτων μεταξύ των κόμβων και σε εξοικονόμηση πόρων στα δίκτυα αισθητήρων.

Η σχεδίαση και υλοποίηση του συστήματος, οι προσομοιώσεις και η σχετικές αναφορές αναλύονται στο αγγλικό κείμενο που ακολουθεί.

ABSTRACT

In the last few years there has been a lot of interest in the field of sensor networks' deployment and data management. We introduce an overlay, peer to peer architecture which provides a querying interface for multiple sensor networks, organized with area criteria. The peers that represent single sensor networks are organized in a hierarchy tree which is used for query forwarding and result delivery. The most important features of our system are the automatic hierarchy construction based on the peers' area declaratives, the awareness of sensing capabilities throughout the hierarchy and the efficient query multiplexing and result reuse. We focus on the dynamic entrance and departure of sensor networks in the system and transparent failure handling. Simulation experiments were conducted in order to evaluate the architecture. Results indicate that our design decisions minimize the communication between the peers and computational load of the actual sensing devices at the edges of the architecture.

Table of Contents

Chapter 1

Introduction

In the last few years a lot of research has been made in the field of environment monitoring devices. Various experimental platforms, which contain sensors of various types and send the acquired readings through wireless interfaces, have been designed and manufactured by university and industry research groups. Such devices become cheaper and more powerful and a large number of them are already produced for modern applications like smart-houses, vehicle and transportation management systems, production and supply chains, weather condition probing and detection systems. During the next years sensors are going to become part of more aspects of our everyday life. By becoming smaller and consuming less power, they will be installed in many traditional appliances, in buildings, in portable devices even in our clothes and shoes.

The first research projects in the field of wireless sensor networks were investigating the communication protocols, routing schemes, data collection and aggregation and power-consumption control within a singe network. As sensors become more affordable, we would like to take advantage of their rising presence in order to monitor wide areas of our environment, covered by autonomous sensor networks. A future sensor-use scenario would provide that anyone can deploy a simple sensor network and publish the produced data, making them available to a universal sensor database. However, to reach that day many challenges need to be faced: hardware and software heterogeneity between different sensor networks, complication in the deployment of sensor networks and in data publishing, efficient discovery of non-permanent sensor devices, failure handling, and scalability issues. Obviously, special middleware must be developed, which will bridge the gap between sensor owners and data consumers. In the last 2-3 years various platforms have been devised, providing abstractions for querying multiple sensor networks through the Internet.

We propose a peer to peer platform for querying multiple sensor networks through the Internet with area criteria. Every sensor network is represented by a peer of the system, which receives queries from external users or other peers and forwards them to the sensor network. An area declarative is assigned to each peer. Peers form a hierarchy according to the area declaratives assigned by their owners. The proposed architecture has been designed with the following goals in mind:

- *Ability to add new, and remove existing, sensor network subsystems in a straightforward way, without invalidating the rest of the infrastructure:* Sensor owners are able to connect their network to the system, through an API, in a transparent way. They can add or remove sensor devices of different type or even support an opportunistic sensor network which uses passing-by sensors. Every new sensor network which becomes part of the system updates the architecture for its sensing capability and becomes aware of any queries for which can provide readings.
- *Allow remote applications to submit monitoring queries to the system and receive sensor readings over the Internet*: An API is provided for this reason which abstracts the P2P architecture which lies between the sensors and the end users who request the data.
- *Focus is on long-lived queries with potentially very wide area coverage*. The peers hold state of the posted queries. Sensors, or networks of sensors, which entered the system after the query post, can contribute transparently to the result feed.
- *Support query processing in an abstract area-oriented fashion, with transparent query distribution to, and result aggregation from, the various sensor network subsystems*: Client applications are able to request data either from a narrow or a broader area. Aggregation operators can be applied to the sensor feeds according to the user-queries.
- *Enable simple query multiplexing and result de-multiplexing for better scalability*: When similar queries are posted, the results are efficiently reused to avoid excess network traffic and sensors' power drainage.
- *Self-organization of different sensor networks for better scalability*: The peers of the system automatically create a tree hierarchy according to their area declaratives. As we show with experiments, by using this hierarchy scheme we can achieve more efficient query forwarding, state keeping, and result reuse and thus, spare network and sensors' resources.

The sensor networks' owners give area declaratives to the peers in a DNS like form (e.g.: *earth.europe.greece.thessaly.volos.port*). The declaratives imply father-child relationships between peers (e.g.: *earth.europe.greece.thessaly.volos.port.gate1* can be considered as "child" of the previous node) though the final hierarchy is decided by a standard and more complex scheme that we describe later. Both sensor deployers and client users are able to have a view of the available peers of the system.

This is a typical use scenario from a user’s point of view: Bob wants to know the average temperature in Athens every 2 hours for the next 5 days. He visits a website, which invokes our Client API and posts the query to the system. All the available peers that can give temperature readings in the desired area are properly “informed”. If new sensor networks with temperature sensing capability join the system, they offer their data after been connected to the hierarchy. Sensor networks’ departures and failures are confronted by the P2P network and are not visible to the end-user.

The innovation of our system lies mainly to the hierarchy organization scheme we propose based on area declaratives. Based on the tree hierarchy we propose efficient query forwarding and result multiplexing and reuse. Another innovative feature is the automatic detection of sensor devices at the edges of the system.

In the next chapter we refer to some projects related to our work, both early ones, about single sensor network data management, and posterior, in the field of combination and management of multiple networks of sensing devices. In Chapter 3 we present the system architecture, in Chapter 4 we describe how the peers’ hierarchy is constructed and in Chapter 5 the way queries are submitted to, and managed by, the system. An implemented sensor gateway is described in Chapter 6 and in Chapter 7 we give some simulation results that either explain some of our design options or prove the efficiency gain we get from the self-organized peer hierarchy. Finally we conclude our work and look on future plans in the field of sensor networks’ management.

Chapter 2

Related Projects

During the last few years, environment monitoring devices have become smaller, cheaper and more powerful. Sensors of all types are ready to become part of our cities, houses, cars and even our clothes. Much research has been made in various fields of sensor networks' deployment and management. The common target of the projects presented in this chapter is the efficient monitoring of our environment through the use of sensing devices. In the first section we present some devices and architectures that can be used by one to build a single sensor network in order to monitor a specific area in an efficient and transparent way. In the next 6 sections we present projects which aim to merge data from different networks and provide an abstraction for users to query multiple sensor networks, possibly with location criteria. Finally we present some research efforts studying the integration of wireless sensor networks and grid computing.

2.1. Early projects in the field of sensor networking

Naturally, the first research efforts in the field of sensor networking aimed to the design and fabrication of hardware platforms (usually referred as motes) which contain simple sensors and are able to communicate over the air with each other and/or more powerful systems (most commonly a PC) in order to acquire and store the produced readings. **Mica Motes** [1], **Smart-Its** [2] and **Intel Motes** [3] are some of these experimental platforms with which one can deploy a small-area wireless sensor network. A list of wireless sensor motes can be found here [4]. Along with the hardware platforms, communication protocol stacks and APIs were created in order to facilitate the communication of PCs (through some kind of gateway) with the motes. Later more advanced features were added and lead to the development of simple operating systems providing various services such as sensor data storage, message routing, power control and more.

The most widely cited effort in the field of sensors' operating systems is **TinyOS** [5] by Berkeley University and Intel Research. TinyOS is an embedded operating system written in **nesC** [6], a subset of the C programming language optimized for the resource limitations of sensor devices. TinyOS applications are developed in nesC and are made out of software components, some of which represent hardware abstractions and are connected to each other using proper interfaces. TinyOS provides components and interfaces for widely used abstractions like packet communication, message routing, sensing, actuation and data storage. All I/O operations in TinyOS- enabled motes which take a long time, are asynchronous and have a callback. For optimization purposes these callbacks, called events, are linked statically to the applications during compiling. While being non-blocking enables the TinyOS to maintain high concurrency, it leads the programmers to write complex code using many small event handlers. To support larger computations the "tasks" feature is provided. Tasks are non-preemptive and run in FIFO order. A TinyOS component can post a task, which the operating system will schedule to run later. This simple concurrency model is fairly sufficient for I/O centric applications, but its difficulty with CPU-driven applications has led to several proposals for adding thread-execution. TinyOS code is statically linked with application code, and compiled into a small binary which is then installed in the sensor motes. TinyOS is free and open source and has been ported in a large variety of experimental hardware platforms.

Apart from the internal functions of wireless sensor motes, various software platforms have been developed in the field of sensor data collection and representation. For example **MoteLab** [9] is a sensor network testbed developed at Harvard University which provides a web interface that makes it easier for users to program the motes, create sensor jobs, reserve job-execution time slots, collect the sensor data, and perform administrative functions. Other sensor network management projects include **EmStar** [7], **Kansei** [8] and various other approaches which are, in most cases, designed for specific applications.

A powerful system built upon the previously referred TinyOS, is **TinyDB** [10, 11] by Berkeley University. TinyDB is a query processing system used for extracting data and information from a network of TinyOS-enabled sensor motes. It does not require writing nesC code for the motes as it provides a simple, SQL-like interface from which the user specifies the data to be extracted, along with additional parameters, like the rate at which data should be refreshed. Given a query specifying the user data-interests, TinyDB collects the desired data from sensor-motes in the environment, filters it, aggregates it and routes it to the host-PC. The primary goal of TinyDB is to free developers from writing complicated low-level code for sensor devices and to offer important features like metadata management (providing a metadata catalogue to describe the types of sensor readings that are available in the network), high level queries declaration, sensor network topology management (by tracking neighbouring motes and maintaining routing tables), multiple query management (allowing multiple queries to be run on the same set of motes at the same time), results' reuse and query forwarding to new-coming motes. TinyDB provides a Java API for writing PC applications that query and extract data from the sensor network and also contains a simple graphical query-builder and result display that uses the API. In order to use TinyDB, its TinyOS components must be installed onto each mote in the sensor network.

2.2. Simple sensor networks' gateways

In this section we present some projects introducing software components that make wireless sensor devices accessible through the Internet.

VIPBridge [12] is a platform through which users can send queries to many, distinctive sensor networks. The main idea of the system is the mapping of every available sensor of the system with a unique IP (version 6) address. The target is achieved through the use of a bridge-component in every sensor network, which is aware of the number of sensors it represents and maps them with unique IPv6 addresses. Every time an application needs data (or meta-data) from a sensor, the relevant query is sent in IP level; it is then received by the bridge-component, becomes properly transformed and finally is forwarded to the target sensor through the proper communication protocol. The results produced by sensor devices follow the reverse route. There are similar projects [13, 14, 15] whom basic target is to make sensor devices visible to the Internet and accessible from applications via classic TCP/IP stack.

A more advanced approach is [16] where services from different sensor networks can be accessed and combined with the use of *JXTA* technology [17] and *Universal Plug and Play* standard [18]. JXTA is used to form a network of P2P nodes exchanging data from sensor networks and UPnP provides a platform for transparent access of sensing services. Each node (*P2P bridge*) represents a specific sensor network and declares its attributes (e.g.: location, number and type of sensors etc) in a JXTA advertisement. Sensor data is exchanged, merged and filtered amongst the nodes through JXTA messages. Node discovery and message routing are conducted by the JXTA infrastructure. Client applications access the available sensing services through a UPnP Gateway. For that cause a *UPnP proxy* is created for each service.

2.3. The HiFi Project

HiFi [19] is a research project aiming in the collection and aggregation of data produced by large fan-in systems. The proposed architecture can be used in scenarios where organizations with hierarchical structure need to process large amounts of data produced in their edges. Typical scenarios of that type are nation-wide production and supply chains empowered with RFID tags and readers, power production and consumption management networks, networks collecting data from sensing devices or computer and communication monitoring systems. Such systems require collection, filtering and cleaning of data produced in their edges, successive aggregation as data move inwards the system hierarchy, strong temporal and spatial focus and effective integration within and across different enterprises. The HiFi architecture's major components are a *Meta Data Repository*, the *Data Stream Processor* and the *HiFi glue* (the least two being parts of the system nodes).

The *Metadata Repository (MDR)* is a globally accessible registry for system-wide information. The metadata the MDR holds is of three types: schema, views, and system information. The schema contained in the MDR is the schema over which a specific application's queries and views are written. The views stored in the MDR are those exported by each HiFi node. The MDR also maintains a mapping of the views exported by a node and its physical location, which is vital for keeping the nodes' topology. The system information contained in the MDR includes node capabilities, access control, and information related to organizational boundaries and administrative domains. Additionally, the MDR maintains runtime information, such as the current set of queries running on each node, the network usage, and connection status of the nodes.

The *Data Stream Processor (DSP)* is responsible for all single node data stream processing. The core functionality expected of a DSP is the ability to process continuous queries, add queries and sources on-the-fly and cancel queries. A DSP can also provide functions for modifying and suspending a query along with data streams archiving. The DSP is oblivious of HiFi and could be any stream processor such as **TelegraphCQ** [20], **Aurora** [21] or **STREAM** [22].

The *HiFi Glue*, which runs on each HiFi node, is the software component which seamlessly binds together the system. It coordinates its local DSP, communicates with other HiFi nodes, and manages incoming and outgoing streams. The HiFi Glue consists of services, which perform actions such as query planning, management of DSP, resource, views, archives and cache, handling of queries and produced data and system management.

The HiFi research team has built a testing version of HiFi using the TelegraphCQ stream query processor and the TinyDB sensor database system.

2.4. The IrisNet Project

IrisNet [23] is a distributed architecture enabling convenient deployment of wide area sensing services collecting data from heterogeneous sensor networks. IrisNet comprises of two different types of modules, *sensing agents (SAs)* and *organizing agents (OAs)*. All agents run onto Internet-connected PCs.

Each *SA* is directly connected to one or more sensing devices varying from temperature and pressure meters to webcams and microphones. Every SA running on a host provides a common runtime environment for the services running on the IrisNet, to share and filter the sensors' data.

OAs are organized in groups (one group for each service). A group of OAs creates the distributed database and query processing infrastructure used for a specific service. Each OA holds a local database, storing sensor data from various SAs. A group of OAs combines these databases into a distributed database dedicated to the specific service. The data is replicated and moved into the distributed database according to the service's special requirements. OAs are organized in a location-based hierarchy. A distributed algorithm is deployed throughout the OAs, using statistics held by them, to decide which parts of the distributed database are replicated or partitioned, targeting to smaller average query response time and network traffic.

Various services are created by the IrisNet research group (a *Parking Space Finder* service, a *Network and Host Monitoring* service and a *Coastal Imaging* service). The system is designed for dynamic implementation and deployment of sensing services. The SAs are easily programmable. Service editors may write and upload simple code to SAs (senselets) which dynamically filter and store sensor data with desired attributes. They are also able to use SAs to collect and feed sensor data from their own sensor networks or even implement their own SAs for specific hardware not yet supported by IrisNet. The data feeds from the sensing devices are internally represented in XML format which is well suited in describing hierarchical data with self describing tags. Users' queries are processed with the use of XPath and other XML technologies.

2.5. The SenseWeb Project

Microsoft’s **SenseWeb** [25, 26] is a platform which provides mechanisms to store sensor data, to process queries, to aggregate and present results through easy-to-use map-like web-interfaces. The project’s main objective is to provide sensor network owners with tools that make their sensor-feeds accessible from many users worldwide. The project consists of four main components: the *GeoDB* sensor index, the *DataHub* data publishing toolkit, the *IconD* aggregator and the *client-side GUI*.

GeoDB is a geographically indexed database maintaining the sensor descriptions. A sensor description is the meta-data describing the location, sensor type, owner, data rate and visualization options for the sensors, represented in XML format. GeoDB indexes data by using hierarchical triangular mesh (HTM) scheme which is suitable for location-related queries. Indexing is implemented in SQL server. No real-time data is stored in this database.

DataHub is the web service which provides sensor meta-data to the GeoDB database and sensor data to the system in response to user queries. A sensor network owner first registers the sensor description with DataHub, which redirects the description to GeoDB, and then the publisher can start sending sensor data. Both scalar (e.g.: temperature, humidity) and multimedia (e.g.: audio, photos, video) data are supported and are represented using an ontology that describes inheritance, associations, and compositional relationships between various sensor data types. Data may be cached or permanently stored following the system’s needs.

IconD is the system’s query processor and aggregator. Given the users' queries based on location range, map zoom level, sensor type, and aggregation operators, IconD queries the GeoDB for sensor meta-information and then corresponding DataHub services to get the real-time sensor data. It properly aggregates the data and finally creates icons that will be displayed at the client interface.

The sensor network owner is able to upload data from cameras or TinyOS-enabled sensors by simply using the SenseWeb Data Publishing Toolkit (DataHub web-service). Users query the data through a web-page [27]. The GeoDB and the IconD aggregator are transparent to both the data owners and users. Up to date users can get data about traffic, temperature and pollution conditions in various US cities.

2.6. The “Global Sensor Networks” Project

Global Sensor Networks (GSN) project [28] introduces a middleware which supports flexible integration and discovery of sensor networks, enables fast deployment of new platforms, provides distributed querying, aggregation and combination of data, and supports the dynamic adaptation of the system configuration during operation. The main target of the system is to face the heterogeneity of the available software and hardware platforms and thus to minimize the unnecessary and repetitive implementation of similar functionalities for different platforms. An important feature is the possibility to filter sensor network data from local or remote sensor networks through simple SQL-like queries. The design of GSN follows four main design goals: *Simplicity* (a minimal set of powerful abstractions is used), *adaptivity* (adding new types of sensor networks and dynamic (re-) configuration of data sources is supported during run-time), *scalability* (through a peer-to-peer architecture), and *light-weight implementation* (small memory usage, low hardware and bandwidth requirements, web-based management tools).

The key abstraction in GSN is the *virtual sensor*. Virtual sensors provide an abstraction hiding the implementation details of access to sensor devices and they are the actual services provided and managed by the system (*GSN container*). A virtual sensor corresponds either to a data stream received directly from sensor devices or to a stream coming from other virtual sensors. A virtual sensor can have any number of input streams and produces one output stream. The specification of a virtual sensor provides all necessary information required for deploying and using it.

The production of a new output stream element of a virtual sensor is triggered by the arrival of a data stream element from one of its input streams. Afterwards, proper timestamps are given to the data stream, the desired SQL filtering is performed, the data is temporarily stored and finally any consumers of the virtual sensor are notified for the new stream element.

The production time of sensor data is a very important value. Each data stream is represented as a sequence of timestamped tuples. Multiple time attributes can be associated with data streams and can be manipulated through SQL queries. GSN provides a number of features to control the temporal processing of data streams, in order to avoid undesirable delays and exhaustion or unneeded reservation of resources.

GSN follows a container-based architecture. Each *container* can host and manage many virtual sensors regulating almost every function of them including remote access, communication with the sensor network, security policy, persistence, data filtering, concurrency, and access to resources. Sensor network owners create virtual sensor descriptions which contain key-value pairs (in XML format) and publish them to a peer-to-peer directory so that virtual sensors can be discovered and contacted based on any combination of their properties, for example, geographical location or sensor type. GSN nodes are organized in a peer-to-peer network targeting in greater scalability.

The GSN implementation consists of the *GSN-CORE* (mainly the GSN container), implemented in Java, and the platform-specific *GSN-WRAPPERS*, implemented in Java, C, and C++, which are used to communicate with the actual sensing devices depending on the used technology. For deploying a virtual sensor the sensors' owner has to specify a *virtual sensor descriptor* in XML format, if GSN already supports the concerned hardware and software. Supporting a new type of sensing device can be achieved by supplying a *Java wrapper* compliant to the GSN API interfacing the system to be supported. Currently GSN provides wrappers for TinyOS-enabled motes (Mica, Mica2, Mica2Dot and Telos), USB and wireless cameras, and some RFID readers. GSN implementation also includes *visualization tools* for data and network structure presentation.

2.7. The Hourglass Project

Hourglass [29] is another project which tries to address the need for rapid development and deployment of applications that consume data from multiple, heterogeneous sensor networks. Hourglass is an Internet-based infrastructure for connecting a wide range of sensors, services, and applications in a transparent way. The infrastructure consists of an overlay network of well-connected machines which provides service registration and discovery, and routing of data streams from sensor networks to client applications. Hourglass supports a set of internal services such as filtering, aggregation, compression, and buffering of sensor data. It also allows third party services to be deployed and used in the network. Other features of the system include the preservation of data flow in cases of disconnection, the support for participants of widely varying capabilities (from powerful servers to PDAs), the utilization of powerful and well-connected machines and the effective separation of communication paths for short-lived control messages and long-lived stream-oriented data.

Circuit is the key abstraction of the system that links a set of data producers, a data consumer, and in-network services into a data flow. Control messages are used to set up the sensor data channels that travel over multiple services. Any data produced, are processed by intermediate services and then delivered to consumers. The structure of a circuit is specified in the Hourglass Circuit Descriptor Language (HCDL), an XML-defined language that is used to describe circuits to be established by Hourglass. Circuits can transparently face temporary hosts' disconnections. A circuit has a unique circuit identifier that is used to refer to it throughout the system.

The nodes in a circuit are realized by Hourglass *services*. A service can function as a pure data producer, a pure data consumer, or both (in that case called operators). The services in Hourglass are of two categories, generic services that are useful to a wide-range of applications, offering for example buffering, filtering or storage of data, and application specific services. The set of the available services is partitioned with the use of *topics* which describe the services' attributes. *Service endpoint*s are defined that bind circuit nodes to actual service instances.

The services in Hourglass are arranged into *service providers*. A service provider is comprised of one or more Hourglass nodes. Each service provider is contained in a single administrative domain and joins or leaves the system as a unit. A service provider must support a minimum functionality in the form of a *circuit manager* (which manages the circuit creation process and monitors its status ever since) and a *registry* (which acts as a repository of information about the various services and circuits) in order to join the system.

The establishment of a new circuit is initiated by an application either directly or through a proxy service. The application contacts with one of the circuit managers which "hides" the communication with the actual data producers or operators. Sensor data streams are routed from producers to consumers along the circuit-defined paths. The system reuses the actual network connections between services as much as possible and provides query multiplexing. When a service provider is disconnected from the rest of the Hourglass system, local services become aware of the lack of heartbeat messages on their circuit links and act properly in order to temporarily store any produced data.

The Hourglass research team has created health-related applications built upon Hourglass [24] and is currently investigating the efficient placement of operators in a Data Collecting Network like Hourglass [30].

2.8. Sensor Networks and Grid Computing

Recently, research efforts have emerged studying how sensors can be integrated into grid computing applications. One of them is the **Discovery Net** project [32] which is a grid-based framework for developing and deploying knowledge discovery services to analyze data collected from distributed high throughput sensors. However, frameworks of this type tend to use sensor grid architectures that are custom built for specific applications. Although these application-driven architectures are efficient and can deliver good performance for the targeted applications, they are not flexible with generic use-scenarios.

Another approach is [42] where sensor data is enclosed in XML format within SOAP envelopes (in order to be compatible to grid standards) and then are forwarded to applications using web-services and standard Internet protocols.

There are also projects which aim to define middleware architectures connecting sensor networks with the grid. One of them is the **Common Instrument Middleware Architecture (CIMA)** project [33] which aims to “grid-enable” instruments and sensors as real-time data sources to facilitate their integration with the grid. The CIMA middleware is based on current grid standards such as **OGSA** [34]. This middleware architecture uses a standard instrument representation format and software stack. A problem with this approach is that the middleware architecture might be too complex to be implemented on simple sensor devices with low computational and processing capability.

Another approach is the **SPRING** framework [35] which integrates wireless sensor networks with grid computing by using proxies as interfaces between the sensor networks and the grid, supporting a wide range of sensor devices, even the less computationally powerful ones. The system tries to address the challenges and design issues arising when trying to create sensor-enabled grids like the lack of efficient grid-APIs for sensors, the dynamic and prone to faults nature of communication links between sensor devices , proprietary communication protocols, scalability issues, sensors’ power management combined with Quality of Service(QoS) requested from grid applications, sensor resources scheduling, security problems of wireless sensor devices and availability issues. The system’s main component is the *Wireless Sensor Network proxy* (*WSN-proxy*), which acts as the interface between a sensor network and the grid. The proxy serves several important functions, facing the previously mentioned challenges. *First*, it exposes the sensor network resources as grid services and translates the sensor data from its native format to a suitable OGSA format. *Second*, the proxy controls the communication between the wireless sensor network and the grid. By using techniques like caching, buffering, and link management, the proxy is able to cushion the effects of unexpected sensor nodes disconnection or hardware faults. *Third*, the scalability of the sensor grid is enhanced as new wireless sensor networks can be added seamlessly to an existing sensor grid. *Finally*, the WSN-proxy provides various services like power management, scheduling, security, availability, and QoS for the underlying wireless sensor network. Apart from the WSN-proxy a SPRING-based sensor grid contains various software components (layers) found in classic computational or storage grids like *grid meta-schedulers, user interfaces and APIs*. In the testbed created by the research team, standard grid solutions are used such as the **Globus Toolkit** [36] to implement the grid interfaces, the **Community Scheduler Framework** [37] to implement the grid meta-scheduler and the **Sun Grid Engine** [38] which plays an important role in the implementation of the WSN scheduler and the Resource Scheduler of the compute cluster.

Chapter 3

System Architecture

In this chapter we will present some of the basic principles of our system along with its software components and some of the used data structures.

3.1. Motivation and basic principles

Sensor devices become cheaper, more powerful and energy- efficient, and gradually become part of more aspects of our everyday life, with rising presence in buildings, home devices, vehicles, even in clothes and shoes. While the first research projects had to do mostly with the management of data produced within a single sensor device or a simple network of them, the last few years various projects have emerged which provide data collection and management from different sensor networks. In the previous chapter we described the most important efforts in this field.

We introduce a peer to peer platform which provides a querying interface for multiple sensor networks, organized with area criteria. Our system tries to face the challenges of sensor data collecting platforms: *hardware and software heterogeneity between different sensor networks, complication in the deployment of sensor networks and in data publishing, discovery of non-permanent sensor devices, failure handling, and scalability issues*.

The most important feature of our system is the *hierarchy organization scheme* we introduce, based on area declaratives. Additionally we propose a *sensing capabilities' awareness scheme* throughout the hierarchy, which minimizes unneeded queries to sensor networks that do not support the desired properties, and *efficient query multiplexing and result reuse*, which reduces the communication between the nodes of the system and the computational load of the actual sensing devices at the edges of the architecture. Attention is also paid in *data representation* of queries and results, in *handling of communication links' or hosts' failures,* in the *transparency* of adding and removing existing sensor network subsystems and *the facilitation of writing communication code for* devices that are not already supported.

The system comprises four main components, the *Application Client* API, the *Peer Elements*, a *Peer Element Registry* and *Sensor Network Front Ends*. For each participating sensor network there is a representative host-PC where a Peer Element (*PE*) is installed in order to manage the queries for the network it represents, and collect any available data. For each type of monitoring device used in the sensor network, a respective Sensor Network Front End (*SNFE*) is installed in the representative-PC and is registered to its Peer Element being an interface between the peer to peer architecture and the Sensor Network Gateway that directly communicates with the network of monitoring devices. The Peer Element Registry (*PER*) keeps state for every participating sensor network. Every time a Peer Element joins or leaves the system the Peer Element Registry is contacted, in order to coordinate the peers' hierarchy update. End users post their queries to the system through applications or web-interfaces that use the Application Client API (*AC*). The Application Client component keeps state of all the posted queries per client interface and properly forwards them, after contacting the Peer Element Registry in order to find a Peer Element that can serve them.

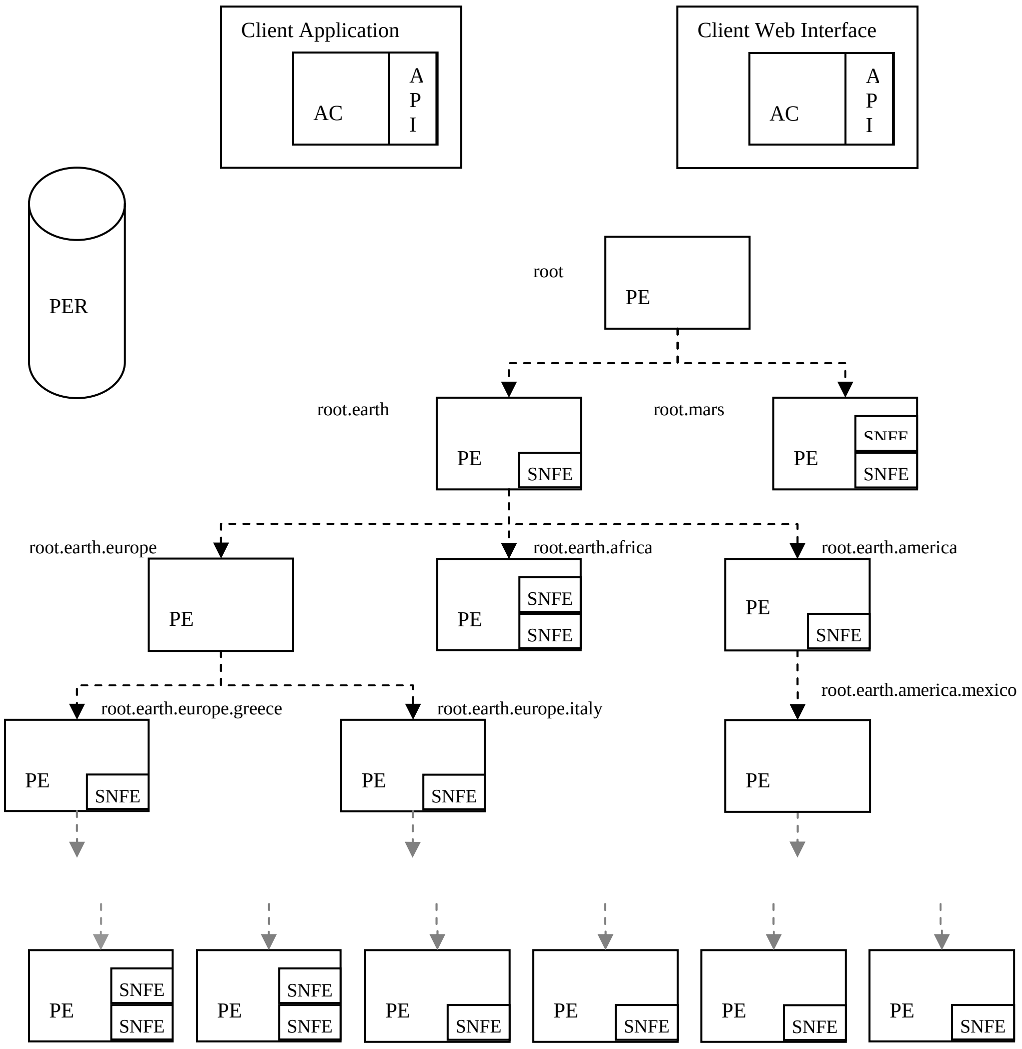


Figure 3.1: The System Overview

3.2. The Application Client (AC) Component

This component resides within the application space. It provides an API and internal processing logic through which communication takes place with the rest of the system. For each query, the AC finds through the PER a suitable PE to which it forwards the query, then keeps the query state and becomes responsible for receiving the corresponding result-data. The application receives the query results and is able to cancel the query by calling the respective functions of the AC API.

The AC API provides the following functions:

QueryID postQuery(queryString): With this method, client applications submit the desired queries to the system. The queryString must be of the form of "**`query:[avg|max|min|sum|count]<senseType>in:<area>for:<totalTime>every:<time Interval>`**". When this method is invoked, proper state about the query is held in the AC component and the query is sent to the system, so as to be forwarded to any PEs that can provide related sensor data. The method returns either a unique (in the AC scope) QueryID or an error code in cases of connection failures or malformed query string. If the desired sense type is not yet supported in the PEs that represent sub-areas of the desired area the QueryID is returned along with a *NOT-YET-SUPPORTED* code. If returned, the ID can be used as argument to API methods for query management or result retrieval.

List<ResultData> getResults (QueryID): This function checks if there is any sensor data for the query with the desired QueryID. Result data is asynchronously received by the AC component and properly stored in data structures of the component. There are two versions of the function, one returning a list of ResultData (a data structure described later) objects and another returning results in stings. Null is returned if no query with the requested QueryID is found in the system.

returnCode cancelQuery(QueryID): This method cancels the query with the respective QueryID. A *CANCEL-QUERY* request is forwarded to the network of PEs which serves this query. All the data and state held within the AC are erased if PEs acknowledge for local query cancel. Proper success or error codes are returned.

areaStrings availableAreasUnder (areaOfInterest): This is a utility method which returns the area-declaratives of all the PEs representing sub-areas of areaOfInterest.

sensesString (areaOfInterest) : This method returns the sensing capabilities for which PEs in sub-areas of areaOfInterest can produce results.

Apart from the API, the AC provides an internal processing logic used to keep state of the posted queries and to manage the results.

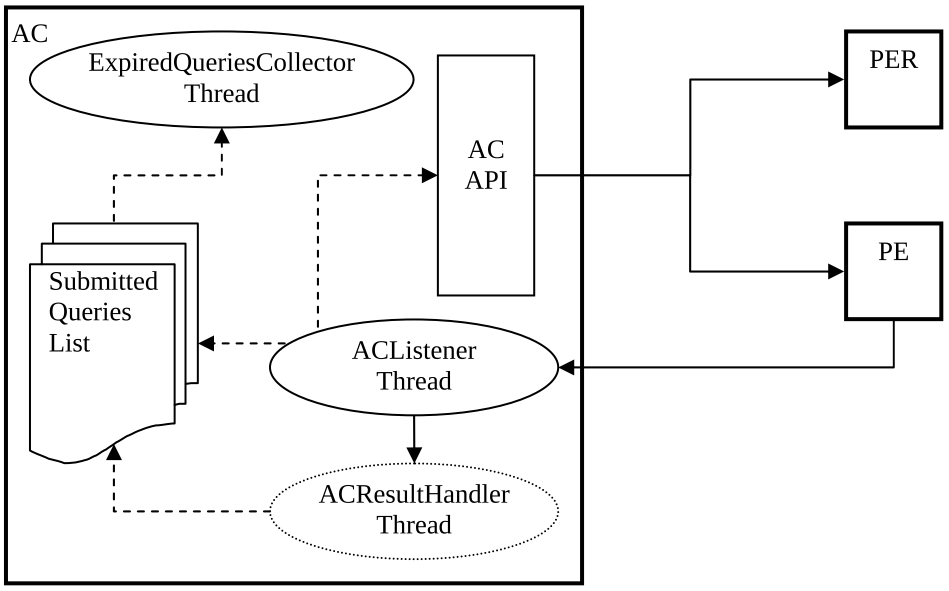


Figure 3.2: The AC component

Every time a query is submitted through the API, a data structure is created (*QueryInACObject*) which describes the query's basic attributes (more information about *QueryInACObject* in section 3.6) and afterwards a *QueryObject* is forwarded to a PE suggested by the PER. The *QueryInACObjects* for all the submitted queries are stored in the *SubmittedQueries* list. As we show in figure 3.2 there is the *ACListener* thread that listens to a socket for *ResultObjects* (also described in section 3.6) coming from various PEs. Whenever a *ResultObject* arrives at the socket an *ACResultHandler* thread is created that parses the *ResultObject* and stores the contained sensor data in the proper object(s) of *SubmittedQueries* list, so as, the application using the AC, is able to get them asynchronously by calling the *getResults* method of the API. More than one *ACResultHandler* threads may run simultaneously handling multiple result objects. There is also the *ExpiredQueriesCollector* thread which checks the *SubmittedQueries* list for queries that expired. If expired queries are found, they are erased from the AC component.

3.3. The Peer Element Registry (PER) Component

This component resides on a well-known server and port, and provides registry and discovery services for PEs. Its role is vital for the hierarchy formation and management. The PER component is the one which makes decisions about each PE's position in the hierarchy.

Every time a new PE joins the system, it first contacts the PER in order to find which PE is going to be its father-PE and whether there are PEs which will become its child-PEs. The PER is also contacted in cases of PE departures or failures so as to update the state of the P2P network. The father-child relationships are stored in the PER registry.

Read Only Requests:	***Hierarchy Updating Requests:***
PE-LOOKUP request	*JOIN* request
AVAILABLE-SENSING-CAPABILITIES request	*DEPART* request
AVAILABLE-AREAS request	
FAILURE request	
UPDATE-SENSING-CAPABILITIES request	

Figure 3.3: The requests handled by the PER

The PER handles two request categories. The first ones are hierarchy updating requests received by PEs that either enter or leave the P2P architecture (*JOIN* and *DEPART* requests). The handling of these requests is synchronous, forcing every other request to wait until the invoking PE sends to the PER an acknowledgement of proper completion. If any error occurs in the underlying protocols, the PER database is roll-backed to its previous state.

The other category contains read-only requests either from ACs (*PE-LOOKUP*, *AVAILABLE-SENSING-CAPABILITIES* and *AVAILABLE-AREAS* requests) or from PE components (*FAILURE* and *UPDATE-SENSING-CAPABILITIES* requests).

We will briefly describe how the PER reacts to these requests:

JOIN: PE nodes send a *JOIN* request when they initialize and join the system. The joining PE sends its network address, its available sensing capabilities and its area declarative. The PER runs an algorithm (which we describe later) and decides, according to the joining PE's area declarative, in which point of the tree hierarchy it is going to be placed, thus which will be its father-PE and if there are any child-PEs. The PER sends back to the PE the father-PE and child-PE network addresses

DEPART: When a PE terminates, it a *DEPART* request is sent to the PER in order to update the hierarchy state. The PER decides which PE is going to become the father-PE of the terminating-PE's child-PEs and informs the terminating PE, which in turn informs its child-PEs. The record of the departing PE is erased from the database if the request handling terminates properly.

PE-LOOKUP: This request is sent by an AC component whenever it has to find the PE which is representative for the target area defined in a query. The PER looks on its database for the proper PE entry and its network address is sent back.

FAILURE: This request is sent by a PE or AC component if another PE does not seem to respond. The PER receives the failing PE's information and checks if it exists in its database. It then checks if the PE is responding. If no response is received in a fair time space and the detecting component is another PE, it sends a response to it giving the permission to run the departure protocol on behalf of the failed PE. IF an AC or a departing PE detected the failure, a *CHECK-PE* request is sent to the root-PE in order to perform the failure handling procedure.

UPDATE-SENSING-CAPABILITIES: Whenever a new sensor network is added in a peer of the system (thus a new SNFE is registered) the available sensing capabilities of the PE component are possibly increased. In that case an *UPDATE-SENSING-CAPABILITIES* request is sent to the PER in order to update the proper entries.

AVAILABLE-SENSING-CAPABILITIES and *AVAILABLE-AREAS*: These requests are sent by ACs. The PER searches its database and sends the requested information to the remote AC.

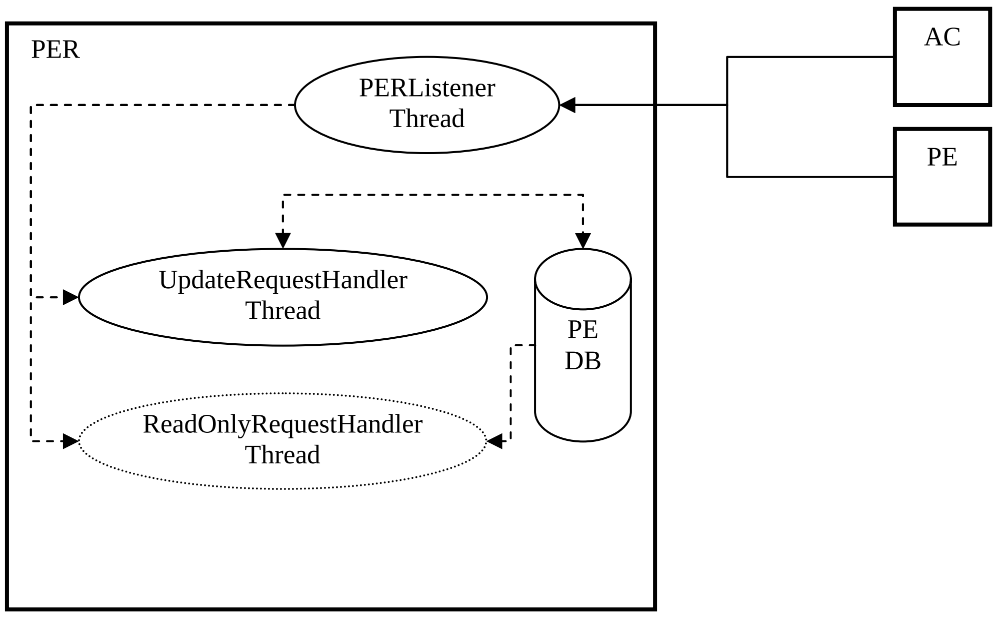


Figure 3.4: The PER component

The Peer Element Registry in our implementation resides in a well known ip and port. There is a *PERListener* thread which listens to a socket on this network address and waits for requests from AC and PE components. When a blocking request is received, the *UpdateRequestHandler* thread is awakened to handle it. Afterwards, it is suspended until another updating request needs to be handled. Hierarchy updating requests (blocking) are not handled simultaneously in order to avoid race conditions. When a read-only request arrives, a new *ReadOnlyRequestHandler* thread is created to handle it. More than one

ReadOnlyRequestHandler threads may run simultaneously handling multiple read-only requests.

As shown in figure 3.4, we use a DBMS to store information about the registered PEs. We use JDBC functions to access the database. In our implementation the DBMS used is version 4 of MySQL. However the PER is easily reconfigurable and any JDBC-enabled DBMS can be used provided that we acquire the corresponding JDBC driver.

```
+-------------------+-------------+------+-----+---------+----------------+
| Field             | Type        | Null | Key | Default | Extra          |
+--------------------+-------------+------+-----+---------+----------------+
| id                 | bigint(20)   |      | PRI | NULL    | auto_increment |
| area               | varchar(200) |      |     |         |                |
| representativeFor  | text         |      |     |         |                |
| sensingCapabilities| varchar(200) |      |     |         |                |
| ip                 | varchar(50)  |      |     |         |                |
| port               | int(11)      |      | 0   |         |                |
| fatherID           | bigint(20)   |      | 0   |         |                |
| realFather         | int(11)      |      | 0   |         |                |
| markedBy           | bigint(20)   |      | 0   |         |                |
+--------------------+-------------+------+-----+---------+----------------+
```

Figure 3.5: The PER database table

For every PE registered in the PER database the following entries are filled-in:

ip and port: These fields contain the network address in which the PE listens for requests from ACs or other PEs.

id: This field is created automatically by the database in order to define uniquely the PE registries.

sensingCapabilities: This field describes the sensing capabilities supported in the area the PE represents. Capabilities are not necessarily provided by sensors of the particular PE, but it may be supported by any PE which is one of its hierarchy descendants.

area: This field contains the area declarative of the PE, as it is defined by its owner.

representativeFor: This field contains the areas for which the PE is representative for. For example if a PE with area declarative "*earth.greece.thessaly*" exists and a PE with area: "*earth.greece.thessaly.volos.port.gate1*" joins the system, the second PE becomes a child of the first and the first becomes representative for the areas "*earth.greece.thessaly.volos*" and "*earth.greece.thessaly.volos.port*".

fatherId: In this field the id of the father-PE is stored.

markedBy: This variable is used in failure handling procedure. If it is equal to -1 the PE is not marked as failed. If it is equal to another PE's id, it means that this PE is marked as failed by the PE with the respective id. It prevents the simultaneous failure handling for the same PE by more than one PEs.

realFather: This field is a boolean variable showing whether the father-PE of the described PE is its direct father (e.g.: a.b.c is direct father of a.b.c.d) or an ascendant and became its father because intermediate PEs do not exist (e.g a is father of a.b.c.d if a.b and a.b.c PEs do not exist. We are able to get this information by other fields but we added it to avoid complicated string-parsing database queries.

3.4. The Peer Element (PE) Component

This component mediates between ACs and available sensor networks, perhaps via other PE components, to support the desired query distribution and result delivery. PEs may be installed on any computer with a connection to the Internet; one could place PEs on routers or DNS servers. PEs can be added and removed in a dynamic fashion, while obeying a certain join protocol. The entire intelligence of self-organization, query multiplexing, distribution and forwarding as well as result aggregation and de-multiplexing is implemented with this component.

Hierarchy Updating Requests:	***Query/Result/Notifications Management Requests:***
FATHER-TERMINATING request	*QUERY* request
CHILD-TERMINATING request	*CANCEL-QUERY* request
NEW-CHILD request	*RESULT* request
NEW-FATHER request	*SNFE-REGISTER* request
ADD-SENSING-CAPABILITIES request	*CHECK-PE* request
SUB-SENSING-CAPABILITIES request	

Figure 3.6: The requests handled by the PE component

The PE components accept requests from all the other components of the system. We divide these requests in two categories (Figure 3.6).

The fist category contains requests submitted in order to inform the PE for a change in the hierarchy. In order to avoid race conditions requests of this type are handled one at a time (a FIFO queue is used):

FATHER-TERMINATING: This request is sent by the father-PE of PE1[1] when it terminates. The old father-PE sends the network address of the new father-PE. The PE1 holds the state about its new father and sends to it a NEW-CHILD-REQUEST.

[1] we refer to the PE which receives the described request as PE1

CHILD-TERMINATING: This request is sent by a child-PE when it terminates. When PE1 receives such a request, it checks if it has a child-PE with the sending PE's attributes and removes it from its child-PEs list. Additionally it updates its available sensing capabilities in case the terminating PE is the only which supported some of them. In that case PE1 sends a *SUB-SENSING-CAPABILITIES* request to its father-PE.

NEW-FATHER: This request is sent by a PE which enters the system and becomes a new father-PE for one or more PEs. The receiving PE updates its father-PE and sends any pending queries.

NEW-CHILD: This request is sent by PE which enters the system and becomes the child-PE for a PE. The receiving PE keeps state of its new PE, sends to it any queries that it can serve and possibly updates its (and its ancestors') sensing capabilities.

ADD-SENSING-CAPABILITIES: This request is sent by a PE when it becomes able to support one or more new sensing capabilities, either when a new SNFE is registered to it or when it has received an ADD-SENSING-CAPABILITIES request by one of its child-PEs. If the receiving PE doesn't already support the particular sense type, it updates its sensing capabilities and forwards the request to its father.

SUB-SENSING-CAPABILITIES: This request is sent by a PE when it becomes unable to support one or more particular sensing capabilities due to either one of its child-PEs termination or the receiving of a *SUB-SENSING-CAPABILITIES* request by one of its child-PEs. If the sending-PE was the only one between the receiving-PE's children that supported the particular sensing capability, the receiving PE updates its sensing capabilities and forwards the request to its father.

The other category contains requests for management of queries, results and other notifications:

QUERY: The receiving PE receives a *QueryObject* by another PE or an AC, keeps state of it (creates and stores a *PendingQueryObject*) and decides in which child-PEs and which SNFEs is going to forward it.

CANCEL-QUERY: The receiving PE receives the id of a query that must be cancelled. If it has previously received this query, it erases the corresponding QueryInPEObject and forwards the request to the appropriate PEs and SNFEs.

RESULT: This request is received either by a registered SNFE or by a child-PE. The receiving PE parses the received ResultObject and stores any available sensor data to the proper *QueryInPEObjects* of the *PendingQueries* list.

SNFE-REGISTER: As we will describe later, SNFEs can be added in a node of the system in a dynamic and transparent way. Every SNFE is registered to its PE by sending this request. The receiving PE holds state of its new SNFE and checks if it supports any new sensing capabilities. In that case it updates its sensing capabilities and informs all its ancestors.

CHECK-CONNECTION: This is a request serving failure check purposes. Whenever a PE or AC component, or the PER cannot access a PE, this request is sent. If the PE is up and running it has to respond to this check request. If it doesn't respond in a fair time space it will be marked as failed and is going to be properly deleted from the system.

CHECK-PE: This is a request serving failure check purposes and is sent only to the root-PE node. Whenever a departing PE or an AC component cannot access a PE the PER is informed and sends this request to the root-PE. If the root-PE receives the request it creates a FailureHandler thread that becomes responsible for running the departure protocol for the failed-PE.

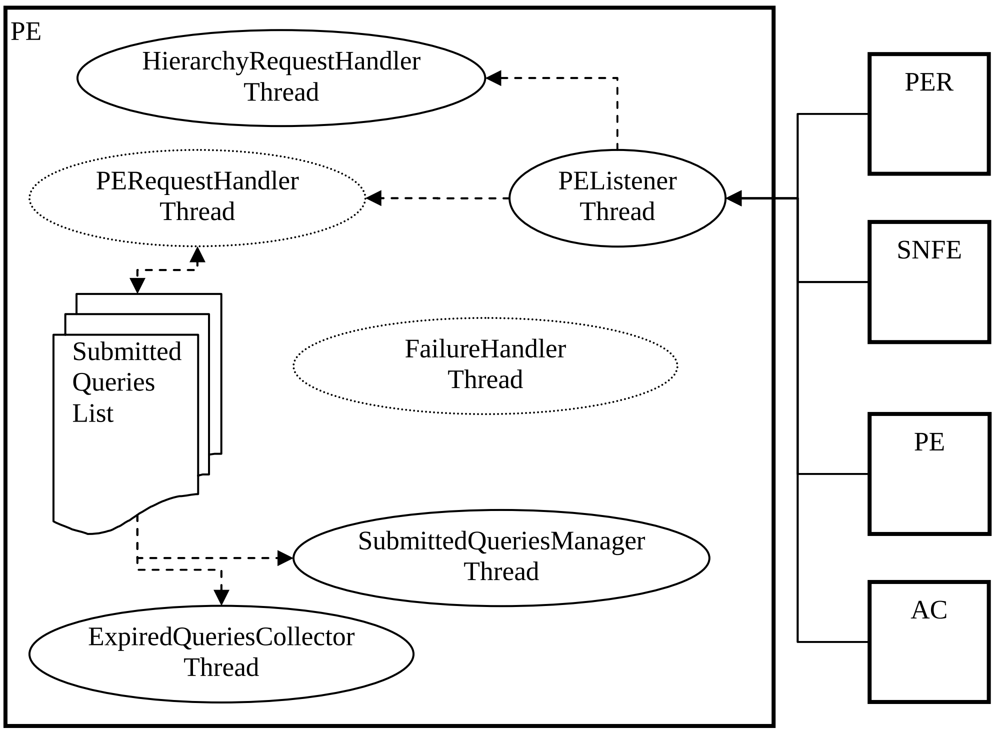


Figure 3.7: The PE component

The PE listens for requests in an IP and port defined by its owner. The *PEListener* thread listens to a socket binded to this network address. Whenever a hierarchy updating request is received, the *HierarchyRequestHandler* thread is awakened to handle it. Any simultaneous hierarchy updating requests are put in a FIFO queue and are handled one at a time. For every other request *PEListener* creates a *PERequestHandler* thread which handles it. More than one

PERequestHandlers can run simultaneously. There is also the *SubmittedQueriesManager* thread which checks the *SubmittedQueries* list in fixed intervals. If there is available sensor data for a submitted query, it is “packed up” in a ResultObject and it is forwarded to the AC which posted the query (if the PE is the first which received it) or to its father-PE. Another thread, *ExpiredQueriesCollector*, is also running through the *SubmittedQueries* list every few milliseconds and erases any expired queries. Finally, there may be *FailureHandler* threads which run the failure handling protocol in case another PE does not seem to respond properly.

3.5. The Sensor Network Front End (SNFE) Component

This component provides abstract query submission and result delivery functions that are independent of the technology used to implement a sensor network. Its role is to mediate between a concrete sensor network and a PE that represents it to the rest of the system.

In order to install an SNFE component to one of the system’s PEs a class implementing the *SensorNetworkGateway* Interface must be created. This class is going to support the Interface’s functionality by sending the messages to the actual physical Sensor Network. The functions which must be implemented are briefly described:

`void sendQuery (String queryID, long totalTime, int timeInterval, String senseType):` This function is responsible for the initiation of the query. The proper messages are sent to the physical sensor network in order to initiate a query for the given *total running time, time interval* and *sense type*. The *queryID* will be the query’s declarative, so it is advised to be stored locally in the implementing class.

`List<SensorValue> getValues(String queryID):` This function is used for acquiring data for the query with ID equal to *queryID*. It must return a list of *SensorValue* objects (the data structure is described in section 3.6) or null if no data exists.

`int cancelQuery(String _queryID):` This function is responsible for sending the proper messages to the sensor network which stop the execution of the desired query.

It is advised to use a data structure in the implementing class in which sensor data is going to be placed for every query. The available SNFE sensing capabilities are defined by the owner of the sensor network in String form. In a real case scenario sensor owners would agree on some basic sense type definitions. Different SNFEs binded on the same PE must be listening on different ports. For our tests, we created two classes implementing the SensorNetworkGateway Interface.

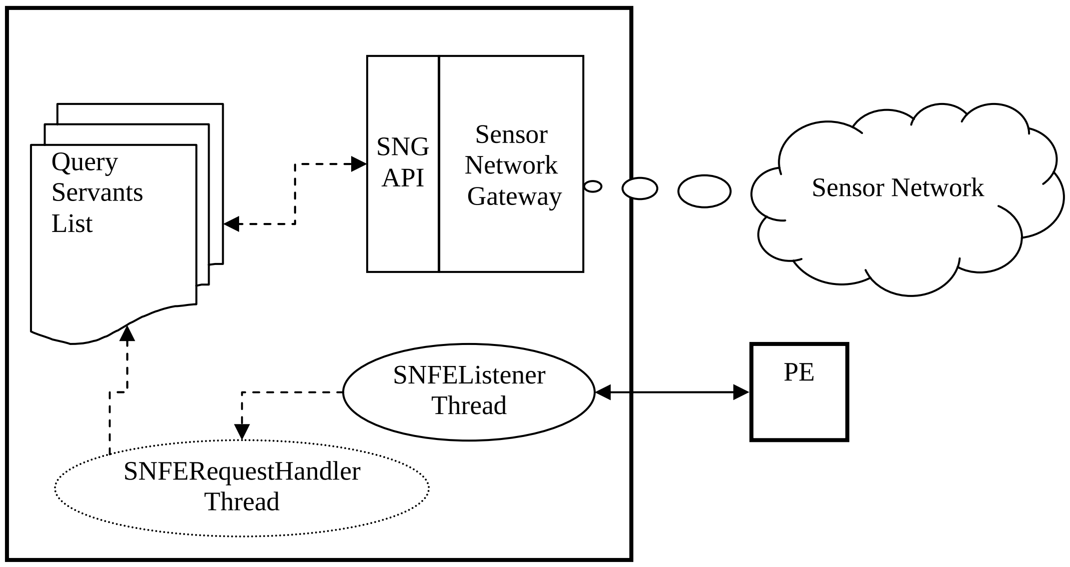


Figure 3.8: The SNFE component

Each SNFE component listens to a port defined by its owner for requests from the PE it is binded to. The *SNFEListener* thread listens for *QUERY* and *CANCEL-QUERY* requests. When such a request is received a *SNFERequestHandler* thread is created. If the request is a *QUERY* request, the received *QueryObject* is parsed and the proper query is forwarded to the physical sensor network through the *SensorNetworkGateway* object's *sendQuery* method. A *QueryServant* thread is created ,which periodically (in a period defined in the QueryObject) checks for sensor data by calling the *SensorGateway getResults* method. If the request received is a *CANCEL-QUERY* request, the SensorNetworkGateway's *cancelQuery* method is called. When a query expires, the corresponding *QueryServant* object is deleted.

3.6. Data Structures

Throughout our system we use data structures which describe basic entities, like queries, sensor data etc. These structures are implemented as Java Objects.

QueryObject:

```
String QueryID:         The query identifier in the PE hierarchy scope
long ACID:              The query identifier in the scope of the query-invoking AC
String senseType:       The sense type for which sensor data is requested
String aggregationType: The aggregation over sensor data (MAX|MIN|AVG|COUNT|NONE)
String targetArea:      The area from which sensor data is requested
long totalTime:         The quey run-time (in seconds)
int timeInterval:       The intervals in which sensor readings must be produced
ACInfo invokingClient:  The AC which initially posted the query
PEInfo sentFrom:        The PE (or AC) which sent the QueryObject is sent
```

Figure 3.9: The QueryObject fields

A QueryObject is sent either from an AC to a PE, when a query is submitted to the system by the application using the AC, or from a PE to another PE when a query is forwarded. The fields of a Query Object describe the actual query attributes.

The value of the *sentFrom* field is equal to the *invokingClient* value when the QueryObject is sent from an AC to the first query-serving PE.

The *QueryID* field is the identifier of the query and is unique for the PE hierarchy. It comprises of the ID of the first PE which received the query and a unique value given by the first PE.

QueryInPEObject:

```
QueryObject query:          The query attributes
Calendar startTime:         The query’s start moment
String endTime:             The moment the query expires
String linkedToID:          The id of the query from which sensor data are used
List<SNFEInfo> sentToSNFEs: The SNFEs the query is sent to
List<PEInfo> sentToPEs:     The PEs the query is forwarded to
List<ResultData> data:      The sensor data received for the query
```

Figure 3.10: The QueryInPEObject fields

A QueryInPEObject is created and stored in PE entity whenever a QueryObject is received. It holds the attributes of the query, any available sensor data ready to be forwarded, and query managing information like its start and end time. PE components keep lists of QueryInPEObjects in order to have a complete control over submitted queries. The QueryInACObject and QueryInSNFEObject structures contain the same fields and are used in the scopes of AC and SNFE components respectively.

ResultObject:

```
String QueryID:          The ID of the query for which the data is produced
List<ResultData> data:   The sent ResultData objects
long sequenceNumber:     Result data sequence
```

Figure 3.11: The ResultObject fields

A ResultObject is sent from a PE to its father-PE, or to an AC, whenever sensor data is available in the *data* list of a QueryInPEObject. The sequenceNumber field isn’t used by the system; it is filled in case the client application needs to use it.

ResultData:

```
Calendar timestamp:                    The moment the object is created
String area:                           The area of the producing SNFE
String senseType:                      The sense type of the data.
List<SensorValue> sensorValues:        The list of SensorValues
```

Figure 3.12: The ResultData fields

ResultData objects are created by SNFE components containing *SensorValue* objects along with query information and are stored in QueryInPEObjects ready to be forwarded to other PEs or ACs. The list of *SensorValues* represents the actual sensor readings.

SensorValue:

```
List<Double> value:       The sensor value in double format.
String stringValue:       The sensor value represented as a string.
String aggregationType:   The aggregation type of the value
double aggregationWeight: The number of sensor devices which produced the value
String sensorID:          The id of the sensor device which produced the value
```

Figure 3.13: The SensorValue fields

A sensor value object represents a sensor node reading or an aggregated value from many sensor nodes. The *aggregationWeight* field indicates the number of sensor devices from which the aggregated value was produced. If no aggregation is performed the weight is equal to 1. If it is supported by the used SensorGateway, the sensorID field contains the ID of the sensor device which produced the value.

PEInfo:

```
String area:              The area in which the PE node is located
String ip:                The ip address of the host PC
int port:                 The port in which the PE listens for requests.
String representativeFor: The areas for which the PE is representative.
String sensingCapability: The supported sensingCapabilities
```

Figure 3.14: The PEInfo fields

A PEInfo structure contains the information which describes the PE’s basic parameters and its network address.

The ACInfo is a similar data structure which holds the ip and port of an AC component.

The SNFEInfo is another similar data structure which holds the port of an SNFE component and its available sensing capabilities.

Chapter 4

Hierarchical Structure

In this chapter we present how the peers' hierarchy is constructed and updated when sensor networks join or leave the system. Initially we present the area model used and then we describe the PEs' join procedure and use some typical join scenarios in order to explain its basic principles. After that we describe the departure procedure and finally show how the peers handle failures occurring during hierarchy update actions.

4.1. Area Hierarchy Model

The world is divided into areas of interest. Areas are referenced via area names which are structured in a hierarchical fashion (in the spirit of DNS).

Each PE must be registered under an area name of this type, and is responsible for handling the queries targeted to this area, by querying its SNFEs (if available) and by forwarding the query to any other PEs that correspond to its sub-areas. For example a PE with name "*earth.eu.gr.thessaly.volos.port*" would be responsible for handling queries about the port of Volos, which in turn could rely on two additional PE's named "*earth.eu.gr.thessaly.volos.port.dock1*" and "*earth.eu.gr.thessaly.volos.port.d-ock2*".

A vital concept of our architecture is the area representation. Initially, when a PE joins the system, it is representative for the area defined in its configuration. However if this PE has some child-PEs which are not direct child nodes (i.e. their area strings are more than one levels longer than its area string), it becomes representative for the areas of the intermediate PE nodes which still haven't joined the system.

For example if a PE with area "*root.earth.europe.greece*" (PE1) exists in the system and a PE with area "*root.earth.europe.greece.thessaly.volos*" joins the system, the PE1 is representative not only for its area but also for "*root.earth.europe.greece.thessaly*". If another PE joins the system with its area variable equal to "*root.earth.europe.greece.macedonia.thessaloniki.airport*", PE1 becomes also representative for the area "*root.earth.europe.greece.macedonia*" and the area "*root.earth.europe.greece.macedonia.thessaloniki*". Finally if a node with area "*root.earth.europe.greece.macedonia*" joins the system, it becomes representative for the area "*root.earth.europe.greece.macedonia.thessaloniki*" and "*root.earth.europe.greece.macedonia*" and PE1 becomes again representative for the areas "*root.earth.europe.greece.thessaly*" and "*root.earth.europe.greece*".

The information about which areas a PE is representative for, is kept in the PE "*representativeFor*" variable which is stored in the PE's scope and in the PER database. The PE variable is updated each time a *NEW-CHILD* or *CHILD-DEPARTING* request is received. The PE lookup in the PER is performed using the representativeFor variables of the PEs.

4.2. Join Procedure

Every PE which participates in the architecture has to be registered during its startup process. The owner of the PE defines its host ip and port, the PE's area declarative and other internal parameters. The host information and the area declarative are stored in a PEInfo object. In order to be registered to the system, the PE sends a JOIN request to the PER and the PEInfo object which describes it.

The PER checks if the joining PE's area-String is well-formatted and decides which, already registered, PE is going to become the joining PE's father-PE. The PER also checks if there are PEs in the system which must become the joining PE's child-PEs and then sends to the joining PE an *OK* response, its updated PEInfo object and the PEInfo objects which describe its father-PE and its child-PEs (if any). If the PE receives an *OK* response, it gets its updated PEInfo along with the father-PE's and child-PEs' PEInfo objects.

Then it sends a *NEW-CHILD* request to its father-PE, receives any related queries and is registered in the scope of the father-PE as one of its child-PEs. Afterwards it sends a *NEW-FATHER* request to each one of its child-PEs, followed by its PEInfo and the child-PE's old father-PE PEInfo. The child-PEs register the joining PE as their new father and during the process the joining PE and its father-PE update their *representativeFor* and *sensingCapabilities* variables.

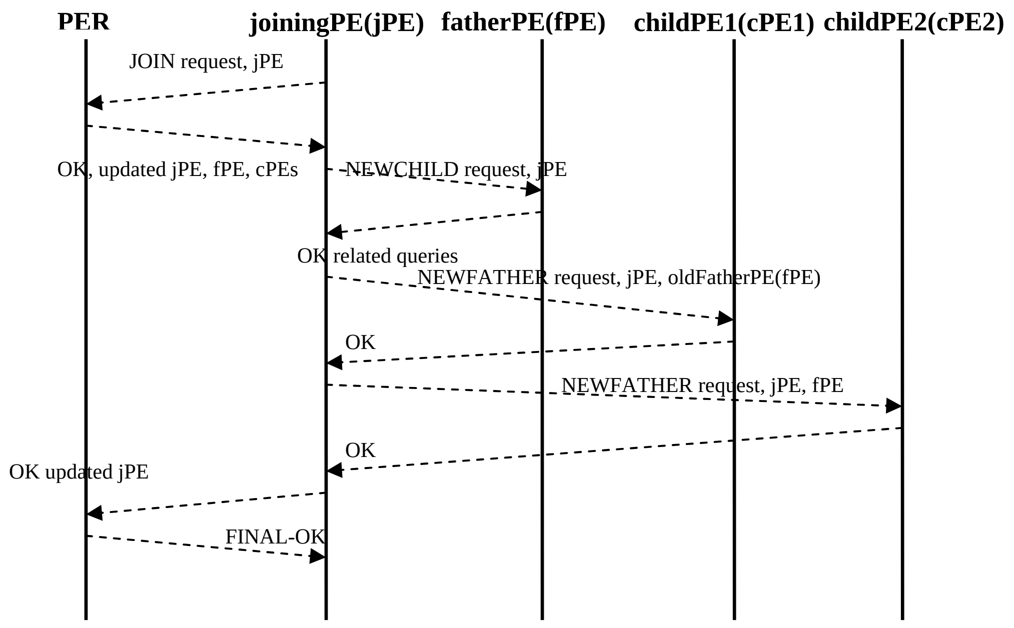


Figure 4.1: A typical join procedure without failures

Finally, if everything goes fine, the joining PE sends an *OK* response to the PER and its updated PEInfo (with the new *representativeFor* and *sensingCapabilities* variables), the PER receives and stores the information in its database, commits the changes and sends a *FINAL-OK* response to the joining-PE. After that the joining-PE is part of the architecture and is able to receive requests from ACs and other PEs. A typical join procedure is briefly described in figure 4.1 and in figure 4.2 we present the PE-side join pseudocode. Failure occurrences are not included in these figures as we analyze failure handling in the last section of this chapter.

PE.join():
```
send JOIN request, PEInfo to the PER
receive response from PER
if response=OK{
        receive updated PEInfo, father-PE PEInfo, child-PEs
        send NEW-CHILD request, my PEInfo to father-PE
        receive response from father-PE
        if response=OK{
                receive related queries (if any)
        }
        else{
                send FATHER-ERROR response to the PER
                prompt the PE owner to try to join later
        }
        for each child-PE{
                send NEW-FATHER request to child-PE
                send previous father-PE PEInfo, new father-PE PEInfo to child-PE
                receive response from child-PE
                if response=OK{
                        update representativeFor variable
                }
                else{
                        delete the child-PE from the child-PEs list
                }
        }
        send OK-FINAL response to the PER
}
else{
        prompt the PE owner to try to join later
}
```

Figure 4.2: The PE-side join-pseudocode

The PER is the component which has the complete view of the system, thus it is the one that decides which PE is going to be the joining PE's father-PE. When the PER receives a *JOIN* request it keeps a variable with the joining-PE's area (fatherArea). Then an iterative algorithm is executed in order to find a PE in the database, which is representative for fatherArea and will become the joining-PE's father-PE. At the end of each iteration, the fatherArea is reduced by one level.

If the PE (representingPE) is found on the first iteration (meaning that representingPE is already representative for the joining PE's area) it is checked if the representingPE.area is sub-area of joining-PE.area. If so, the joining-PE's father-PE is set to representingPE.father-PE and the representingPE.father-PE is then set to the joining-PE. Contrary, if representingPE.area is equal to joining-PE.area, representingPE becomes the father-PE of the joining-PE. After the father-PE is found, the joining-PE's child-PEs are discovered. Each child-PE found, is added in the child-PEs list.

If the joining-PE area is "lower" than the father-PE area more than 2 levels, the variable checkArea is created, containing the (father-PE.area.levels+1) levels of joining-PE.area (e.g. if the joining-PE area is *"root.earth.europe.greece.thessaly.volos"* and the father-PE area is "*root.earth.europe*", the checkArea variable is set to "*root.earth.europe.greece*"). The database is queried in order to find how many PE nodes have an area which is sub-area of checkArea and are not already the joining-PE's child-PEs. These PEs are not direct child-PEs of the joining-PE and we name them cousin-PEs. If the number of cousin-PEs is greater than the *maxCousins* variable (defined in the PER configuration) the joining-PE becomes representative for checkArea and the cousin-PEs are added to the child-PEs list.

The iterative algorithm stops either when the father-PE is found or the fatherArea variable becomes an empty string (in that case the joining-PE must be the first that joins the system and becomes the root-PE). Afterwards, the updated PEInfo object (with a possibly updated *representativeFor* variable) is sent to the PE and the father-PE and child-PEs PEInfo objects follow. Then, the PER waits for a response from the PE. If an *OK* response is received, the PER receives the updated PEInfo object from the joining-PE (with the *representativeFor* and *sensingCapabilities* variables possibly updated), the database is updated and the changes are committed. If an error occurs, the database changes are rollbacked.

```
onReceive JOIN-REQUEST from a PE (joining-PE):
if ! (area.startsWith root){
 abort join: malformed area
}
fatherArea= joining-PE.area
tries=0
while(!fatherArea.equals("")){
        tries++
        search the DB for a PE which is representative for fatherArea(repPE)
        if found{
                if (tries==1){
                        if(repPE.area is sub-area of joining-PE.area){
                                father-PE=repPE.father-PE
                                repPE.father-PE= father-PE
                                add repPE to the child-PEs list
                        }
                        else{
                                give a new unique area
                                                (newPE.area+=".#n" , n=0,1,2,3,....)
                                father-PE=repPE
                        }
                }
                else{
                        father-PE=repPE
                }
                update status in DB
                find child-PEs and add them to the child-PEs list
                if joining-PE.area.levels – father-PE.area.levels>1{
                        checkArea=newPE.area-(levelsFromFather-1);
                        find all the PEs "under" checkArea which are not the
                                                joining-PE’s child-PEs(cousin-PEs)
                        if number of cousin-PEs>=maxCousins {
                                newPE becomes representative for checkArea
                                add the cousin-PEs at the child-PEs list
                        }
                }
        }
        fatherArea=cutLevel(fatherArea,1)
}
send OK response,updated PEInfo,father-PE PEInfo and child-PEs  to the PE
receive response from PE
if response=OK{
        receiveupdated PEInfo (representativeFor variable updated)
        finalRegister of the joining-PE to the database
        send final OK response to the PE
        commit changes to the database
}
else{
        rollback the changes to the database
}
```

Figure 4.3: The PER-side join-pseudocode

The *cousin check feature* is vital in the tree hierarchy creation process. As we show in the evaluation chapter (section 7.6), if this feature isn’t used, we may have PE nodes with a large number of child nodes, or PE nodes with a very high tree depth. By using the cousin check feature we not only reduce the number of maximum child-PEs on a PE-node, but we also try to preserve the hierarchy information given by the area declaratives.

We could expand the cousin check feature by setting a limit of child-PEs per PE. This would lead to an even more balanced hierarchy where there would be complete control of the PE-nodes’ branch degree. However if we used a child limit scheme we would cause many “fictional” area declaratives which would not provide a clear view of the system status.

Additionally, the father-child associations defined by the PE owners would not be preserved**.** ***In order to respect the hierarchy relationships defined by the owners of sensor networks, we chose not to use a child limit feature.***

The first PE which joins the system automatically becomes the root-PE node of the architecture. Normally the root-PE must be owned by the system administrator and be located in the same host with the PER.

4.3. Join Scenarios

In this section we are going to explain some aspects of the join procedure, using some small scale scenarios. Initially we describe a very simple join scenario where the joining-PE becomes representative for the area defined in its area declarative and gets 2 child-PEs. Then we present a scenario explaining what happens if a PE joins the system and has the same area declarative as an already joined PE. After that we present 2 scenarios explaining the cousin check feature and finally we present a scenario where the area defined in a joining-PE's declarative is already represented by a PE of a lower area-level.

Join Scenario 1:

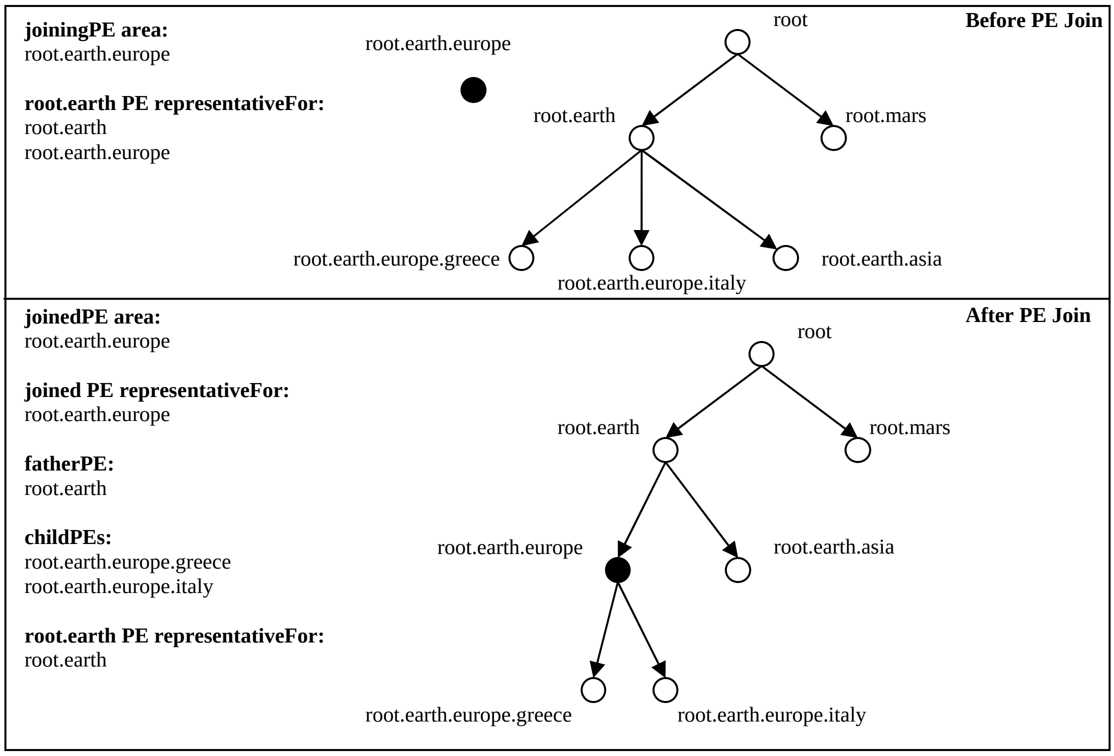


Figure 4.4 : Join Scenario 1

This is a simple join scenario. The PE nodes with area parameters "*root.earth.europe.greece*" and "*root.earth.europe.italy*", previously child-PEs of the "*root.earth*" PE node, must become child-PEs of the joining PE. The "*root.earth*" node, which was previously representative for the areas "*root.earth*" and "*root.earth.europe*" (because of its child-PEs), is now representative only for "*root.earth*", as the new-coming node is representative for its own area. The area of the new PE is one-level sub-area of its father PE node, so cousin check is not performed.

Join Scenario 2:

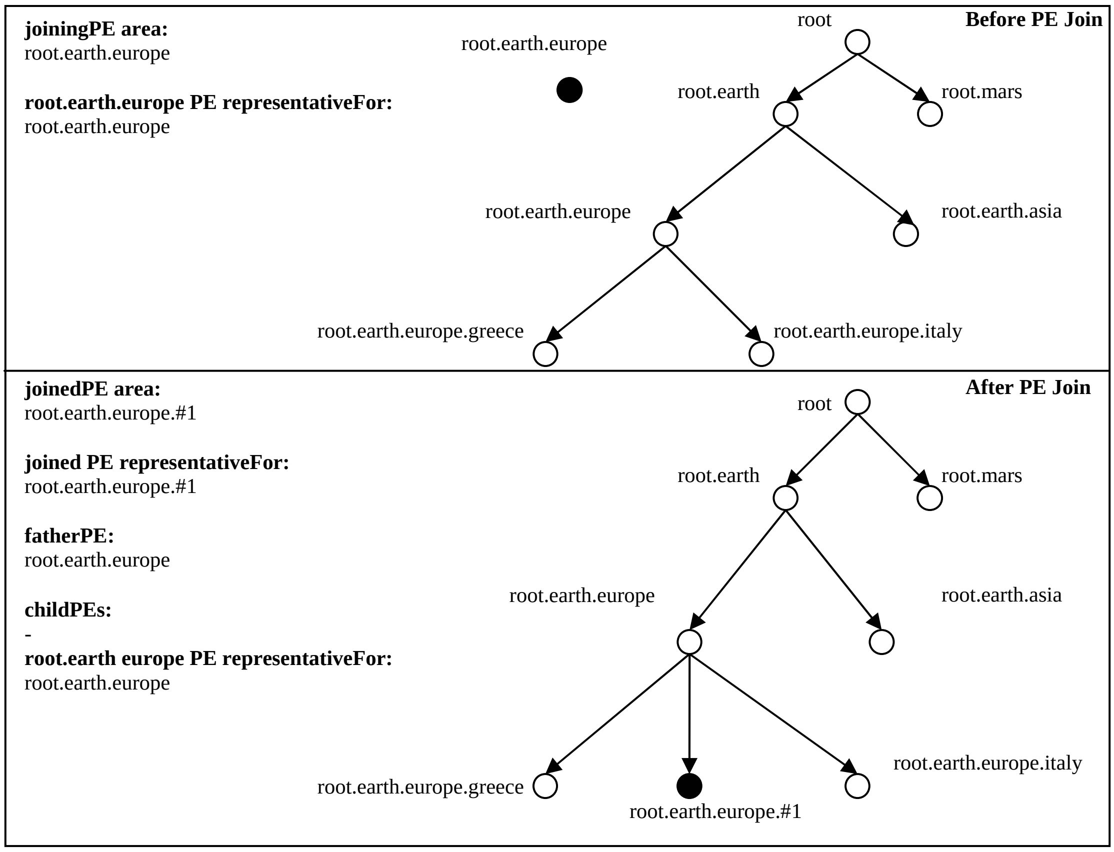


Figure 4.5: Join Scenario 2

This is a scenario showing what happens if a PE joins the system with the same area parameter as another PE. As we see there is already a PE in the system with area "*root.earth.europe*". So, when another PE with area "*root.earth.europe*" joins the system, its area parameter has to be altered and escalated per one level. Its new area parameter is "*root.earth.europe.#1*" (if another "*root.earth.europe*" PE joins, its area will become "*root.earth.europe.#2*" etc), it becomes a child-PE of "*root.earth.europe*" node and is representative for its altered area. Again, cousin check is not performed.

Join Scenario 3:

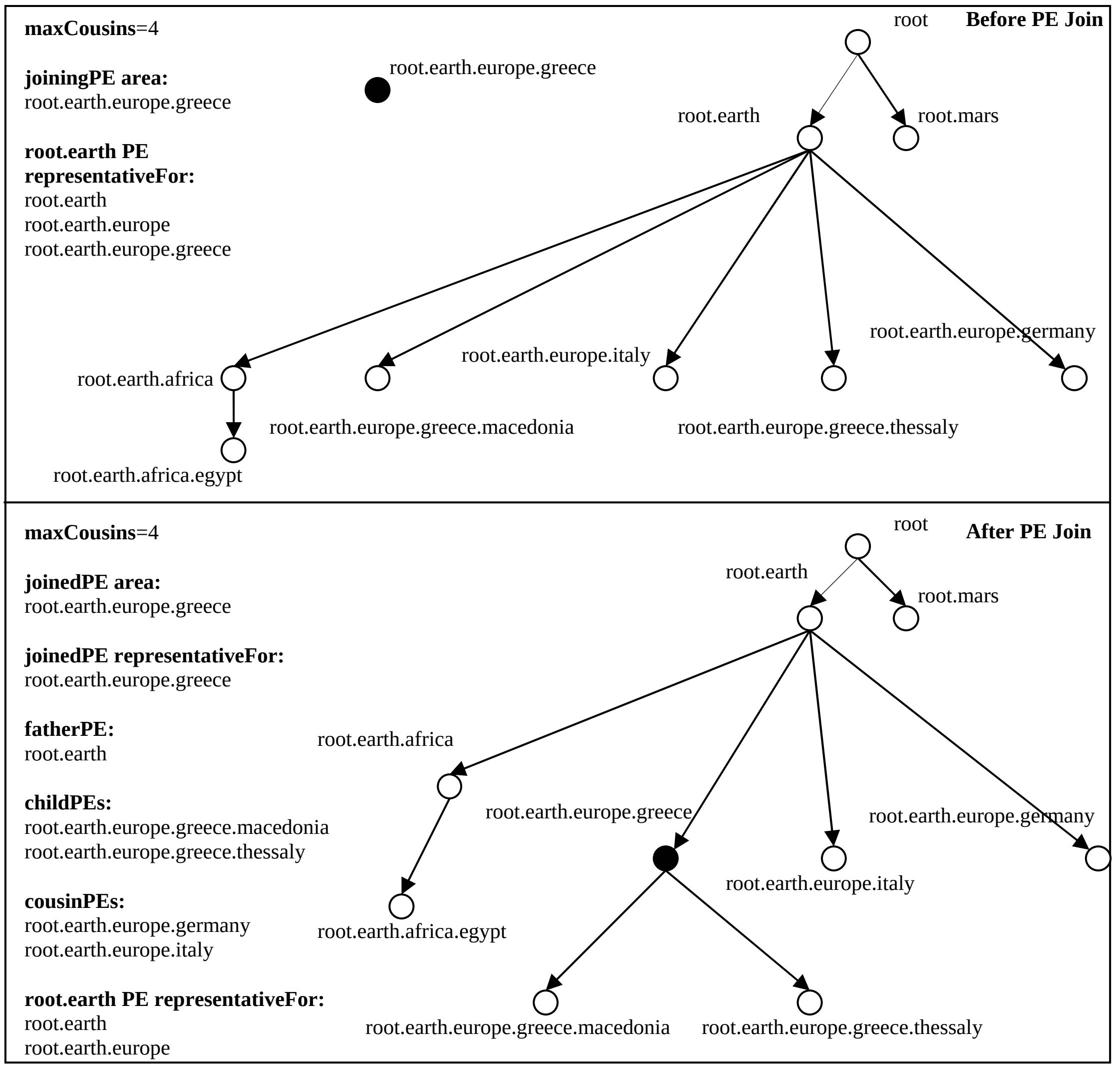
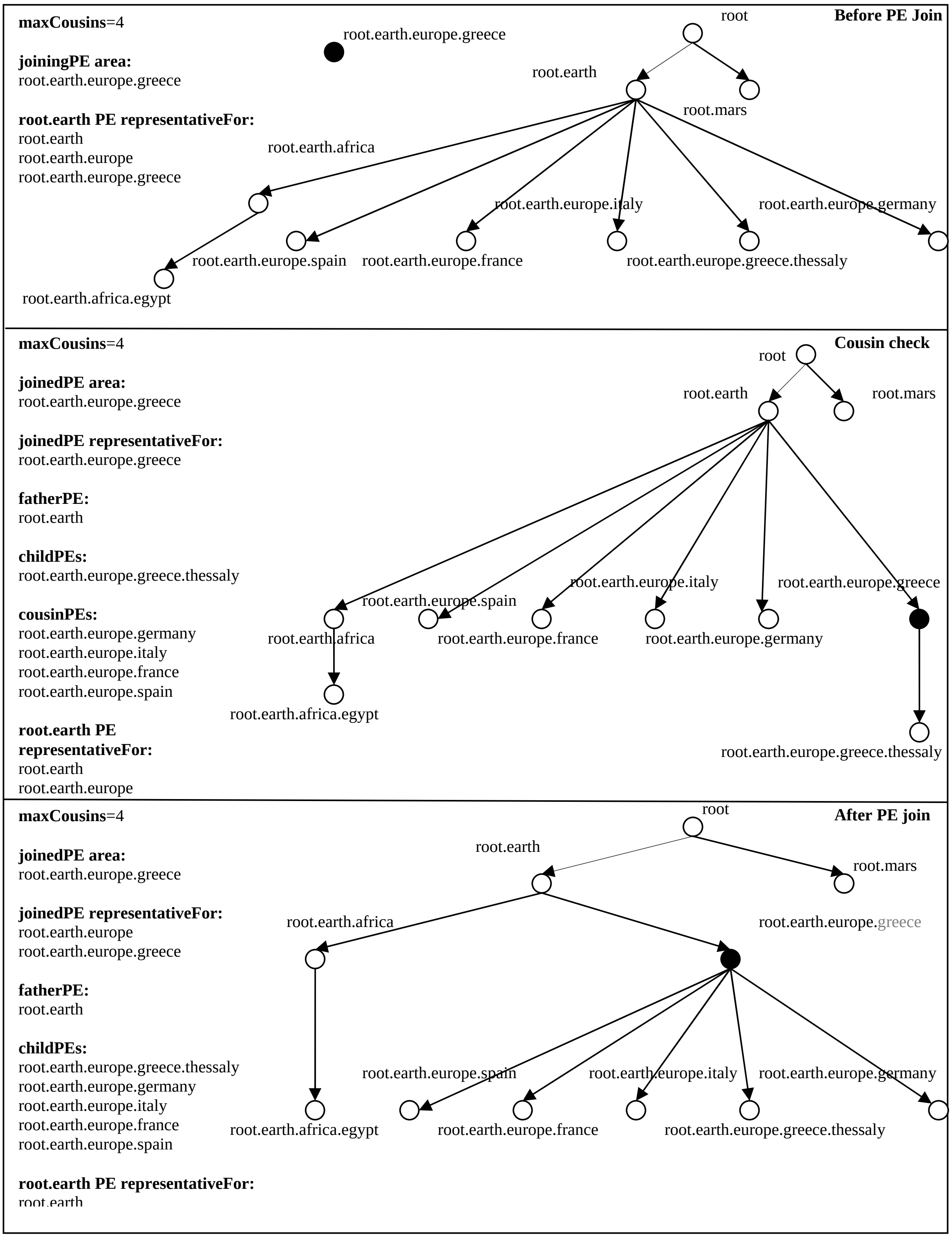


Figure 4.6: Join Scenario 3

In this scenario the area parameter of the joining PE node (“*root.earth.europe.greece*”) is 2 levels “lower” than its father-PE area (“*root.earth*”). In that case the cousin check scheme is performed. The cousin-PEs of the joining node, are the nodes with area parameters “*root.earth.europe.italy*” and “*root.earth.europe.germany*”, as they are not going to be its child-PEs, they are its father-PE’s child-PEs and their area parameters start with “*root.earth.europe*”. The numbers of cousin-PEs is smaller than maxCousins so the cousin-PEs remain child-PEs of “root.earth” and are not added in the joining node’s child-PEs list. The “*root.earth*” PE which was previously representative for “*root.earth*”, “*root.earth.europe*” and “*root.earth.europe.greece*”, is now responsible for “*root.earth*” and “*root.earth.europe*” areas as the joined PE is representative for its area “*root.earth.europe.greece*”.

Join Scenario 4:



maxCousins=4

joiningPE area:
root.earth.europe.greece

root.earth PE representativeFor:
root.earth
root.earth.europe
root.earth.europe.greece

maxCousins=4

joinedPE area:
root.earth.europe.greece

joinedPE representativeFor:
root.earth.europe.greece

fatherPE:
root.earth

childPEs:
root.earth.europe.greece.thessaly

cousinPEs:
root.earth.europe.germany
root.earth.europe.italy
root.earth.europe.france
root.earth.europe.spain

root.earth PE representativeFor:
root.earth
root.earth.europe

maxCousins=4

joinedPE area:
root.earth.europe.greece

joinedPE representativeFor:
root.earth.europe
root.earth.europe.greece

fatherPE:
root.earth

childPEs:
root.earth.europe.greece.thessaly
root.earth.europe.germany
root.earth.europe.italy
root.earth.europe.france
root.earth.europe.spain

root.earth PE representativeFor:
root.earth

Figure 4.7: Join Scenario 4

In this scenario the area parameter of the joining PE node (“*root.earth.europe.greece*”) is again 2 levels “lower” than its father-PE area (“*root.earth*”), so cousin check is performed. The cousin-PEs of the joining node, are the nodes with area parameters “*root.earth.europe.italy*”, “*root.earth.europe.france*”, “*root.earth.europe.italy*” and “*root.earth.europe.germany*”, as they are not going to be its child-PEs, they are its father-PE’s child-PEs and their area parameters start with “*root.earth.europe*”. The number of cousin-PEs is equal to maxCousins (4) so the joining PE becomes representative for “*root.earth.europe*” and its cousin-PEs become its child-PEs. The “*root.earth*” PE which was previously representative for areas “*root.earth*”, “*root.earth.europe*” and “*root.earth.europe.greece*”, is now representative only for “*root.earth*” as the joined PE is representative for “*root.earth.europe.greece*” and “*root.earth.europe*” areas.

We must denote that the sub-figure in the middle of the previous figure (Cousin Check) does not describe the system status at a particular moment. It shows the cousin-check feature execution.

Join Scenario 5:

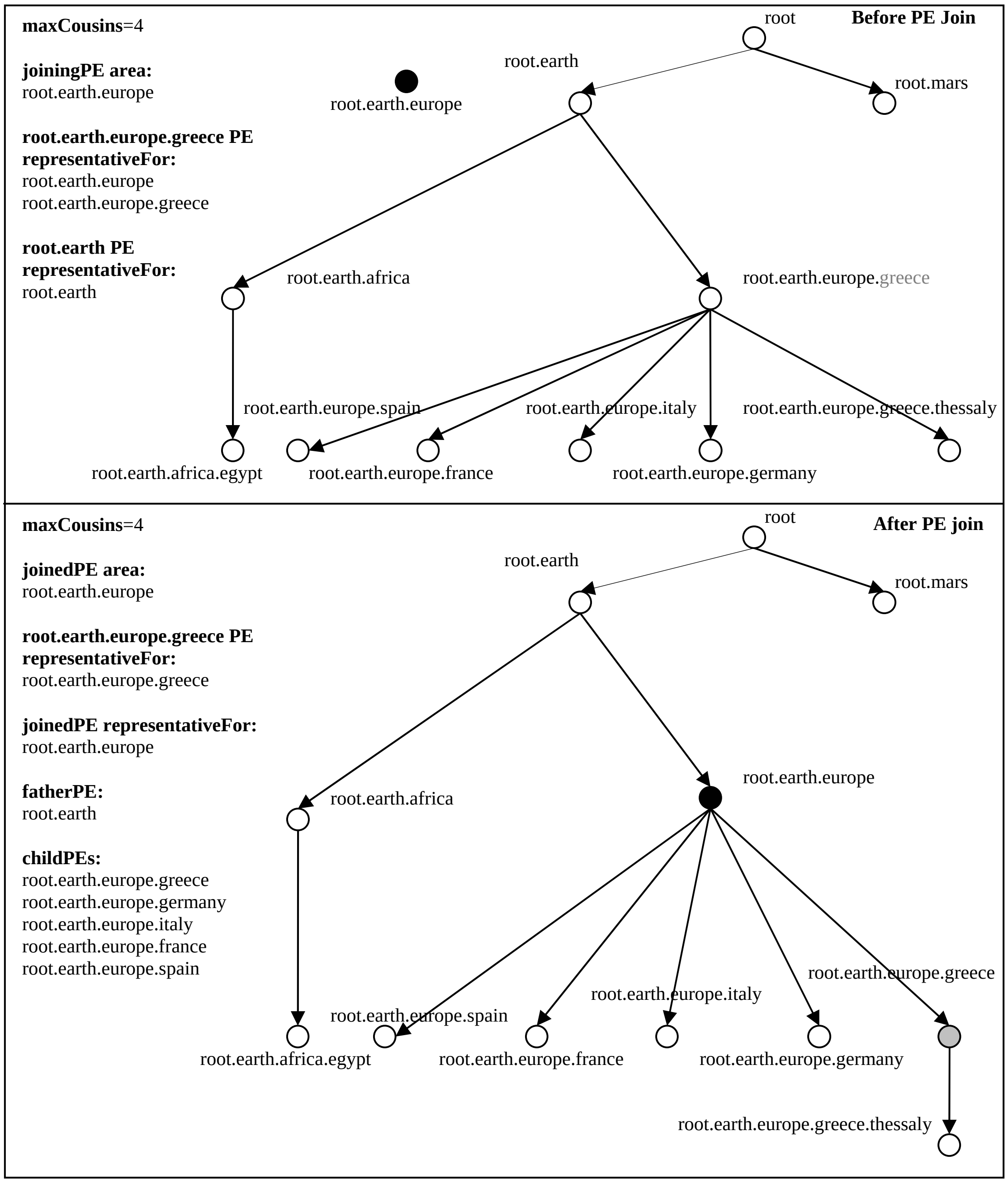


Figure 4.8: Join Scenario 5

In the previous scenario, the joining PE ("*root.earth.europe.greece*") becomes representative for an area ("*root.earth.europe*") which is in a higher level than its own area parameter. As it is described in the PER JOIN request handling pseudocode, if a PE joins later with area "*root.earth.europe*", it becomes representative for its area and the previous representative for its area becomes its child-PE. In figure 4.8 we present the scenario.

4.4. Departure Procedure

Whenever a PE needs to be terminated, it has to be properly unregistered from the system, thus to inform the PER, its father-PE and its child-PEs. The departure procedure is coordinated by the PER component in order to keep the hierarchy consistent in its database.

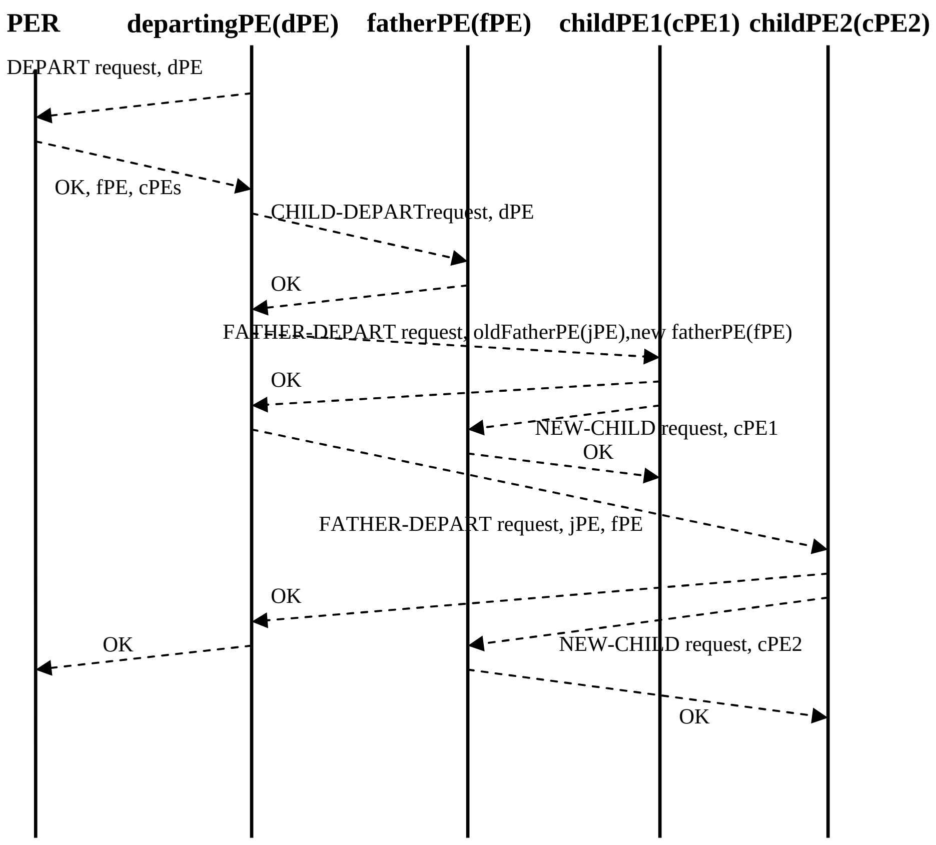


Figure 4.9: A typical departure procedure

Initially the departing-PE sends a *DEPART* request along with its PEInfo to the PER. The PER finds the departing-PE's hierarchy associations and sends to it its father-PE PEInfo and a list of its child-PEs PEInfo objects. After receiving these objects, the departing-PE sends a *CHILD-DEPARTING* request to its father-PE. The father-PE erases the departing-PE from its internal list of child-PEs and sends back an *OK* response. Then the departing-PE sends a *FATHER-DEPARTING* request to each one of its child-PEs along with its PEInfo and their new father-PE PEInfo. Each child-PE receives the old and new father-PE PEInfo objects and sends a *NEW-CHILD* request to their new father-PE followed by their PEInfo and any queries which were previously sent for the departing-PE. The new father-PE receives the child nodes information and any queries of the departing-PE and sends an *OK* response to its new child nodes. Finally the departing-PE sends an *OK* response to the PER which erases it from its database and commits all the changes made.

```
PE.depart():
send DEPART-REQUEST to PER
send my PEInfo
receive response from PER
if response=OK{
        receive father-PE PEInfo
        receive child-PEs PEInfo
        send CHILD-TERMINATING-REQUEST to father-PE
        for each child-PE{
                send FATHER-TERMINATING-REQUEST to child-PE
                send departing-father-PE PEInfo
                send new-father-PE PEInfo
        }
        send OK-FINAL to PER
}
```

Figure 4.10: The PE-side departure pseudocode

The PER sends the father-PE and child-PE PEInfo objects to a departing PE, because the departure protocol may be run on behalf of another, failed PE (we explain this feature in the failure handling section). Additionally, as it is seen in figure 4.9 the *child-PEs respond to the departing-PE before sending a NEW-CHILD request to their new father-PE* in order to let the terminating-PE successfully run the departure protocol without being prevented by other PE failures (e.g. a failure of the new father-PE). For the same reason, in order to avoid deadlocks, if a failure is detected by a departing-PE (either the father or a child node) the root-PE becomes responsible for checking the status of the PE that seems to have failed, at second time. The departing-PE executes with success the departure protocol and the PER database is properly updated.

4.5 Failure Handling

Failures may occur in various instances of hierarchy updating procedures. We deal PE failures as if a PE leaves the system but doesn't take the complete departure procedure. Our architecture's target is to detect any failed PE-nodes and remove them properly without invalidating the constructed hierarchy. An important parameter we consider is the fact that the departure procedure is used in failure handling and for that reason the departing protocol must terminate easily.

Various PE malfunctions may occur or be detected when a PE *joins* the system.

If the joining PE detects a failed child-PE, simply does not add it to its child-PEs list. It doesn't try to handle the child-PE failure on its own. It sends a *FAILURE* request to the PER and then, the PER sends to the root-PE a *CHECK-PE* request in order to run the failure handling protocol which we describe later in this section. The failure is handled by the root-PE and not by the new-coming PE because is not a stable part of the PE's hierarchy until the join procedure terminates.

If *the joining PE cannot connect with its father-PE*, it sends a *FAILURE* request to the PER component, which in turn makes the root-PE responsible for handling the PE failure. The join procedure is aborted, thus the new-coming PE does not become part of the system. The PE owner is prompted to try to start the component later, when the PE hierarchy will have been restored.

It is also possible that the new-coming PE fails during join procedure. *If the joining PE fails before receiving the father-PE and child-PE information*, the PER does not register it to its database, and rollbacks all the database changes. *If the joining PE sends a receive acknowledgement for the father-PE and child-PE information and afterwards does not send a FINAL-OK response to the PER*, the PER registers it to the system (it does not abort the join procedure immediately in order to avoid inconsistencies with PEs possibly updated before the joining-PE's failure) and then sends a *CHECK-PE* request to the root-PE. It is possible that one of the child-PEs detects its new father-PE failure when it sends a *NEW-CHILD* request to it. In any case the failure of the just-joined-PE is going to be detected (either by one of its child-PEs or by the root-PE) and it is going to be properly deleted from the system.

During *PE departure*, *if the departing PE cannot communicate with either the father-PE or one of its child-PEs*, it informs the PER component (which again sends a CHECK-PE request to the root-PE in order to handle the failure) and takes no further failure handling action. This is done in order to avoid deadlocks, because, as we describe later, the departing protocol is used in failure handling.

If a PE that received a FATHER-TERMINATING request cannot connect to its new father, it creates a FailureHandler thread in order to handle its new father-PE failure. After the failure is properly handled, the PE that detected the failure re-registers to the system (thus it runs the whole join procedure) in case other ancestor-PEs have also failed.

When the root-PE receives a *CHECK-PE* request from the PER, it first tries to communicate with the PE for which the *CHECK-PE* request was sent. If the respective PE doesn't respond, a FailureHandler thread is created in order to run the failure handling procedure.

```
FailureHandler thread(PEInfo notRespondingPE):
send FAILURE request to the PER
send the notResponingPE PEInfo
receive response from the PER
if response=OK-TERMINATE{
        depart(notRespondingPE)
}
```

Figure 4.11 The PE FailureHandler thread pseudocode

The FailureHandler thread is created taking as argument the PEInfo describing the not-responding PE node. Initially it sends a *FAILURE* request to the PER followed by the PEInfo of the not-responding PE (nrPE). If the FailureHandler receives an *OK-TERMINATE* response, it runs the departure protocol on behalf of the failed PE.

When the PER receives a FAILURE request it checks if the nrPE is in the database. If it finds it, it sends to it a *CHECK-CONNECTION* request. If the PE doesn't respond, the PER checks the type of entity that sent the *FAILURE* request. If the detecting entity is an AC or a PE that joins or leaves the system, the PER sends a CHECK-PE request to the root-PE in order to handle the failure and an OK response to the detecting entity. If the detecting entity is the root-PE or a PE that doesn't join or leave the system (thus it detected the failure during query forwarding or query canceling or result forwarding or sensing capabilities update) the PER flags the nrPE as faulty, keeps track of the PE which detected the failure and sends back to the detecting PE an *OK-TERMINATE*. If another PE detects the PE failure during the procedure, an *IGNORE* response is sent by the PER, indicating that the failure is handled by another PE. Other possible PER responses to *FAILURE* requests are the *NOT-REGISTERED* response, showing that the PE is no longer in the system and the *PE-ALIVE* response, indicating that the PE hasn't really failed.

```
onReceive FAILURE-REQUEST from a PE or AC (detecting entity)
receive the notRespondingPE PEInfo object
try to find the related PE (nrPE) in the database
if the nrPE doesn't exist in the database{
        send a NOTREGISTERED response to the detecting entity
        return
}
if the nrPE is marked by another PE(or AC)
        send back an IGNORE response
}
else{
        send a CHECKCONNECTION request to the nrPE
        if the nrPE responds OK {
                send back a PEALIVE response to the detecting entity
        }
        else{
                if the detecting entity is an AC, a joining PE or departing-PE{
                        send a CHECK-PE request to the root-PE
                        send an OK response to the detecting entity
                }
                else{
                        mark the PE with the detecting component's ID
                        send back an OK-TERMINATE response to the detecting entity
                }
        }
}
```

Figure 4.12 The PER FAILURE request handling pseudocode

Chapter 5

Query Submission and Management

In this chapter we present how the system manages the queries it receives from its end users. In section 5.1 we describe how PE nodes become aware of the sensing capabilities supported by their descendant PEs in order to avoid unneeded query forwarding. In the next two sections we present the query forwarding and query canceling mechanism. The way our system forwards sensor data to the users that requested them, is described in section 5.4. Finally, we describe how our system handles failures occurred during the execution of procedures studied in this chapter.

5.1. Awareness of Sensing Capabilities

An important feature of the architecture is the *Awareness of Sensing Capabilities*. Each PE is aware, at any time, of the sensing capabilities supported by the PE nodes in its sub-tree (meaning that the root-PE *sensingCapabilities* variable contains all the available sensing capabilities of the PE network). This feature is very useful, as it prevents unnecessary query forwarding to PE nodes which cannot support the desired types of senses. In section 7.5 we measure the effectiveness of the sensing capabilities' awareness scheme.

The locally available sensing capabilities of a PE may be declared by its owner in its configuration. Additionally, whenever a new SNFE component registers to a PE node, it is checked if it provides sensing capabilities which were not previously supported by the PE. If so, the *sensingCapabilities* variable of the PE is updated, an *ADD-SENSING-CAPABILITIES* request is sent to the PE's father-PE and an *UPDATE-SENSING-CAPABILITIES* request is sent to the PER. If a PE departs the system, its father-PE, after handling the *CHILD-TERMINATING* request, checks if it was the unique child-PE which supported one, or more, of its sensing capabilities. If so, the father-PE sends a *SUB-SENSING-CAPABILITIES* request to its own father-PE and an *UPDATE-SENSING-CAPABILITIES* request to the PER.

When a PE receives an *ADD-SENSING-CAPABILITIES* request it first checks if it already supports the new-coming sensing capabilities due to another child-PE or an SNFE. If the capabilities are not supported, it updates its *sensingCapabilities* variable, sends an *ADD-SENSING-CAPABILITIES* request to its father-PE and an *UPDATE-SENSING-CAPABILITIES* request to the PER. Finally it checks if there are any unsent queries with the newly supported sensing capabilities and forwards them to the PE that sent the *ADD-SENSING-CAPABILITIES* request.

```
on receive ADD-SENSING-CAPABILITIES request from a child-PE:
receive the new sensing capabilities (one or more)
if one or more of the new capabilities are not already supported{
        update sensingCapabilities variable
        send an ADD-SENSING-CAPABILITIES request to the father-PE followed by
                                            the newly supported capabilities
        send an UPDATE-SENSING-CAPABILITIES request to the PER
}
for each qeury in the SubmittedQueriesList{
        if the sense type of the query is one of the new sensing capabilities {
                forward the query to the child-PE which sent the request
        }
}
```

Figure 5.1: The *ADD-SENSING-CAPABILITIES* request handling

When a PE receives a *SUB-SENSING-CAPABILITIES* request it first checks if it is able to support the removed sensing capabilities through another child-PE. If the capabilities are not any more supported by any of its child-PEs, it updates its *sensingCapabilities* variable, sends a *SUB-SENSING-CAPABILITIES* request to its father-PE and an *UPDATE-SENSING-CAPABILITIES* request to the PER.

```
on receive SUB-SENSING-CAPABILITIES request from a child-PE:
receive the subbed senses (one or more)
if one or more of the new senses are not any more supported by other PEs{
        update senses variable
        send a proper SUB-SENSING-CAPABILITIES request to the father-PE followed by
                                                 the not supported senses
        send an UPDATE-SENSING-CAPABILITIES request to the PER
}
for each pendingQuery in the PendingQueriesList{
        if the sense type of the query is one of the new senses{
                forward the query to the child-PE which sent the request
        }
}
```

Figure 5.2: The *SUB-SENSING-CAPABILITIES* request handling

Obviously when the PER receives an *UPDATE-SENSING-CAPABILITIES* request, it updates the PE's record in the database.

The *ADD-SENSING-CAPABILITIES* and *SUB-SENSING-CAPABILITIES* requests are Hierarchy Updating requests and cannot be executed simultaneously with other such requests, in order to avoid hierarchy inconsistencies.

5.2. Query Forwarding

The architecture we propose uses the constructed hierarchy tree in order to deliver the user defined queries to the actual sensor networks. Users interact with an application (or service) which uses an AC entity. For each query a user submits, the AC-API's *sendQuery* method is invoked. The users define the desired sense and aggregation type, the target area (i.e. the area of interest from which sensor data need to be delivered), the total time the query must run and the time interval between sensor readings. The AC first communicates with the PER to find the PE which is representative for the query targetArea. If there isn't a representing PE for the desired area, a proper response is sent to the AC. The PEInfo describing the representative PE (which is the first PE of the system to receive this specific query) is sent to the AC entity. The AC forms a query object containing all the query parameters and sends it to the first-PE. The first-PE checks if it has a similar query running on its scope, from which it can share data and forwards the query to any child-PEs and SNFEs which can provide data for the desired sense type. The queries are forwarded to the lower-level PEs, following the area hierarchy. During the query delivery an acknowledgement scheme is applied in order to detect failed PE or SNFE components.

```
on receive QueryObject:
if Q1 comes from an AC define a unique queryID
create a QueryInPEObject with the received query's parameters (Q1)
if a similar query exists (Q2){
        if Q2.endTime is after Q1.endTime{
                if Q2.timeInterval>Q1.timeInterval{
                        update Q2 timeInterval on local SNFEs
                }
                Q1 is linked to Q2
                if Q1 came from an AC mark it as resultForwarding
        }
        else{
                cancel Q2 on local SNFEs it was sent
                Q1 becomes running
                send Q1 to all local SNFEs with Q1.senseType
                if an SNFE doesn't respond delete it and update sensing capabilities
                any queries linked to Q2 become linked to Q1
                Q2 is linked to Q1
                if Q1 came from an AC mark Q2 as resultForwarding
        }
}
else{
        Q1 status becomes running
        send Q1 to all local SNFEs with Q1.senseType
        if an SNFE doesn't respond delete it and update sensing capabilities

}
forward Q1 to all the child-PEs which support Q1.senseType
if an PE doesn't respond create a FailureHandler thread to handle its failure
store Q1 in the PendingQueriesList
```

Figure 5.3: The PE-side query forwarding pseudocode

In figure 5.3 we present the query forwarding algorithm as it is implemented in the PE component. When a PE receives a queryObject it first checks if it is sent by a PE or an AC entity. If the queryObject comes from an AC (meaning that the receiving PE is the first PE which received the query) a unique, per system, id is given to the query (a combination of the PE id and a unique, per PE, local id). Then the PE checks if there is a similar query running on its scope. If such a query exists, the PE has to decide which of the two queries is going to be the running query and which is going to be linked to the other's resources. If the old query's end time is after the new one's, the new-query is *linked* to the old and if the new query's time interval is smaller than the old's the updated time interval is sent to any SNFEs serving the old query. If the new query is received from an AC, it is marked as "*resultForwarding*". If the new query's end time is after the old one's the old query and any queries previously linked to it become *linked* to the new query. The updated end time is sent to any SNFEs serving the old query. The smallest between the queries' time intervals is adopted and if the new query is received from an AC, the old query is marked as "*resultForwarding*". If a similar query doesn't exist, the query object is marked as *running* and it is forwarded to any SNFEs which declared that they provide data for the desired sense type. Regardless of being *running* or *linked*, the received query object is forwarded to any PEs which declared that they (or one or more of its progeny-PEs) can provide data of the desired sense type. Finally the state of the query is stored in the form of the QueryInPEObject in the *SubmittedQueries* List.

The created Q*ueryInPEObject* has one of the *three levels of result forwarding*:

running: A query which is forwarded to all the available SNFEs and forwards its results to the father-PE or an AC. Other queries may be linked to it, sharing the result data it holds. A *running query* is also a *result forwarding query*.

linked and result forwarding: The query is linked to another query object, thus it is not additionally forwarded to the local SNFEs and uses any available data from the running query it is linked to. It has to forward result data to the father-PE or an AC.

linked and not result forwarding: The pending query is linked to another query, thus it is not forwarded to the SNFEs. Additionally it doesn't have to forward any result data to its father-PE because it is served by the query it is linked to.

A query is *similar* to another when they have the same sense and aggregation type. The total time, the time interval and the target area parameters may differ.

The *sensing capabilities' awareness feature* is very useful in that point as it prevents unnecessary query forwarding to sub-trees which cannot provide various types of sensor data at a particular time point. As we explained in section 5.1, if an SNFE with a new sensing capability is registered to a PE, the fact is publicized upwards the tree hierarchy. If unsupported queries exist in a PE and a new sense-update event is received from one of its child-PEs (matching with the inactive queries' sense type), they are forwarded downwards the hierarchy to request data from the new-coming sensor devices (represented by the previously mentioned SNFE component(s)).

We must denote that if a PE receives a query from an AC for a sense type that is not yet supported in its sub-tree, it properly informs the sending AC. The end user has the choice to either cancel the unsupported query or leave it in the PE scope in case the desired sense type becomes supported by a new-coming SNFE.

5.3. Query Canceling

The users of the system are able to cancel a query they sent to the system via the client application. The query canceling request travels through the hierarchy tree following the same route as the original query, in order to reach to all the PEs, and their SNFE components, which participate in the query.

```
on receive CANCEL-QUERY request:
receive the queryID(QID) of the query to cancel
if there is a query in the SubmittedQueries list(PQ) with PQ.quertID=QID {
        for each PEInfo in the sentToPEs list{
                forward the CANCEL-QUERY request to the PEInfo
                wait for response
                if not received within a fixed time space{
                        manage child-PE failure
                }
        }
        for each SNFEInfo in the sentToSNFEs list{
                send the CANCEL-QUERY request to the SNFE
                wait for response
                if not received within a fixed time space{
                        delete the SNFE from the SNFEs list
                        update available senses
                }
        }
        delete the query
        if there are any queries linked to it{
                find the one which lasts longer
                make it running query
                link the others to it
        send an acknowledgment response to the sending PE (or AC)
}
else{
        send a NO-SUCH-QUERY response to the sending PE (or AC)
}
```

Figure 5.4: The PE-side query forwarding pseudocode

As we show in figure 5.4, when a PE receives a *CANCEL-QUERY* request (CQ) it first checks if there is a *QueryInPEObject* with the CQ.queryID. If such a query exists it deletes it from its *SubmittedQueries* list and then forwards the *CANCEL-QUERY* request to the SNFEs and the PEs it had sent the cancelled query. The component waits for proper acknowledgements from the PEs and SNFEs in order to detect failures.

5.4. Result Forwarding

Result forwarding follows the hierarchy tree reversely to the query forwarding route. Thus, the result objects must be sent from every participating, per query, PE to the first PE that received the query (PE1) through the constructed hierarchy. PE1 finally sends the sensor data to the AC which originally posted the query. Result forwarding upwards the tree hierarchy is asynchronously performed. Result objects, coming from an SNFE or another PE, are received by the PERequestHandler thread. Then the related QueryInPEObject is found in the SubmittedQueries list of the PE, and the received sensor data is properly stored.

```
on receive ResultObject (RO):
for each QueryInPEObject(QIPO) in the SubmittedQueries list {
        if (QIPO.queryId==RO.queryID) OR
                        (QIPO.linkedToID== RO.queryID AND QIPO=resultForwarding){
                store the RO.resultData to the QIPO.data list
        }
}
```

Figure 5.5: The result receiving pseudocode

As it is described in figure 5.5, when a PE receives a ResultObject, it runs through all the queries in its SubmittedQueries list. If a QueryInPEObject having the same queryID with the ResultObject (the ResultObjects keep state of the query for which they are produced, on their creation time in the SNFE) is found, the ResultData of the QueryObject is stored in the pending query's result data list. The result data is also stored in pending queries which are linked to the query with RO.queryID and are marked as resultForwarding.

```
PendingQueriesManager thread:
while(true){
        sleep(checkTime)
        for each QueryInPEObject(QIPO)in the SubmittedQueries list {
                ...
                if (QIPO.resultForwarding==true) AND (QIPO.data is not empty){
                        if QIPO.lastTimeSent+ QIPO.timeInterval before currentTime{
                                create a ResultObject
                                send it to QIPO.sentFrom (PE or AC)
                                wait to receive a receive ack for a fixed time
                                if ack is received{
                                        clear the PQO data queue
                                }
                                else{
                                        if QIPO.sentFrom is not an AC{
                                                handle father-PE failure
                                        ]
                                }
                        }
                }
                ...
        }
}
```

Figure 5.6: The result forwarding pseudocode

Another thread running in the PE, the *SubmittedQueriesManager,* checks at specific time intervals all the pending queries of the PE, to find any available sensor data and forwards it either to the father-PE or to an AC component, from which the related query came from. Its functionality is described in figure 5.6. In fixed time intervals (defined in the PE configuration) the SubmittedQueriesManager thread runs through all the query objects in the SubmittedQueries list and, among other things, checks if they have any ResultData in their data list. If ResultData is found in a query object, and the last time a ResultObject was sent for this query was at least a timeInterval time space before the current time, a ResultObject is created containing the result data and the query's parameters and is sent to the fatherPE, or the AC which invoked the query. When an acknowledgement is received, the pending query's data list is cleared. The thread's access over the SubmittedQueries list object is protected to achieve mutual exclusion.

If an aggregation mode is requested by the user, the aggregation is performed on SNFE level, thus no aggregation is performed within the PEs of the hierarchy. Each SNFE aggregates the data, when it is created, and sends the aggregated value and its weight. If further aggregation is desired, it may be performed on client application level.

5.5 Failure Handling

PE nodes running the procedures described in this chapter may detect failed PE nodes. As we denoted in the previous chapter we deal PE failures as if a PE leaves the system but doesn't take the complete departure procedure.

At *sensing capabilities update functions* (either *ADD-SENSING-CAPABILITIES* or *SUB-SENSING-CAPABILITIES*) if the father-PE doesn't respond, a **FailureHandler**[2] thread is created in order to handle the detected failure. The sensing capabilities of the PE nodes, above in the hierarchy, will be updated when the hierarchy is fixed.

At *query forwarding,* if a PE forwarding a new query, cannot communicate with one of its child-PEs it keeps state of the fact that one of its child-PEs didn't receive a query and creates *FailureHandler* thread which has to run the departing protocol on behalf of the non responding PE after taking permission from the PER. In case the non responding child-PE hasn't really failed (for example if a very temporary network disconnection occurred), the query is sent properly to it by the PendingQueriesManager thread, which periodically checks for unsent queries and follows the same failure handling strategy.

[2] We describe the FailureHandler functionality along with the PER FAILURE request handling pseudocode in section 4.5 .

At *query canceling,* the exact procedure is followed as in query forwarding.

At *result forwarding*, if a PE cannot communicate with its father-PE, the PendingQueriesManager thread keeps any result data found in a QueryInPE object, and creates a *FailureHandler* thread in order to reinstate the tree hierarchy. Any result data collected during the failure handling procedure will be eventually forwarded by the PendingQueriesManager thread, when the tree hierarchy is fixed. If a PE cannot communicate with an AC component, it takes no action and just keeps the result data in its scope until the related query expires, in case the AC reconnects.

If an AC node cannot communicate with a PE (in order to send one of its queries) it sends a *FAILURE* request to the PER and a proper error value will be returned to the client application.

Chapter 6

Implemented Sensor Gateways

In order to test the system we proposed, we created a simple Client Application using the AC component and we also developed two classes implementing the *SensorNetworkGateway* Interface. The first is a very simple sensor network simulator (*SimulatingGateway*), which produces random values.

The other one is a real sensor network gateway that connects to a network of sensing devices (***Smart-Its*** [2]) and acquires real time data (*SmartItsGateway*).

In this chapter we present the real sensor motes' platform used in our tests and then we describe the SmartItsGateway component.

6.1. The Smart-Its Platform

In order to create and deploy a simple sensor network with a corresponding Sensor Gateway, we used Smart-Its sensor motes. Smart-Its are simple sensing devices that are able to produce sensor data of various types and send them over the air to other Smart-Its devices or a PC. The platform's key building block is the particle (Figure 6.1.a), a board equipped with a microcontroller, a short range wireless communication interface, and a conventional battery. The particle is extended by plugging different sensor boards on it (Figure 6.1.b). In order to collect data from a set of Smart-Its nodes and store them to a PC, a USBBridge (Figure 6.1.c) or an X-Bridge (Figure 6.1.d) must be used. These devices are similar to a particle device except that they have an extra communication interface and convert Smart-Its packets into UDP packets and vice versa.

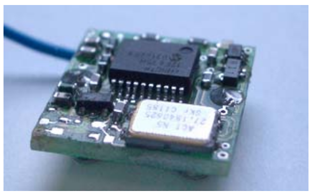


Figure 6.1.a : The Smart-Its particle

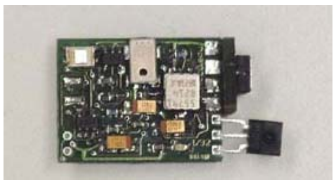


Figure 6.1.b : A sensor board

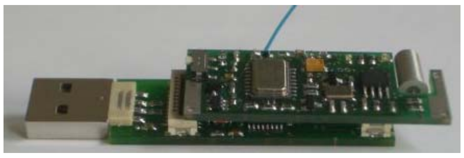


Figure 6.1.c : The USBBridge

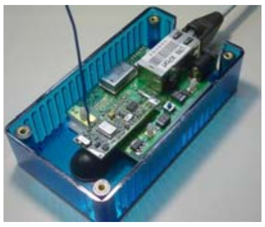


Figure 6.1.d : The XBridge

Figure 6.1: The Smart-Its platform

A custom runtime system is preinstalled in the particles. Each packet is broadcast over the air and is received by all nodes in range. Packets are filtered on receive, based on a subject interest mask.

In our setup we used a particle with the default Particle Software installed on it, a sensor board and a USBBridge. In order to collect data to the PC that hosted the PE and SNFE components, we used the USB PC Software provided by the Smart-Its web site[2]. We also used the Java libparticle API in order to send and receive messages from the *SmartItsGateway* Class.

6.2. The SmartItsGateway Component

The SmartItsGateway component is a Class that implements the *SensorNetworkGateway Interface*. We will describe the implemented methods:

`void sendQuery (String queryID, long totalTime, int timeInterval, String senseType):`

This function initiates the query. When invoked a Query object(containing the start moment, the time interval, the end moment, the queryID, the sense type and an empty list of SensorValue objects) is created and stored in the RrunningQueries list. If no query with the same sense type runs in the gateway, the proper *libparticle* methods are used in order to create a SmartIts packet which will be sent to all the particles in range and make them produce data for the requested sense type. If another query is running on the gateway with the same sense type and greater time interval, a packet is sent that updates the query time interval in the particles.

`List<SensorValue> getValues(String queryID):` This method returns available SensorValue objects for a Query object with ID equal to *queryID* from the RrunningQueries list. Null is returned if there are no data available.

`int cancelQuery(String _queryID):` When this function is invoked, the Query object with the corresponding queryID is erased from the *RunningQueries* list. If no other query with the same sense type is running on the gateway, the proper *libparticle* methods are used in order to create a SmartIts packet which will be sent to all the particles in range and force them to stop producing data for the cancelled query's sense type.

Apart from the Sensor Network Gateway API, some internal entities are implemented:

The *ResultListener* thread uses the *libparticle* library and listens (through the USBBridge) for result packets. Each time a result packet is received a *PacketHandler* thread is created that clears the packet data, transforms them to SensorValue object(s) and stores them in the proper Query object.

There is also the *GatewayManager* thread which runs through the *RunningQueries* list, erases any expired queries and sends query-terminating packets if needed. Additionally it re-sends query-setup packets in fixed time intervals in case new particles come in range.

Chapter 7

System evaluation

In this chapter we present some experiments we conducted that either explain some of our design options or show the efficiency gain we get from the self-organized peer hierarchy. Initially we present the simulator component we created in order to evaluate our system and describe its basic parameters and the variables it measures. Afterwards we present experiments evaluating the cousin-check, sensing capabilities' awareness and query multiplexing features.

7.1. The Simulator Component

In order to evaluate the system, we created a simple simulator with which we measured the system performance. The simulator component creates a number of fictional PE nodes, simulates the join protocol for each one of them and forms their hierarchy tree. Then it creates some fictional queries and simulates their processing. Each PE is described by its area and available sensing capabilities and each query is described by the requested sense type, the target area and the (fictional and random) end time. Variables which describe the system performance are probed during the simulation.

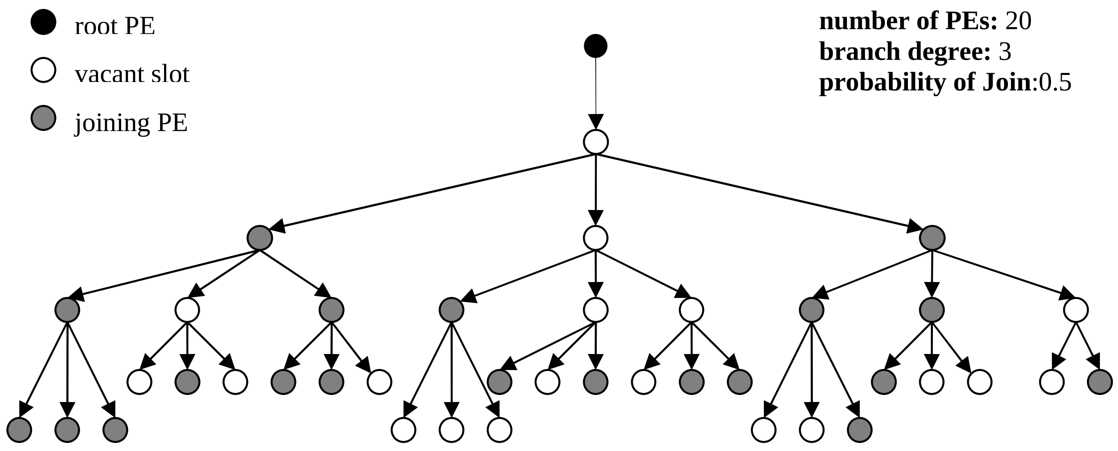


Figure 7.1: A small-scale simulation scenario

7.2. The Simulation Parameters

The simulator parameters we describe in this section can be set during the simulator configuration:

number of PEs: The number of PE descriptions that the area creator is going to produce.

branch degree: In order to simplify the simulation, we consider the environment in which the simulated architecture is deployed, as a set of areas which are organized in a tree hierarchy. If we take the earth as an example environment, the hierarchy levels may comprise of a continents level, a sub-continents level, a countries level, a regions level, a prefectures level, a cities level and so on. This attribute is the maximum number of the sub-areas each area has in the next layer.

probability of join(pJoin)**:** This attribute is the probability that an area is going to participate in the system by deploying a PE. Areas are created according to the "branch-degree" attribute and for each area a PE is created with the defined *probability of join*. The procedure stops when "number of PEs" PE descriptions are created.

available sensing capabilities: The totally available number of sensing capabilities in the simulated set of PEs.

maximum sensing capabilities per PE(maxSC)**:** The number of sensing capabilities each PE gets is random and is between 0 and *maxSC.*

number of queries**:** The number of queries simulated.

join method**:** The method by which the PE nodes join the system, with the cousin check feature either activated or deactivated.

maxCousins: The homonymous variable of the cousin-check feature (if activated).

processing mode: The way the random queries are processed, either with the sensing capabilities' awareness scheme and query multiplexing enabled or not. Three processing modes are available: 1 is without sensing capabilities awareness and without query multiplexing, 2 is with sensing capabilities awareness enabled and without query multiplexing and 3 is with both features enabled.

7.3. The System Status Variables

In this section we describe the system variables that are measured by our simulator and are used to evaluate the architecture.

In order to analyze the PE joining procedure the simulator measures:

maximum number of child-PEs(on some PE-node)**:** The simulator tracks the number of child-PEs on every PE-node. This variable is the number of child-PEs in the most child-loaded node (the node with the most child-PEs).

maximum tree depth: The maximum depth of the constructed tree of PE-nodes.

In order to analyze query processing the simulator measures:

total QueryInPE objects (TQ)**:** The number of submitted queries in all the PE nodes of the system. As we showed in chapter 5 whenever a QueryObject is received, a QueryInPEObject is created and stored in the scope of the PE.

unsupported QueryInPE objects (UQ)**:** This variable shows the unsupported query objects in the set of simulating PEs. In chapter 5 we denoted that if sensing capabilities' awareness is enabled, a query that is received by a PE which cannot support the requested sensing capability (meaning that neither its SNFEs nor its descendant-PEs support it), is not forwarded to any of the PE's SNFEs or PEs and is stored internally in case the required sense type becomes available by a new SNFE. These query objects are named as unsupported. No data are received from, or forwarded by, such query objects.

supported QueryInPE objects (UQ)**:** The set of *total QueryInPE objects* that are *not unsupported.*

linked QueryInPE objects(LQ)**:** The number of QueryInPE objects which are linked to another pending query. As we described in chapter 5 linked query objects are the ones that aren't forwarded to SNFEs and reuse data from other queries with the same sense and aggregation type attributes (*running QueryInPE* objects).

running *QueryInPE* objects(RQ): The number of query objects which are not linked to another and receive data from SNFEs and PEs to which were previously forwarded.

result forwarding QueryInPE objects(RFQ)**:** The number of QueryInPE objects which are marked as result-forwarding, in all the nodes of the system. As we described in section 5.2, the set of result-forwarding QueryInPEs comprises of all the running QueryInPE objects and a subset of the linked QueryInPE objects, which have to send result data to an AC or a PE regardless if they primarily use or reuse sensor data. This variable is an important metric of the system status as it shows the query objects that produce network traffic amongst the PE components.

***QueryInSNFE objects(SNFEQ)*:** The number of SNFE query objects (thus the actual queries that keep the sensor devices busy). This is another important performance metric as it shows the physical wireless sensor networks' load.

7.4. Cousin Check Scheme Evaluation

In order to evaluate the cousin check scheme that is used whenever a PE joins the system, we simulated the join procedure of a set of PEs, first without the cousin check feature and then with this feature enabled.

During the first testing simulations we noticed that if the system doesn't perform cousin check, the root-PE of the system has too many child-PE nodes. If such a thing happens to a real system, the root-PE would become representative for many areas, thus receiving a large number of queries from AC components and hierarchy requests from PE components, which could cause a network and memory overload in its host PC. In order to avoid child-overloaded nodes, we introduced the cousin check scheme and in order to measure the improvement we get from it, we conducted experiments for various sets of PEs with different *numbers of PEs* and *probabilities of join* measuring the maximum number of child-PEs in the constructed hierarchy for the two different join modes (cousin check enabled or not).

In the next page we present 4 graphs from an experiment showing *that if cousin check scheme is not used, the smaller the pJoin is, the greatest maximum number of child PEs we have in the set of PEs*. Contrary, when cousin check is used, the maximum number of child-PEs remains relatively stable.

The varying simulation parameter in each one of the experiments presented on the next page, is the *probability of join (pJoin)*. The parameter which is different among the experiments is the *number of PEs* (500, 1000, 2000 and 4000 nodes). The *branch degree* parameter is set to 7 to all 4 experiments, as it is a fairly plausible value describing a real environment. The *maxCousins* parameter is set as equal to *branch degree* (7) (simulations with greater *maxCousins* showed smaller improvements). The simulation is run in both PE join modes.

```
number of PEs:                        500/1000/2000/4000
branch degree:                        7
probability of join(pJoin):           varying (0.1:0.1:1.0)
available sensing capabilities:       unconcerned parameter
maximum sensing capabilities per PE:  unconcerned parameter
number of queries:                    unconcerned parameter
join method:                          cousin check inactive / cousin check active
processing mode:                      unconcerned parameter
maxCousins:                           7
```

Figure 7.2.a:The simulation parameters of the following graphs

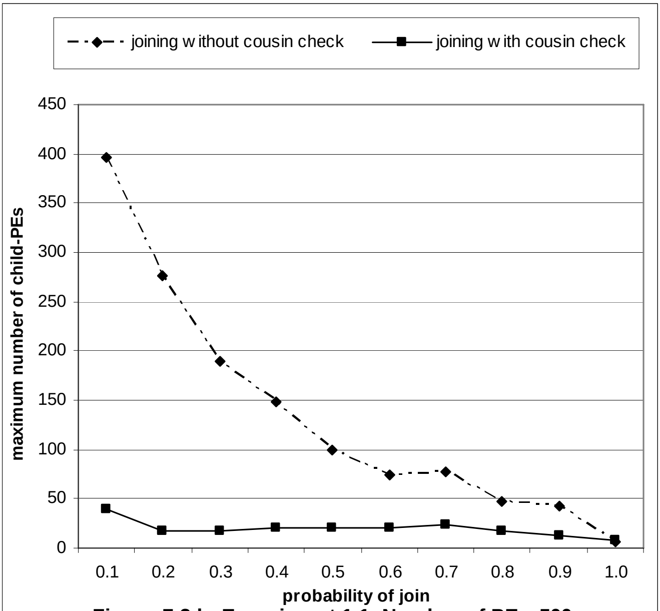


Figure 7.2.b: Experiment 1.1: Number of PEs: 500

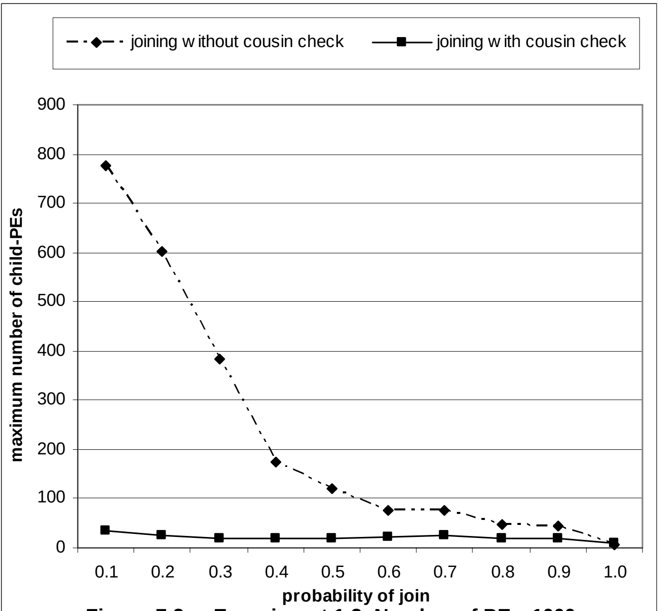


Figure 7.2.c: Experiment 1,2: Number of PEs: 1000

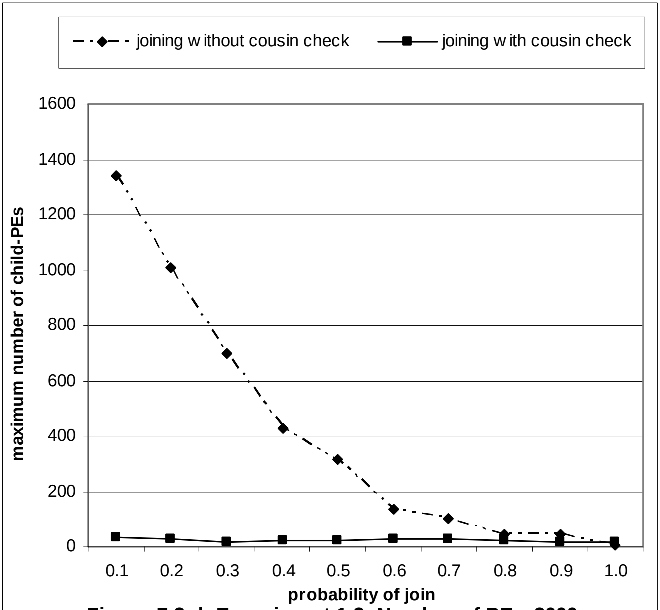


Figure 7.2.d: Experiment 1.3: Number of PEs: 2000

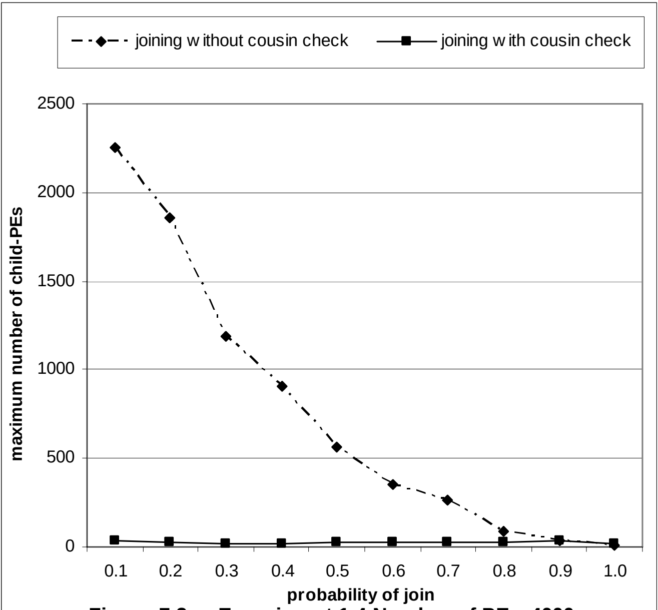


Figure 7.2.e: Experiment 1.4:Number of PEs: 4000

Figure 7.2 : Experiment 1 (Cousin check Evaluation)

As we see in the previous graphs, the maximum number of child PE nodes rises excessively when we don't use the cousin check scheme and we have an environment with small join probability. However, when cousin check is performed, the maximum number of child-PEs is much smaller and remains at the same order of magnitude regardless of the probability of join. We also denote that the maximum number of child-PEs is highly depended on the number of PEs parameter if cousin check is not used.

The maximum depth among the PE nodes of the system is also a good metric for the hierarchy quality, but in simulations for number of PEs of this scale we don't have big differences.

7.5. Sensing Capabilities' Awareness Scheme Evaluation

In section 5.1 we described the sensing capabilities' awareness scheme used in our system, thus how each PE node becomes aware of the sensing capabilities supported by all the PEs in the sub-tree below it. If this feature is not used, when a query arrives to the PE which is representative for the defined target area, it has to be forwarded to all the PEs which are hierarchy descendants of the first PE, regardless of the fact that maybe the desired sense type is not supported by any of it. Obviously this can cause many unnecessary query requests and unsupported query objects in the PE components.

We conducted simulations in order to measure the gain we get from the adoption of the sensing capabilities' awareness scheme and we present a representative example which refers to a PE and query set which can be mapped to a real-life sensing environment scenario.

In the experiment presented on the next page, the join probability parameter varies from 0.1 to 1 (with a step of 0.1). The *branch degree* parameter is set to 7 and so does the maxCousins parameter. 10000 PEs are created and each one of them has a random number of sensing capabilities between 0 and 2 (*maxSC*=2) choosing from a set of 8 available capabilities. There are 200000 queries created choosing a target sense type from the available sensing capabilities set. The simulation is first run with sensing capabilities' awareness feature deactivated and then with the feature activated. Query multiplexing is not enabled in both cases.

In the first graph we show *the percent of the total QueryInPEs objects, with sensing capabilities' awareness enabled, which are unsupported QueryInPEs objects*. In the second graph *we show the number of total QueryInPE objects for each probability of join with the sensing capabilities' awareness feature either activated or not.*

```
number of PEs:                         10000
branch degree:                         7
probability of join(pJoin):            varying (0.1:0.1:1.0) with a step of 0.1)
available sensing capabilities:        8
maximum sensing capabilities per PE:   2
number of queries:                     200000
join method:                           cousin check activated
processing mode:                       1(sensing capabilities awareness deactivated)/
                                       2(sensing capabilities awareness activated)
maxCousins:                            7
```

Figure 7.3.a:The simulation parameters of the following graphs

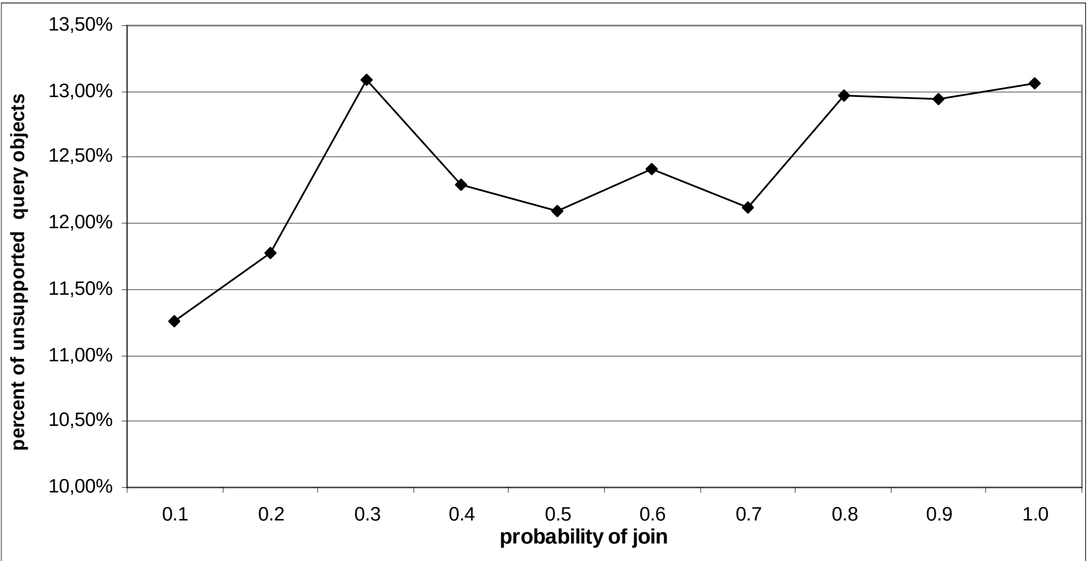


Figure 7.3.b: Percent of Unsupported QueryInPE objects with sensor awareness enabled

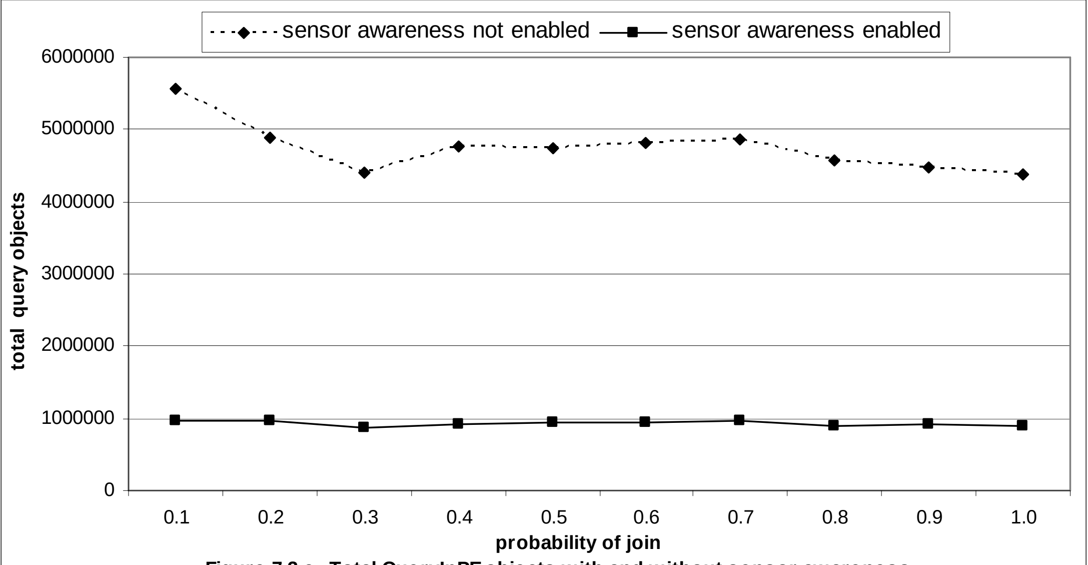


Figure 7.3.c: Total QueryInPE objects with and without sensor awareness

Figure 7.3 : Experiment 2 (Sensing Capabilities' Awareness feature evaluation)

In graph 7.3.b we show the percent of the total queries, with sensing capabilities' awareness, which are unsupported (i.e. query objects whom sense-types are not yet supported by the containing PEs' sensing capabilities)

The unneeded forwarding of the unsupported queries, if sensing capabilities' awareness is not enabled, causes a bigger number of total QueryInPE objects as we show in graph 7.3.c. As we see in this graph, the number of the total QueryInPE objects in the system is much smaller (about 80-83% in this PE/query set) when sensing capabilities' awareness is used, than when we forward all the queries, regardless of the supported capabilities. The number of excessive query objects not only shows the unnecessary overload of the PE nodes, but it also implies the unneeded query requests sent from PE nodes to descendant PEs which cannot support the desired sensing capabilities.

The fact that the number of total query objects for both processing modes, does not range a lot due to different join probabilities, shows that the constructed hierarchy is efficient as the system seems to work well regardless of the rate the environment areas decide to join.

7.6 Query Multiplexing Evaluation

In section 5.2 we described how our system supports query multiplexing and result data reuse. Whenever a new query request arrives to a PE node, either by an AC or another PE, it is checked if there is a similar query running on the node. If such a query exists, either the new-coming query, or the already running one, becomes linked to the other and share results. So, if query multiplexing is enabled, the total QueryInPE objects are divided into two subsets, the linked QueryInPE objects and the running QueryInPE objects. As we showed in section 7.5, the running queries and some of the linked queries form another subset of QueryInPE objects, the result forwarding QueryInPE objects. This category of query objects is very important as it gives us an idea of the query processing status of the whole system. If query multiplexing feature is not enabled, the number of result forwarding QueryInPE objects is equal to the number of supported QueryInPE objects (total – unsupported query objects).

In the next figure we present the results of an experiment we run in order to measure the performance gain we get from the use of query multiplexing for varying join probabilities. In the first graph we present *the percentage of result forwarding query objects when query multiplexing is used to result forwarding query objects when it is not.* In the next graph we present *the percentage of SNFE query objects when query multiplexing is used to SNFE query objects when it is not.*

In the presented experiment the join probability parameter varies from 0.1 to 1 (with a step of 0.1. The *branch degree* parameter is set to 7 and so does the maxCousins parameter. 10000 PEs are created and each one of them has a random number of sensing capabilities between 0 and 2 (*maxSC*=2) choosing from a set of 8 available capabilities. There are 200000 queries created choosing a target sense from the available sensing capabilities set. The simulation is first run with query multiplexing deactivated and then with the feature activated. Sensing capabilities' awareness is enabled in both cases.

```
number of PEs:                        10000
branch degree:                        7
probability of join(pJoin):           varying (0.1:0.1:1.0)
available sensing capabilities:       8
maximum sensing capabilities per PE:  2
number of queries:                    200000
join method:                          cousin check activated
processing mode:                      2(query multiplexing deactivated) /
                                      3(query multiplexing activated)
maxCousins:                           7
```

Figure 7.4.a:The simulation parameters of the following graphs

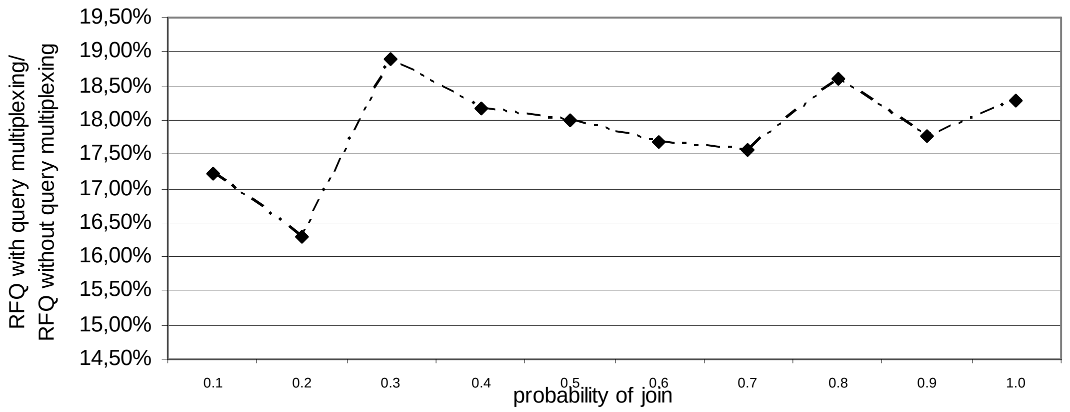


Figure 7.4.b: Percent of Result Forw arding QueryInPE objects w hen query multiplexing is used to Result Forw arding QueryInPE objects w hen query multiplexing is not used

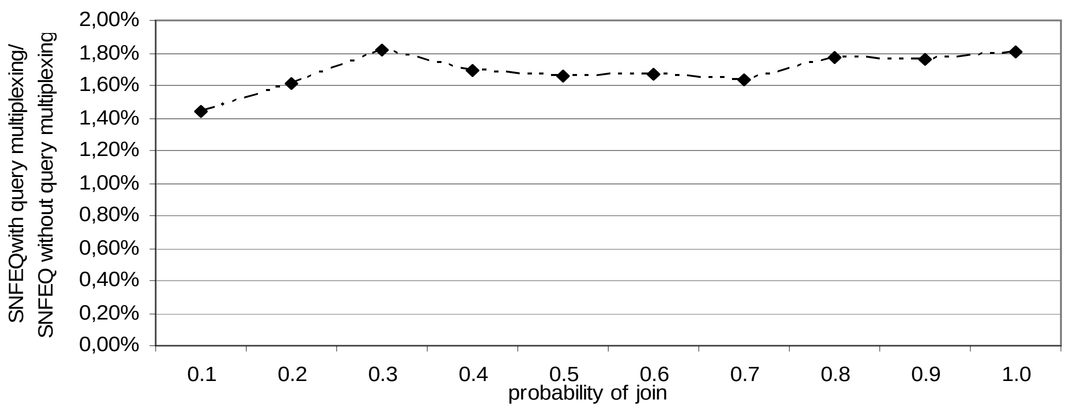


Figure 7.4.b: Percent of SNFE query objects w hen query multiplexing is used to SNFE query objects w hen query multiplexing is not used

Figure 7.4 : Experiment 3
(Query Multiplexing feature evaluation for varying join probabilities)

As we see in the first graph, we have an important decrement of the result forwarding queries when query multiplexing is used, which leads to more efficient query processing. The observed decrement doesn't seem to link with the changes in probability of join, a fact that shows the efficiency of the constructed tree hierarchy.

In the second graph we notice that the SNFE query objects, are decreased when query multiplexing is used.

It is obvious that query multiplexing is very efficient as it not only reduces network traffic amongst PE-nodes but it also leads to fewer queries to the available sensor devices that consume less energy and produce less communication messages.

Finally we present an experiment that shows the performance gain from the use of query multiplexing in PE sets with varying maximum sensing capabilities per PE. In order to study the gain we get from the query multiplexing feature, we measure the percent *of result forwarding QueryInPE objects percentage to supported QueryInPE objects* (RFQ/ (TQ – UQ)). When query multiplexing isn't used the percentage is 100% as result forwarding queries are equal to supported queries.

```
number of PEs:                      10000
branch degree:                      7
probability of join(pJoin):         0.7
available sensing capabilities:     8
maximum sensing capabilities per PE: varying (1-8)
number of queries:                  200000
join method:                        cousin check activated
processing mode:                    3(query multiplexing activated)
maxCousins:                         7
```

Figure 7.5.a:The simulation parameters of the following graph

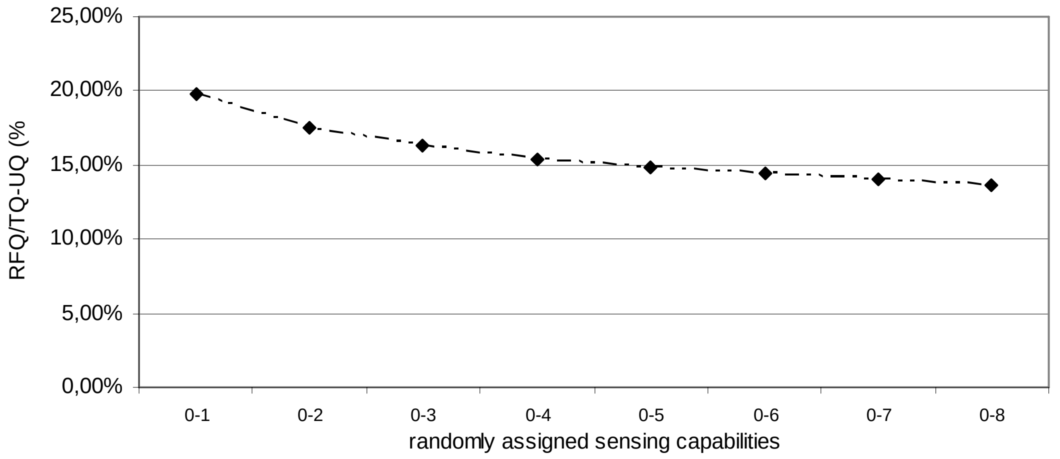


Figure 7.5.b: The result forwarding queries/supported queries ratio with varying maximum sensing capabilities per PE

Figure 7.5 : Experiment 4
(Query Multiplexing feature evaluation for varying maximum sensing capabilities)

The join probability parameter is set to 0.7. The *branch degree* parameter is set to 7 and so does the maxCousins parameter. 10000 PEs are created. There are 200000 queries created choosing a target sense from the available sensing capabilities set. The varying parameter is the maximum number of (random) sensing capabilities per PE. It varies from 1 to 8. The simulation is run with the query multiplexing and sensing capabilities' awareness features enabled.

As we see in the graph we have better performance as the PEs acquire more sensing capabilities.

We must denote that all the simulations presented in this chapter present a view of the system in a particular moment with a particular set of queries running on a particular set of PEs.

Chapter 8

Conclusions and Future work

8.1. Conclusions

In the last few years there has been a lot of interest in the field of sensor networks. It is accepted that in the next years, networks of sensors are going to be present in various areas of our planet. In chapter 2 we presented some projects which aim in the collection of data from different sensor networks.

The objective of this thesis was to provide a distributed, self-organized framework for querying multiple sensor networks through the Internet with area criteria that will face the challenges of sensor data collecting overlay networks. The most important feature of our system is the automatic organization of the participating peers, that represent the actual sensor networks, and the utilization of the constructed, area based, hierarchy to send, in an efficient and transparent way, the users' queries to the edges of the architecture, where the actual sensing devices receive them through a simple interface. As we showed in the evaluation chapter, the hierarchy self-organizing algorithms along with the query multiplexing and sense awareness scheme we propose, enable efficient result data reuse in peers' and sensors' levels, which leads in less power consumption, smaller memory footprints and fewer communication messages.   We paid a lot of attention in the transparent and dynamic entrance and departure of peers in the system. Any peer failures are properly handled and do not affect the status of the properly running nodes' hierarchy.

The major innovation of the system we propose is its self-organization scheme. In other similar projects mentioned in Chapter 2, which aim in the collection of data from different sensor networks, either no hierarchy structure is used, or the hierarchy is static and defined a priori.

8.2. Future work

The system we propose is quiet efficient as it not only provides a transparent interface between client applications and sparse sensor networks, but it also succeeds in reducing the consumed resources both in the peers' hierarchy and in the actual sensor networks. However, its functionality can be further enhanced and expanded.

The self-organization algorithm is currently performed only during the join of each participating peer to the system. More efficient hierarchy organization will be achieved if an extra, distributed re-organization algorithm is executed in all the peers during runtime. Excessive child-peers or large tree depths would be detected and confronted with peers' shift in upper or lower levels. Of course, something like that would presuppose an updated failure handling scheme in order to achieve distributed consensus throughout the peers' set.

As we described, query and result data are represented as Java Objects. In order to be more easily handled by client applications (or web pages) they could be represented in XML format possibly in a widely adopted data representation language like SensorML [40], EEML [31] or TML [41]. The exchanged messages between peers would become bigger, but we could achieve more manipulation options and maybe reduce the processing time in client applications.

In order to make the system more reliable, a distributed version of the Peer Elements Registry could be implemented. This would also reduce the PER request processing time. The distributed version of the component could be implemented either with a custom-made protocol or with an off-the-self distributed registry solution.

The classes we developed implementing the SensorNetworkGateway Interface are rather simple. A more advanced smart-its gateway could be added to the system, with support for multiple sense types and possibly advanced query and result routing between the particles. TinyDB could also be used in order to create a gateway class communicating with TinyOS-enabled sensor motes. Of course, custom gateway classes can be developed for any sensing device that can be accessed through a host PC.

Finally, advanced client interfaces can be created that will use the provided Application Client component API. They may be either classic Java applications or JSP-enabled web-pages used for system monitoring and query submission.

References

[1] http://web.syr.edu/~gkamathh/topics/micamote.html *The Mica Motes web page.*

[2] http://particle.teco.edu/ , *the smart-its devices web page.*

[3] http://www.intel.com/research/exploratory/motes.htm, t*he Intel Motes web page.*

[4] http://www.btnode.ethz.ch/Projects/SensorNetworkMuseum , *List of wireless sensor motes*

[5] http://www.tinyos.net/, t*he TinyOS project web page.*

[6] http://nescc.sourceforge.net/ , t*he nesCprogramming language web page.*

[7] V. Naik et. al.: "Kansei: Sensor testbed for at-scale experiments". *In 2nd Intl TinyOS Technology Exchange, February 2005.*

[8] L. Girod, Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, Deborah Estrin: "Em∗: A software environment for developing and deploying wireless sensor networks". *In Proceedings of the 2004 USENIX Technical Conference, April 2004.*

[9] G. Werner-Allen, P. Swieskowski, and M. Welsh: "Motelab: A wireless sensor network testbed". *In Proceedings of the 4th Intl. Conf. on Information Processing in Sensor Networks (ISPN '05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS), pages 73–78, April 2005.*

[10] http://telegraph.cs.berkeley.edu/tinydb/documentation.htm, *the Tiny DB documentation web page.*

[11] S. Madden, J. Hellerstein, and W. Hong : "TinyDB:In-Network Query Processing in TinyOS".

[12] S. Lei, W. Xiaoling, X. Hui, Y. Jie, J. Cho, and S. Lee: "Connecting Heterogeneous Sensor Networks with IP Based Wire/Wireless Networks" *in Proceedings of the The Fourth IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems, and the Second International Workshop on Collaborative Computing, Integration, and Assurance (SEUS-WCCIA'06) - Volume 00 Pages: 127 - 132.*

[13]A. Dunkels, J. Alonso, T. Voigt, H. Ritter, J. Schiller : "Connecting Wireless Sensornets with TCP/IP Networks". *In Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004), Frankfurt (Oder), Germany, February 2004.*

[14] H. Dai, R. Han : "Unifying Micro Sensor Networks with the Internet via Overlay Networking". *In Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks, pages: 571 - 572*

[15] Z. Marco, K. Bhaskar: "Integrating Future Large-scale Wireless Sensor Networks with the Internet", *USC Computer Science Technical Report CS 03-792, 2003.*

[16] M. Isomura, T. Riedel, C. Decker, M. Beigl, H. Horiuchi: "Sharing sensor networks". *In 26th IEEE International Conference on Distributed Computing Systems Workshops, 2006. ICDCS Workshops 2006. pages: 61- 61*

[17] http://www.sun.com/software/jxta/ , t*he JXTA technology web page.*

[18] http://www.upnp.org/ , the UPnP forum web page

[19] M.J. Franklin, S.R. Jeffery, S. Krishnamurthy, F. Reiss, S. Rizvi, E. Wu, O. Cooper, A. Edakkunni, and W. Hong: "Design Considerations for High Fan-in Systems: The HiFi Approach". *In CIDR 2005: 290--304.*

[20] S. Chandrasekaran et al. :"TelegraphCQ: Continuous Dataflow Processing for an Uncertain World". *In CIDR (2003).*

[21] D. Abadi et al.: "Aurora: A New Model and Architecture for Data Stream Management". *In VLDB Journal, (August 2003)*

[22] R. Motwani et al.: “Query Processing, Resource Management and Approximation in a Data Stream Management System”. *In CIDR (2003).*

[23] http://www.intel-iris.net/research.html , *the Iris Net documentation web page*

[24] http://www.eecs.harvard.edu/~mdw/proj/codeblue/ , *the CodeBlue project web page.*

[25] A. Santanche, S. Nath, J. Liu, B. Priyantha and F. Zhao: "SenseWeb: Browsing the Physical World in Real Time". *Demo Abstract, ACM/IEEE IPSN06, Nashville, TN, April 2006.*

[26] S. Nath, J. Liu, and F. Zhao: “Challenges in Building a Portal for Sensors World-Wide”. *First Workshop on World-Sensor-Web: Mobile Device Centric Sensory Networks and Applications (WSW'2006), Boulder CO, Oct 31, 2006.*

[27] http://atom.research.microsoft.com/sensormap , *Microsoft’s Sensormap web-interface.*

[28] K. Aberer, M. Hauswirth, A. Salehi: “The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks”. *Technical report LSIR-REPORT-2006-006*

[29] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, M. Welsh: “Hourglass: An Infrastructure for Connecting Sensor Networks and Applications”. *Harvard Technical Report TR-21-04*

[30] http://www.doc.ic.ac.uk/~peter/sbon.html, *the SBON project web page*

[31] http://www.eeml.org/ *the Extended Environments Markup Language (EEML) web page*

[32] http://www.discovery-on-the.net , *the Discovery – Net project web-page.*

[33]R. Bramley, et. al.: “Instruments and sensors as network services: Making instruments first class members of the grid”. *Technical Report 588, Indiana University CS Department, December 2003.*

[34] I. Foster, C. Kesselman, J. Nick, and S. Tuecke: “The physiology of the grid: An Open Grid Services Architecture for distributed systems integration”. *Open Grid Service Infrastructure WG, Global Grid Forum, June 2002.*

[35] H. B. Lim, Y. M. Teo, P. Mukherjee, V. T. Lam, W. F. Wong and S. See: “Sensor Grid: Integration of Wireless Sensor Networks and the Grid”. I*n the IEEE Conference on Local Computer Networks, 2005, pages: 91- 99.*

[36] I. Foster and C. Kesselman: “Globus: A meta-computing infrastructure toolkit”. *Intl Journal of Supercomputer Applications, 11(2):115–128, 1997.*

[37] C. Smith. : “Open source metascheduling for virtual organizations with the Community Scheduler Framework (CSF)”. *White paper, Platform Computing, Inc, 2004.*

[38] P. Bulhoes, et. Al.: “N1 Grid Engine 6 features and capabilities”. *White paper, Sun Microsystems, Inc.*

[39] http://particle.teco.edu/software/libparticle/index.html, *the libparticle library web page.*

[40] http://www.opengeospatial.org/standards/sensorml, *the Sensor Markup Language (SensorML) web page.*

[41] http://www.ogcnetwork.net/node/105, *the Transducer Markup Language (TML) web page.*

[42] M. Gaynor, et. al. “Integrating wireless sensor networks with the grid”. *In IEEE Internet Computing, pages 32–39, Jul/Aug 2004.*