SIXTH FRAMEWORK PROGRAMME
PRIORITY 2
"Information Society Technologies"

Project acronym: RUNES
Project full title: Reconfigurable Ubiquitous Networked Embedded Systems
Proposal/Contract no.: IST-004536-RUNES

D5.1
Survey of Middleware for Networked Embedded Systems

Project Document Number: RUNES/D5.1/PU[1]/v1.1 (official version)

Project Document Date: 07/01/2005

Workpackage Contributing to the Project Document: WP5

Deliverable Type and Security: R[2]-PU

Author(s): Cecilia Mascolo, Stephen Hailes, Leonidas Lymberopoulos (University College London)
Gian Pietro Picco, Paolo Costa (Politecnico di Milano)
Gordon Blair, Paul Okanda, Thirunavukkarasu Sivaharan (University of Lancaster)
Wolfgang Fritsche, Mayer Karl (Indusrieanlagen-Betriebsgesellschaft mbH)
Miklós Aurél Rónai, Kristóf Fodor (Ericsson Research, Traffic Lab)
Athanassios Boulis (National ICT Australia)

[1] Security Class: PU- Public, PP – Restricted to other programme participants (including the Commission), RE – Restricted to a group defined by the consortium (including the Commission), CO – Confidential, only for members of the consortium (including the Commission)

[2] Type: P - Prototype, R - Report, D - Demonstrator, O - Other

Abstract:

Ubiquitous computing is an emerging research area, posing significant challenges in both theoretical and technological aspects of middleware systems. Up to today, middleware platforms have successfully been used for delivering solutions for deployment and management of distributed applications. However, whilst middleware systems have been deployed within computing environments interconnected with Local Area or Wide Area Networks, the new ideas the ubiquitous computing “world” has introduced: a) devices with small size, even invisible, either mobile or embedded in almost any type of object someone can imagine (cars, appliances, clothes, etc.), and; b) devices communicating with increasingly interconnected mobile ad-hoc networks, pose new challenges and requirements for the design and deployment of middleware systems targeting at a network embedded system. New solutions yet remain to be devised in order to deploy middleware for heterogeneous, large-scale and continuously changing embedded environments, consisting of devices with different capabilities and limitations.

The document defines what networked embedded systems are and proposes two taxonomy models for organising and reviewing existing networked embedded systems. A number of existing middlewares for networked embedded systems is presented and evaluated, using the dimensions of the two proposed taxonomy models. The document concludes with a summary of the main achievements of the presented middlewares and with a discussion of goals that remain unattended.

Keywords: middleware for networked embedded systems; functional and non-functional requirements; middleware taxonomy; middleware survey

History

Version	Date	Description, Author(s), Reviser(s)
0.8	06/01/2005	Final version of the deliverable 5.1 containing all inputs from the WP5 partners (UCL, PDM, ULANC, ETH, IABG)
1.0	06/01/2005	Reformatted, executive summary added - ETH

Executive Summary

A number of challenging problems are introduced when trying to design and implement distributed applications operating within a networked embedded system. No assumptions can be made about the network topology and the available network resources as they can change rapidly; temporary and unannounced loss of network connectivity happens frequently when mobile nodes move; hosts must be discovered in an ad-hoc manner; nodes are likely to have scarce resources, such as low battery power, slow CPU and little memory; the number of nodes operating in a networked embedded environment can be of orders of magnitude greater than the number of hosts operating within a distributed system with permanent connections, such as a local area or a wide area computer network.

Networked embedded systems may come in many different forms. The most common scenario which is, in fact, envisaged in the RUNES project is that of very hybrid networks composed of different joined subsystems. Our research has shown that there is no middleware readily available for these very heterogeneous scenarios and that most of the middleware somehow assume a more homogeneous topology or set of device characteristics.

Middleware platforms form the “glue” between a distributed application’s execution space and the underlying network operating system, which is used to send messages to remote components, collaborating with the distributed application. Such remote components can be other distributed applications, remote servers, computer hardware or sensors collecting data from the environment where the sensors are embedded. Enabling communication between remote components, located not in a single machine, but rather deployed within large-scale and heterogeneous networked environments, is a difficult task due to diverse list of requirements that need to be taken into consideration by the middleware developers. Scalability, heterogeneity, fault-tolerance, openness and security are some examples of **non functional** requirements that need to be addressed when implementing a middleware platform that can suit the needs of different distributed applications.

On the other hand, a middleware platform resides between a distributed application and the network operating system. This implies that a number of different **functional** requirements must be addressed in order to enable communication between the application that knows nothing about the network infrastructure and the network itself. Requirements such as event notification, logging, addressing, discovery, context-awareness are inherently related to the functionality of the system.

We present a survey of middleware systems for networked embedded systems, as we have found in the literature at the time of writing this document. Using our classification of networked embedded systems, we analyze the existing solutions. In this survey we consider three types of systems and review the middleware developed for these different systems. **Mobile Systems** have been studied for some time now and can be further subdivided into Nomadic systems and Ad Hoc systems. Nomadic systems are those containing a core fixed network and some mobile nodes which are usually just leaves in the network: cellular networks are typical examples of systems with this topology. Ad Hoc systems have a very decentralized structure where no fixed core exists and where each node may be mobile. In both cases the type of network links is wireless. Devices change their location continuously, with different frequency depending on the applications. **Embedded Systems** are systems assuming that the computing components are embedded into some other purpose built device (an aircraft or a car). These systems are most of the time not mobile and very often not networked in large scale, i.e. only few devices are connected, usually with only one other server machine and most of the time not to external networks. The type of the connection is often wired. **Sensor Systems** are often composed by large numbers of possibly tiny devices which have the sole task of monitoring some conditions within an environment and report back to a central server. The most common sensor networks are usually not mobile but the sensors are connected through a radio link. Wireless sensor networks are a specific but a widely deployed example of networked embedded systems. Both the industry and academia has recently shown a lot of interest in wireless sensor networks technologies that enable deployment of a wide range of applications, such as military, environmental monitoring, e-health applications, etc.

As the survey shows, in existing networked embedded systems most of the effort has been put on addressing the requirement of adaptability. However, the adaptation mechanisms of existing middlewares remain static during their lifetime. This means that it is not possible to use an alternative adaptation mechanism when different adaptive behavior is required to be performed by the middleware. For example, it is possible that during the lifetime of a networked embedded system, a new application is introduced demanding a different type of adaptation than the one offered by the middleware. For this purpose, a new adaptation mechanism should be selected and configured within the middleware in order to provide the new type of adaptation required by the application. The problem of selecting and configuring the new adaptation mechanism becomes more complex considering the demand for autonomous operation of the middleware. Since it is not possible to assume human intervention for management of the middleware operating within a networked embedded system, intelligent mechanisms such as expert systems should be studied to see if they can operate within the middleware, such as the middleware itself has the capabilities to dynamically update its adaptation mechanisms.

Another problem that we can see is that although most of the middleware has addressed the requirement for adaptability, no single middleware exists addressing all the non-functional requirements of our list. This is due to the assumptions about the application or the types of embedded devices that the middleware designers have made. For example, some of the work on sensor networks assumes the existence of a single type of sensors embedded in the environment, thus not addressing the requirement for heterogeneity. However, in the most general case of a networked embedded system consisting of many types of devices other than single-type sensors, it is important that the requirement for heterogeneity is addressed by the middleware.

We can also observe, that the requirement of feasibility has not been adequately addressed by existing middleware systems. However, considering the resources limitations (in hardware and in network resources) that are potentially present in a networked embedded system, new mechanisms should be implemented to ensure that the functions or services that the middleware offers to the application are feasible to be performed in a given system/network instance.

Taking into account what we have learned from existing middleware solutions, our goal in RUNES is to create a middleware solution for networked embedded system, which fulfils the identified requirements and solves the problems we have found in existing approaches.

Contents

Page

Figures, Tables

1 Introduction

Advances in electronic and wireless technologies, have resulted in the deployment of a wide range of products with computing and network communication capabilities, other than the personal computers as we use up to today. Devices with small size, such as mobile phones, personal digital assistants, digital cameras, sensors for measuring environmental phenomena or medical conditions, etc. are continuously introduced in everyday life, offering new services to the user of such systems. Whilst the size of the devices mentioned above is becoming smaller and smaller, their computing capabilities are growing and so do the capabilities of the networks supporting their communication. This enables the programmers of networked embedded systems, ie. systems comprised of interconnected embedded devices, to be able to build new and more powerful distributed applications

A number of challenging problems are introduced when trying to design and implement distributed applications operating within a networked embedded system. No assumptions can be made about the network topology and the available network resources as they can change rapidly; temporary and unannounced loss of network connectivity happens frequently when mobile nodes move; hosts must be discovered in an ad-hoc manner; nodes are likely to have scarce resources, such as low battery power, slow CPU and little memory; the number of nodes operating in a networked embedded environment can be of orders of magnitude greater than the number of hosts operating within a distributed system with permanent connections, such as a local area or a wide area computer network.

When developing distributed applications, designers do not have to explicitly deal with problems related to distribution, such as heterogeneity, scalability, resource sharing and fault tolerance. *Middleware* developed upon network operating systems provides application designers with a higher level of abstraction, hiding the complexity introduced by distribution. Existing middleware technologies, such as transaction-oriented, message-oriented or object-oriented middleware have been built trying to hide distribution as much as possible, so that the system appears as a single integrated computing facility. In other words, distribution becomes *transparent*.

These technologies have been designed and are successfully used for stationary distributed systems. However, as it will become clearer in the following, some of the requirements introduced by the dynamic nature of a networked embedded system cannot be fulfilled by these existing traditional middleware. For example, interaction primitives, such as distributed transactions, object requests or remote procedure calls, assume a stable, high bandwidth and constant connection between components. Furthermore, synchronous point-to-point communication supported by object-oriented middleware systems, such as CORBA [CORBA], requires a rendez-vous between the client asking for a service, and the server delivering that service. In networked embedded systems, on the contrary, unreachability is not exceptional and the connection may be unstable. Moreover, it is quite likely that client and server hosts are not connected at the same time, because of voluntary disconnections (e.g., to save battery power) or forced disconnection (e.g., loss of network coverage). Disconnection is treated as an occasional fault by many traditional middleware; techniques for data-sharing and replication that have been successfully adopted in traditional systems might not, therefore, be suitable, and new methodologies need to be explored.

Finally, traditional middleware systems have been designed targeting devices with almost no resource limitations, especially in terms of battery power. On the contrary, even considering the improvements in the development of these technologies, resources of embedded devices will always be, by orders of magnitude, more constrained.

The aim of this survey document is to give an overview of how the requirements usually associated to distributed systems are affected by the high degree of dynamism and the resources constraints experienced within a networked embedded system. We provide a framework and a classification of the most relevant literature in this area, highlighting goals that have been attained and goals that need to be pursued.

This survey document is structured as follows. Section 2 describes what network embedded systems are. Section 3 proposes two taxonomy models for middleware operating within networked embedded systems. Section 4 contains a detailed review of existing middleware for networked embedded systems, and finally; section 5 contains a critical analysis of the reviewed middleware systems, using the taxonomy dimensions introduced in this survey.

2 Networked Embedded Systems

Networked embedded systems may come in many different forms. The most common scenario which is, in fact, envisaged in the RUNES project is that of very hybrid networks composed of different joined subsystems. Our research has shown that there is no middleware readily available for these very heterogeneous scenarios and that most of the middleware somehow assume a more homogeneous topology or set of device characteristics.

In this survey we will consider three types of systems and will review the middleware developed for these different systems:

- Mobile Systems
- Embedded Systems
- Sensor Systems

Mobile Systems have been studied for some time now and can be further subdivided into Nomadic systems and Ad Hoc systems. Nomadic systems are those containing a core fixed network and some mobile nodes which are usually just leaves in the network: cellular networks are typical examples of systems with this topology. Ad Hoc systems have a very decentralized structure where no fixed core exists and where each node may be mobile. In both cases the type of network links is wireless. Devices change their location continuously, with different frequency depending on the applications.

Embedded Systems are systems assuming that the computing components are embedded into some other purpose built device (an aircraft or a car). These systems are most of the time not mobile and very often not networked in large scale, i.e. only few devices are connected, usually with only one other server machine and most of the time not to external networks. The type of the connection is often wired.

Sensor Systems are often composed by large numbers of possibly tiny devices which have the sole task of monitoring some conditions within an environment and report back to a central server. The most common sensor networks are usually not mobile but the sensors are connected through a radio link. Wireless sensor networks are a specific but a widely deployed example of networked embedded systems. Both the industry and academia has recently shown a lot of interest in wireless sensor networks technologies that enable deployment of a wide range of applications, such as military, environmental monitoring, e-health applications, etc.

In this survey we will report on the different middleware that has been implemented for these three types of systems.

3 Taxonomy of middleware systems

Middleware platforms form the “glue” between a distributed application’s execution space and the underlying network operating system, which is used to send messages to remote components, collaborating with the distributed application. Such remote components can be other distributed applications, remote servers, computer hardware or sensors collecting data from the environment where the sensors are embedded. Enabling communication between remote components, located not in a single machine, but rather deployed within large-scale and heterogeneous networked environments, is a difficult task due to diverse list of requirements that need to be taken into consideration by the middleware developers. Scalability, heterogeneity, fault-tolerance, openness and security are some examples of **non functional** requirements that need to be addressed when implementing a middleware platform that can suit the needs of different distributed applications.

On the other hand, as we have already mentioned, a middleware platform resides between a distributed application and the network operating system. This implies that a number of different **functional** requirements must be addressed in order to enable communication between the application that knows nothing about the network infrastructure and the network itself. Requirements such as event notification, logging, addressing, discovery, context-awareness are inherently related to the functionality of the system.

In the following, we will use these categories to distinguish the different existing middleware and identify the main differences among them.

3.1 Taxonomy with respect to non-functional requirements

As we discussed earlier in this section, a number of requirements should be taken into consideration when implementing a middleware system. Middleware is itself a distributed system and as such, any middleware needs to meet the requirements for any type of distributed system, ie. a system compromised of software and hardware components running/located within different computing devices and in different locations, providing services to applications by interacting with each other. In all distributed systems, the interaction between remote software/hardware components is realised by means of message exchanges over an underlying network infrastructure. A list of requirements for a distributed system is provided in [Coulouris]. This list includes the requirements of fault-tolerance, openness, heterogeneity, scalability, resource sharing, transparency, concurrency and security. Our survey will use the list of requirements discussed in [Coulouris], but it will also point out additional requirements, which must be also addressed by the middleware programmers. Table I presents the requirements for a middleware system. We will use these requirements in order to classify and evaluate middleware systems, according to whether and in what degree a specific middleware meets each individual requirement.

Requirement	Description
Heterogeneity	Components written in different programming languages, running on different operating systems, executing on different hardware platforms, should be able to communicate using a middleware platform. Generally, in a distributed system, heterogeneity is almost unavoidable, as different components may require different implementation technologies.
Openness	The capability to extend and modify the system, for example, with respect to changed functional requirements. Adding new services or re-implementing existing ones should be possible within an open distributed system.
Scalability	The ability of the system to accommodate a higher load at some time in the future. A system's load can be measured using many different parameters, such as the maximum number of concurrent users, the number of transactions executed in a time unit, etc.
Failure handling	The ability to recover from faults without halting the whole system. Faults happen when software or hardware components fail to complete their delegated actions/methods, but still any distributed component must continue to operate even if other components they rely on have failed.
Security	Security mechanisms, such as authentication, authorization, and accounting (AAA) functions may an important part of the middleware in order to intelligently control access to computer and network resources, enforcing policies, auditing network/user usage, etc.
Performance	Performance can constitute a requirement of the middleware in various situations. For example, a middleware used for a real-time distributed application should export functions with a minimal, as possible, execution time, whereas a memory-limited device would require optimization of the memory usage of the middleware.
Adaptability	Changes in applications' and users' requirements or changes within the network, may require the presence of adaptation mechanisms within the middleware. For example, a different transport protocol should be chosen by the middleware when a mobile device enters a network supporting a different transport protocol than the one offered by the current network hosting the device.
Feasibility	Constraints of available resources may limit the feasibility of performing certain tasks or offering certain services in a given system/network environment. Mechanisms should be provided to ensure that a function, task or a service is feasible to be provided in a given system/network instance.

Table 3.1: Non-functional requirements of a middleware system

Note that the list of requirements included here is not a closed set, i.e. it is possible that the designer of a middleware wishes to take into consideration additional requirements, such as the requirements for concurrency and transparency, or pricing criteria. Concurrency is needed when services and applications provide resources that can be shared by clients in a distributed system. For an object to be safe in a concurrent environment, its operations should be synchronized in such a way that data remains consistent. Transparency is defined as the concealment from the user and the application programmer if the separation of components in a distributed system, so that the system is perceived as a whole rather as a collection of independent components. Finally, someone could aim at developing middleware that minimizes the use of resources associated to pricing, e.g., the number and the duration of mobile connections needed for certain operations of the middleware system.

In the following, since the scope of this survey is specific to middleware for networked embedded systems, we will explain in more detail the list of requirements shown in Table 3.1 in the context of a networked embedded environment, taking into account the particular factors that potentially can influence the non-functional requirements of the middleware system operating within the networked embedded environment.

Heterogeneity

Industrial machines, automobiles, medical equipment, cameras, household appliances, airplanes, vending machines, toys, and recently sensors and actuators (as well as the more obvious cellular phones and PDAs) are among the myriad possible hosts of an embedded system. As possible, middleware system should include the necessary abstractions in order to cater for the heterogeneous nature of a network embedded environment consisting of different types of devices, but cooperating with the middleware. Moreover, the middleware system should include the flexibility to use the available communication protocols that are eventually supported by particular devices.

Openness

Implementation of new functionality, or changes of an existing functionality should be possible to be permitted within the middleware as the set of applications changes or the set of embedded nodes is updated with new nodes, offering new functionality to the application. Therefore, as in the case of any distributed system, the middleware should have the capability to be extended and modified during its lifetime. Moreover, since data should be continuously be provided to the application, especially in the case of real-time applications, the process of updating or extending the middleware should not require halting its operation while this is being done.

Scalability

Compared to existing typical distributed systems, whose components communicate either with fixed or with mobile connections, the number of nodes in a network embedded system can be several orders of magnitude higher than the nodes in a traditional distributed system, for example a mobile ad hoc network. For example, in a sensor network, it is quite possible that sensors are densely deployed in a particular environment. According to the survey in [Akyildiz02], there exist scenarios, where sensors need to be deployed at a density of approximately 20 nodes/m3, where the size of a single sensor node is less than 1 cm3. As a consequence, it is important that the middleware scales with respect to the number of sensor nodes. One solution to this problem has been given that follows a clustering approach to the design of wireless sensor networks. For example, the LEACH [LEACH] middleware developed at MIT and the middleware presented in [Y.Yu03], partition the sensor network into multiple clusters. More information on existing middleware systems will be given in chapter 4 of this document.

Failure handling

In a network embedded system, some nodes may fail or be blocked to communicate due to lack of power, or be physically damaged or experience environmental interference. Wireless links used for communication in the network can fail or experience problems, such as transmission errors due to the unreliable transmission medium and to the presence of undetected collisions, increased and unpredictable delay, high packet loss, etc. Therefore, the failures occurring within a network embedded system are more common and more frequent than in a mobile ad hoc network or a distributed system with permanent connections between the end-nodes. The failure of individual nodes should not affect the overall task of the network embedded system, thus leading to an increased need for providing mechanisms to ensure fault tolerance to the applications.

Another factor that leads to an increased demand for providing fault tolerance in an embedded network system is the fact that after its deployment, topology changes are likely to occur due to changes in sensor nodes' location, their reachability (due to jamming, noise, moving obstacles, etc.). Furthermore, additional nodes can be redeployed at any time to replace other nodes and some nodes can stop functioning due to lack of power. Even in a continuously changing network topology, the middleware system should be able to perform its tasks and provide reliable services to the application.

Security

Security is a critical, though not broadly addressed, concern in a networked embedded system (eg. a sensor network), since, apart from traditional attacks, the possibility of physically capturing nodes adds a novel issue to the security realm.

Performance

Factors such as energy and memory efficiency must be taken into account considering the energy and memory limitations of the nodes within the network embedded system. For example, as nodes in a sensor network are battery-operated, energy-efficient mechanisms should be implemented within the middleware in order to maximize the system's lifetime. In addition, power-aware protocols should be selectively chosen and configured in order either to minimize the number of nodes running out of energy or to maximize the time within which the application can receive the desired level of information about the phenomenon the application is interested in.

Another performance aspect of the middleware is whether it can satisfy an application's real-time requirements. The requirement for a middleware with real-time capabilities should be addressed in several cases, such as when the middleware is used by applications that wish to receive real-time data (eg. temperature or seismic vibrations) from a sensor network.

Adaptability

The need to implement adaptation mechanisms within the middleware operating in a network embedded system is mainly due to two different types of changes that may occur during its operation.

On one hand, changes are likely to occur within the network. For example, the network topology may change, due to the addition of new nodes or malfunctioning of existing ones. These changes must be taken into account by the middleware so that the needs of the application are still met. Eventually, the middleware must be able to allow the application to modify its behavior if necessary, letting information emerge from the network level.

On the other hand, applications using the middleware may change at run-time their requirements, thus demanding adaptation of the middleware to cater for the new requirements. For example, the precision of data that the application wishes to receive can be changed during the application's lifetime. This in turn implies that the filtering mechanisms on sensor-generated data should be updated to provide the application the new required error tolerance. Another scenario for adaptation is the case where the application demands to obtain a maximum lifetime. This would require a constant, automated process to be implemented by the middleware that continuously selects which sensors should provide data to the application, so that the application's demands are satisfied and the list of sensors providing data to the application does not gets empty due to lack of power of individual nodes or communication losses.

Feasibility

Hardware constraints, such as the available memory of an embedded node, its remaining power and its computational power are some examples of resources constraints that must be taken into account by the middleware system. Another constraint is the limited bandwidth and the increased packet loss and delay expected to be experienced in a mobile ad hoc network of embedded nodes. This means that it is not possible that all tasks can be performed at any instance by every node within the system. Rather, the middleware should be aware of the constraints on the available resources and distribute and coordinate the tasks accordingly, in a way to achieve the desired behavior.

Note that the non-functional requirements discussed in this section, refer to a generic middleware platform, ie. they do not depend on the particular applications using the middleware. On the other hand, functional requirements of the middleware, expressed as services, tasks or functions the system is required to perform depend in many circumstances on the particular application using the middleware and the networking environment where the middleware operates. However, it is important that this survey includes a listing of the generic functional components and services that may present in a middleware system for a networked embedded system.

3.2 Taxonomy with respect to functional requirements

In the previous section, we discussed the main non-functional requirements that a middleware platform should address in order to meet the needs of distributed applications operating within a networked embedded environment. In this section, we will provide an overview of the main functional components and services that potentially need to be implemented within the middleware. We will also provide in the appendix of this document an overview of some state-of-the art implementations of security and location services targeting networked embedded systems.

In section 2 of this document, we identified three categories of networked embedded systems: mobile, embedded and sensors systems. The main functional components of each these categories are the following.

Mobile Systems

Devices in mobile systems may be of a quite large size and have powerful computing capabilities. The kind of applications and the middleware functionality that will be running on these devices will therefore vary quite considerably. In this section we want to identify a couple of important ones, which can be found on many of the middleware we have reviewed.

Sensors Systems

It is worth underlining that no available middleware platform for wireless sensor networks implements all of the functional components listed in this section, however there is at least one platform implementing each component. The aim here is to provide a general view of what functions could be offered by middleware platforms and show how the corresponding functional components are interconnected.

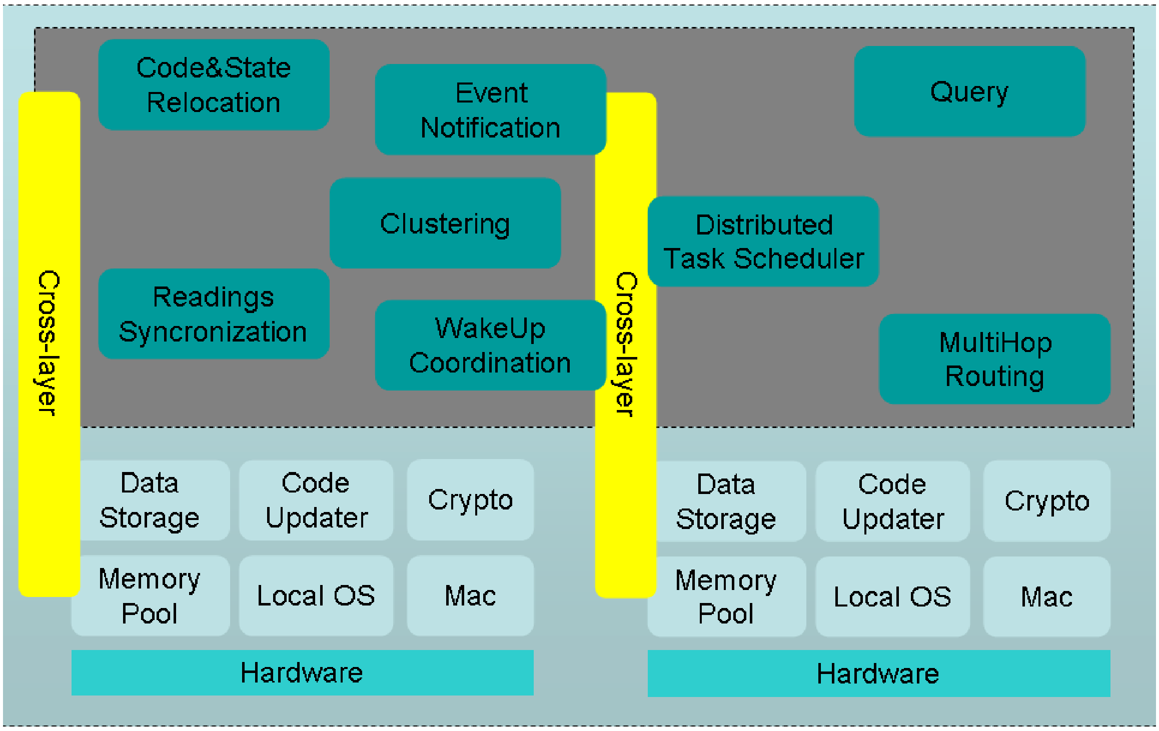


Figure 3.1:Functional components of middleware for wireless sensor networks

Figure 3.1 presents the main functional components that may be present within a middleware system operating within a wireless sensor network. A brief description of the functionality of each component is given below.

3.2.1 Functional Components for Mobile Systems

Event Notification: the ability to deliver messages to one or more recipients. Event notification systems are often also called publish-subscribe systems. In nomadic networks the event notification functionality is usually taken care of by the core fixed infrastructure, while the mobile devices simply receive the events. In ad hoc network, the solutions need to be more decentralized as the nodes have roughly the same kind of roles. Each node has an event notification component. Typically publish/subscribe systems contain information providers and information consumers. The information providers publish events to the information consumers and information consumers subscribe to particular categories of event. The Publish-subscribe (P/S) middleware ensures the delivery of published events to all interested subscribers.

Mobility and location awareness: the ability to roam or move from location to location is an essential component for these kind of systems. In terms of middleware, the mobility is the ability to recognize a new environment and adapt to it.

Addressing: this component recognizes the different devices around in order to be able to communicate with them.

Service discovery: the ability to identify new services and, possibly protocols. This is an essential component of the middleware, which also relate to the ability of relocating.

Code Updater: the ability to dynamically change the protocol and the code of the running applications or of the middleware itself in order to adapt to context.

3.2.2 Functional Components for Embedded Systems

Real time: often these systems are embedded into larger machines and devices that perform critical real time operations. It is the task of the middleware to maintain and monitor time and deadlines in order to maintain these kind of guarantees.

3.2.3 Functional Components for Sensor Systems

Hardware: This component represents the physical devices (CPU, sensors, radio transmitter, etc) within the host.

MAC: This component is responsible for one-hop broadcast and transmission. It may provide a simple collision avoidance strategy. We will not detail it further since it is out of the scope of this survey.

Local OS: It represents the local OS functionalities available on the host such as interrupt management, local task scheduling, basic I/O operations including sensor readings.

Code Updater: It provides mechanisms for replacing the code running on the sensor, ranging from a single application routine to the entire operating system. It supports both weak (only code is transferred) and strong (the state and the program counter are transferred too) mobility.

Cryptographic System: This block contains the standard functions for cryptography such as data encryption, decryption and hashing.

Local Memory Pool: It is a repository for sharing information among different components to save memory. A typical example is the neighbors' list that is exploited by MAC, multi-hop routing, wake-up coordination and others.

Sensor Readings Synchronization: Several applications (e.g., seismographic or building health monitoring) require a precise (in microseconds) synchronization among readings on different sensors. This component is responsible for accomplishing the level of synchronization needed by the application.

Wake-up Coordination: Clearly sensors must sleep most of the time to preserve battery and to increase network lifetime. Conversely, when a transmission takes place, sender and receiver need to be awake simultaneously. Likewise, it is fundamental to prevent collisions by not having too many nodes transmitting at the same time. Hence, an appropriate wake-up scheme must be adopted that takes into account all of these issues.

Cross-layer Communication: Its goal is two-fold: it exposes the hardware status (battery level, sensors characteristics, etc.) to the application or other components and it allows the application to specify the desired QoS in terms of sensing rate and accuracy, reliability and network management. It spans both the local and distributed level.

Multi-hop routing: It provides multi-hop point-to-point and multipoint style of communication. In particular, in the case of multipoint communications, the set of receivers may be identified by the specific subject of the message (multicast or subject-based routing), by its content (content-based routing) or by the receivers' physical location (geocast).

Clustering: Clustering is a fundamental functionality needed in sensor networks (e.g., for data aggregation and for energy-efficient communication). This component models the ability of aggregating nodes based on geographic proximity or other criteria (e.g., battery level, sensor types, etc.). It supports both intra- and inter-cluster communication, for example by providing algorithms for leader election.

Distributed Task Scheduler: It is responsible for allocating task on different hosts according to their specific capabilities and energy status.

High-level components: This set of components exploits the previously mentioned components to offer high-level functionalities to applications.

- Local data storage: this component controls the access to the local data, providing abstractions (eg. SQL-like) for data retrieval.
- (Distributed) event notification: it implements a distributed event notification service, allowing sensors to subscribe to interested events and to publish event notifications. If present, it may leverage off content-based routing as provide by multi-hop routing component, otherwise it may employ different strategies.
- (Distributed) query: it exposes a query-based interface to retrieve data on remote hosts.
- Code + state relocation: while code updater provides primitives for replacing running code on the host, this component takes care of all the issues concerning a distributed reprogramming of the network such as deployment, global consistency, etc.

4 Existing middleware for networked embedded systems

This chapter will present a survey of middleware systems for networked embedded systems, as we have found in the literature at the time of writing of this document. Using the classification of networked embedded systems that we presented in chapter 2 of this document, we will divide this chapter in three sections. The first section will overview important work on middleware targeting mobile systems, the second will focus on embedded systems, while the third on wireless sensor networks.

4.1 Middleware for Mobile Systems

This section overviews and evaluates some important solutions on mobile systems middleware.

4.1.1 GAIA

By extending the reach of traditional computing systems to encompass the devices and physical space surrounding the machines, entities, both physical and virtual, may be allowed to seamlessly interact. Physical spaces become interactive systems, or in other terms, Active Spaces. Such environments are analogous to traditional computing systems; just as a computer is viewed as one object, composed of input/output devices, resources and peripherals, so is an Active Space. However, the heterogeneity, mobility and sheer number of devices makes the system vastly more complex. Applications may have the choice of a number of input devices, such as location sensing system, mouse, pen, or finger and output devices, such as an everywhere display, monitor, PDA screen, wall-mounted display, speakers, or phone.

Gaia brings the functionality of an operating system to physical spaces. Common operating system functions are supported, such as events, signals, file system, security, processes, process groups, etc. Gaia extends typical operating system concepts to include context, location awareness, mobile computing devices and actuators, like door locks and light switches. Gaia is investigating how to build applications in a generic way that make no assumptions about the current hardware setup of a space - applications can be built and then deployed in spaces with different configurations, using the available resources.

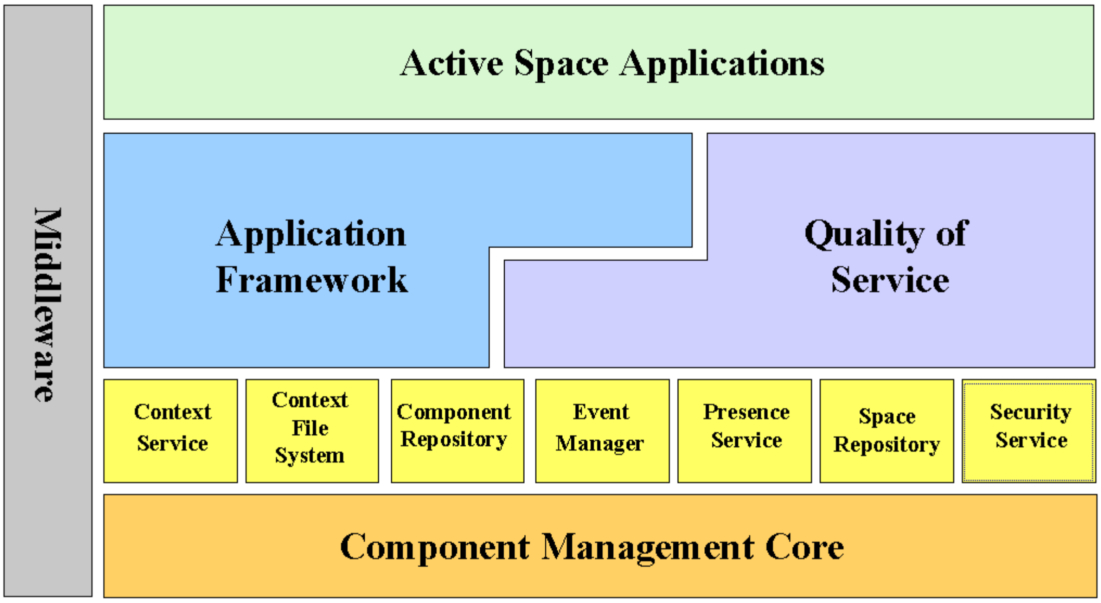


Figure 4.1: Gaia architecture

The key elements of the Gaia middleware are described in the following.

Component Management Core: The Component Management Core (CMC) provides Gaia with the functionality to manipulate components. This manipulation includes component creation, destruction, and uploading. The component manager does not impose any specific middleware protocol to provide support for remote method invocation; instead, it leverages existing middleware platforms. The CMC consists of three abstractions: (1) Gaia Components, (2) Gaia Nodes, and; (3) Gaia Component Containers. Gaia's components constitute the minimum software unit in the system. A Gaia Node is any device capable of hosting the execution of Gaia Components. However, Gaia Nodes organize components into containers (Gaia Containers), which group components and export an interface to manipulate the components that belong to such a group. Gaia nodes export an interface to manipulate component containers (i.e. create, browse, and delete).

Event Manager: Components in a loosely coupled system like an Active Space are typically designed to operate by generating and responding to asynchronous events. The event manager provides a model for decoupled communication between different entities in an Active Space. It allows creating channel categories, browsing these categories and their associated channels and creating and deleting channels associated with particular categories. All Gaia components use the event manager to learn about changes in the state of the space and react accordingly. The *Presence Service*, for example, listens to different channels to learn about new entities (e.g., software services and people entering the space), filters the information, and publishes events to inform the rest of the system about new entities discovered and entities that are no longer available.

Context Service: Making computers obtain, understand and use context while interacting with humans is a difficult task. The main difficulty is that there is no common, reusable model that can be used to handle context. Context service looks on a clausal model for context that is both simple and expressive. The model defines various properties of context as well as operations that can be performed on context. Based on this model of context, an infrastructure to enable context-awareness in ubiquitous computing environments has been developed. The infrastructure allows easy development and deployment of context sensors and context-aware applications.

Component Repository: Gaia Nodes (i.e., devices with a service exporting their functionality to Gaia) host the execution of software components (Gaia Components). However, it is not feasible to assume that every Gaia Node will have a copy of all possible software components that can be executed in the active space. Gaia implements a component repository service responsible for storing all software components known to the active space. This repository stores components as well as information related to the components (e.g., name, hardware platform, and required OS) and exports functionality to browse, store, and upload components to Gaia nodes.

Space Repository: The Space Repository is a centralized database containing information about all active devices and services in an Active Space. It keeps this information up-to-date by listening on the *Presence Channels*, where events about new entities, as well as entities that are no longer active, are sent. All entities in the system have an XML description, which includes properties such as entity type, name, location, etc. The Space Repository can be queried for entities based on these properties. For example, an application may require two large displays. These components can be found by querying the Space Repository for two displays that satisfy a particular dimension constraint. Note that in different spaces, different displays may result from such a query, depending on what resources are available.

Gaia is deployed in a prototype room containing state-of-the-art equipment, including programmable surround sound audio system, five plasma panels, HDTV, webcams, Tablet PCs, X10 devices, IR beacons, bluetooth, wireless ethernet, fingerprint devices, Iris scanners, smart phones, RF badges, and Ubisense location technology. The goal of Gaia is to design and implement a middleware operating system that manages the resources contained in an Active Space. An operating system for such a space must be able to locate the most appropriate device, detect when new devices are spontaneously added to the system, and adapt content when data formats are not compatible with output devices. Traditional operating systems manage the tasks common to all applications; the same management is necessary for physical spaces.

Evaluation

Gaia aims to provide middleware support for active space environments such as smart rooms and living environments. It essentially provides a distributed operating system where all input and output and processing units within a room are considered as a single computer.

Gaia supports the following functionality: *Event Notification*, *Mobility and Location Awareness*, *Addressing*, *Service Discovery* and *Code Updater*. However, the main drawback of the implementation of such functional components is that Gaia depends on centralized services and components. For example, *Event Notification* is based on centralized event brokers suitable only for Local Area Networks. *Addressing* and *Service Discovery* is realized through the use of Gaia's Space Repository that is a not distributed among nodes, but it is a centralized database containing information about all active devices and services in an Active Space. Support for *Mobility and Location awareness* in Gaia is limited to infrastructure based wireless environment. *Code Updater* functionality is offered by means of the Component Repository, which is used to store and upload new middleware components to Gaia nodes. However, it is not clear if and how Gaia includes mechanisms that can perform in an automated way (i.e. without the need of human intervention) the component update within the nodes.

Non functional requirements addressed by this middleware is mainly *openness. Openness* is provided in Gaia by means of the *Component Repository*, where middleware components are stored. This repository provides the functionality to dynamically, when necessary, upload new components to the corresponding Gaia nodes. Interoperability between heterogeneous platforms is not focused in Gaia.

4.1.2 ExORB

The developers of ExORB [Roman04] envisage a middleware infrastructure that enables carriers and developers of distributed applications for mobile phones to be able to configure the middleware, perform software upgrades on it and correct its behaviour at run-time. They use a new technique, called *externalization,* to explicitly externalize the middleware platform's state, logic and internal component structure. The end-result is that developers and carriers have the ability to control all middleware services by accessing, inspecting and modifying their state, logic and structure at run-time while maintaining an error-free user experience. As a result of the constraints in the mobile phone domain, e.g., limited resources and intermittent reliability, middleware services targeting them must: 1) be configurable statically and dynamically, 2) be dynamically updateable to correct errors, and; 3) provide support for run-time upgrades to add new functionality. Contemporary reflective middleware systems offer support for configurability and allow replacement of certain components to adapt to changes in their environments. However, these middleware services assume a basic skeleton in which changes are largely pre-defined. Ex-ORB attempts to provide a design in which it is possible to modify literally every aspect of the system (including the basic skeleton) and perform fine-grained customizations.

ExORB is realized using a construction technique called Dynamically Programmable and Reconfigurable Software (DPRS). DPRS is referred to as:

- Programmable because just like Field Programmable Gate Arrays (FPGAs), which facilitate programming of the behaviour of hardware, it allows programming of the behaviour of software by defining the structure and logic of the software.
- Reconfigurable as it makes it possible for one to access and change structure, logic and state of middleware services.

The changes in the middleware services occur at run-time hence the use of the term 'Dynamically'. DPRS uses a set of three abstractions to construct dynamically reconfigurable middleware services:

Micro-building Block (MBB): Is the smallest addressable functional unit in the system. It receives a set of input parameters, performs an action that may or may not have an effect on its state, and generates a set of output parameters. A good example is the 'registerObject' which receives two parameters as input i.e. a name and an object reference, updates a list (effectively its state) and outputs the number of registered objects in its list. This state attribute is stored as a name and value tuple alongside others in a system provided storage area. Hence the need to state transfer protocols is eliminated to replace MBBs. Instead, replacement of an MBB involves registration of a new instance of the MBB and provision of a pointer to the existing Figure 4.2 presents an MBB's structure.

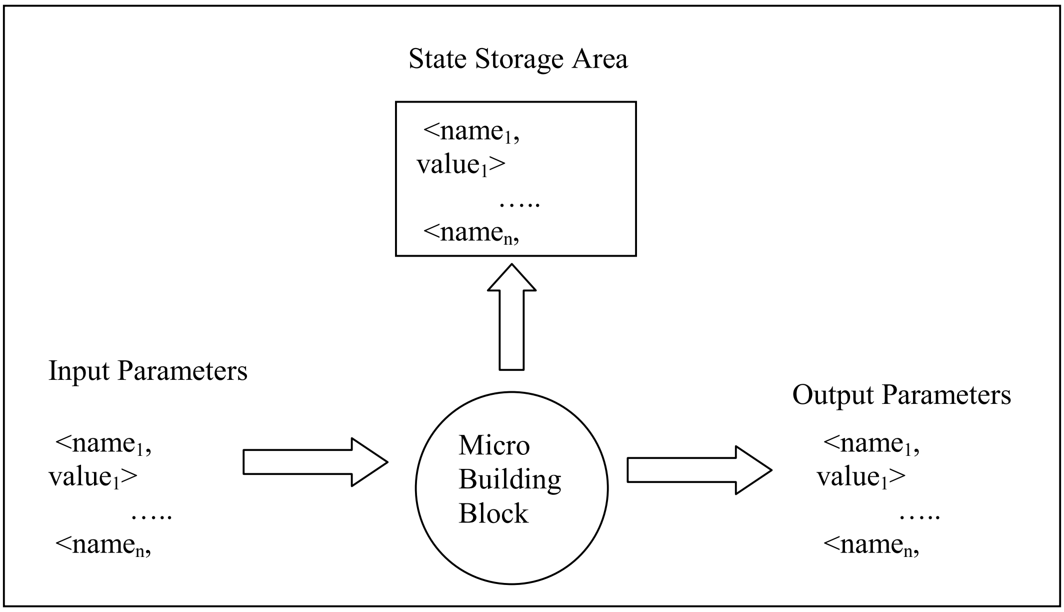

State Storage Area
<name1, value1> ….. <namen,
Input Parameters
<name1, value1> ….. <namen,
Micro Building Block
Output Parameters
<name1, value1> ….. <namen,

Figure 4.2: Micro Building Block Structure

It is important to note that MMBs do not contain references to other MMBs. This 'non-referencing' property makes the process of replacing a MBB.

Action: This specifies the order in which MBBs execute. They could also be seen as defining the logic of the system. There are two types: interpreted action and compiled action. An interpreted action is defined by DPRS as a deterministic directed graph where nodes are MMBs which denote execution states, and edges define the order of transition. Figure 4.3 presents an example.

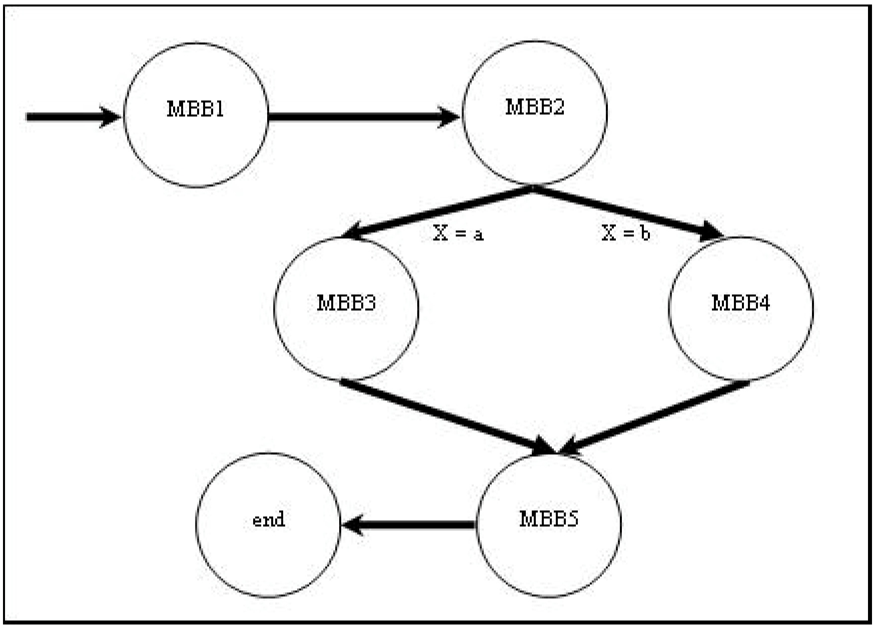


Figure 4.3: Interpreted action example

Execution of an interpreted action is equivalent to traversing the above graph. In this example, MMB1 is called the start node from which invocation starts. Execution then proceeds to MMB2 and depending on value of X, either MBB3 or MBB4 is executed. Finally, MMB5 is executed.

A compiled action is a fragment of code which specifies the order in which MMBs should be executed. A DPRS library is used by the compiled actions to invoke MBBs. The library's role is to receive a MBB name and a set of input tuples and invoke the MBB with the received input values. The figure below illustrates an example of a compiled action.

```
Action Test
{       outputParams = InvokeMBB(MBB1, inputParams);
        outputParams = InvokeMBB(MBB2, inputParams);
        char X = outputParams.get("X");
        if (X ==’a’)
                outputParams = InvokeMBB(MBB3, outputParams);
        if (X ==’b’)
                outputParams = InvokeMBB(MBB4, outputParams);
        outputParams = InvokeMBB(MBB5, outputParams);
```

Figure 4.4: Compiled action example

Domain: This is an abstraction which aggregates collections of related MBBs. Its main role is therefore to provide a storage area to store the structure of the list of MBBs (called domain), the logic of the domain) list of actions), and the state of the domain (state attributes of MBBs and execution state values). They can be sued to manipulate collections of MMBs as a single unit e.g., move, suspend, resume.

Ex-ORB is a multi-protocol Object Request Broker (ORB) communication middleware service that provides client and server functionality independent of wire protocols. The end result is that is that it is possible to invoke server methods over a range of protocols including IIOP and XML-RPC. The same applies to client requests. Although Ex-ORB's implementation only supports the above two protocols, it is possible to use other protocols by developing and deploying additional MBBs at run-time. Ex-ORB's architecture is externalized and this makes it possible to access, inspect and modify it dynamically. It has been built using a Java implementation of DPRS. It has 28 MMBs classified into 11 domains as illustrated in the figure below.

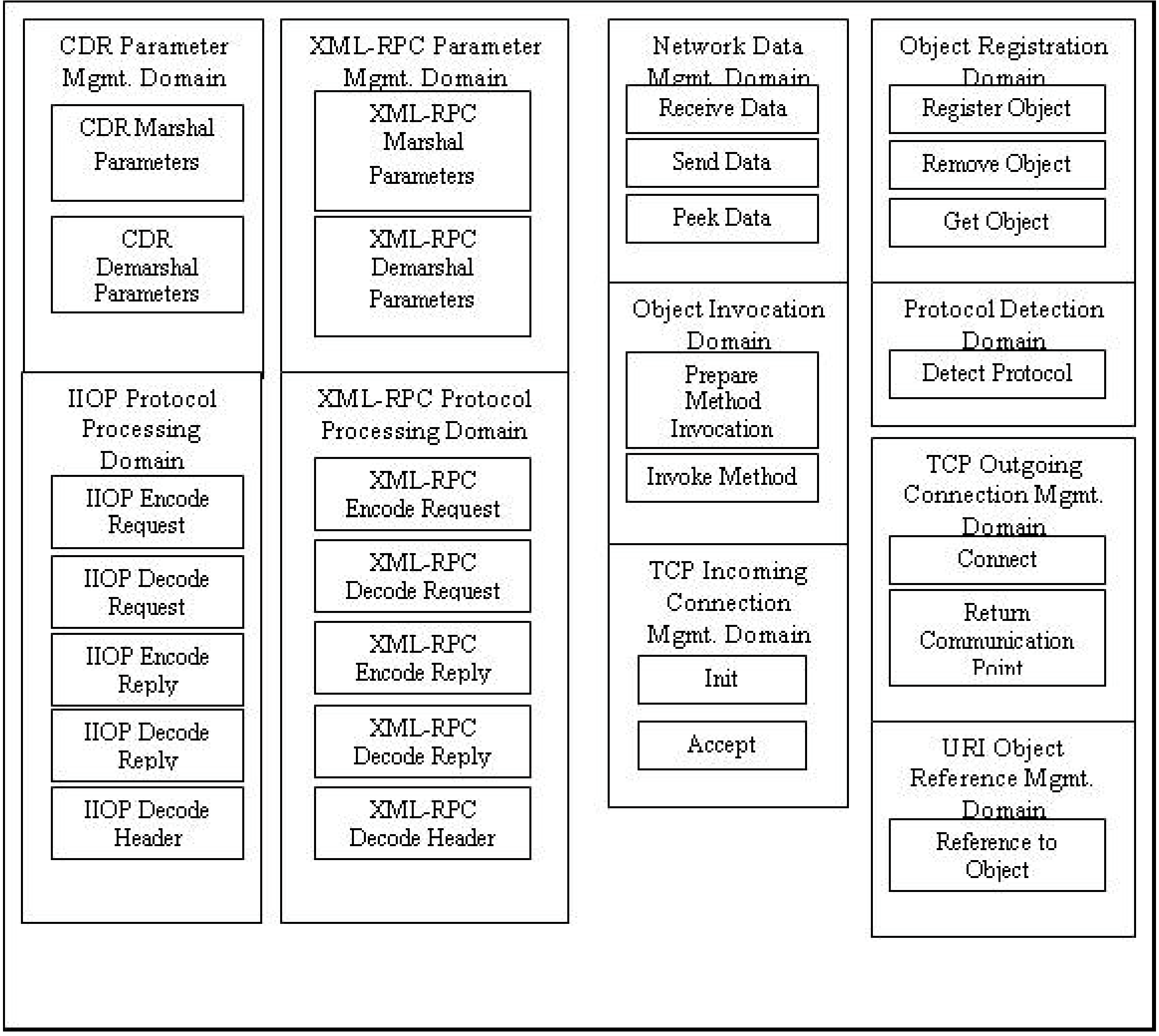

CDR Parameter Mgmt. Domain
CDR Marshal Parameters
CDR Demarshal Parameters
XML-RPC Parameter Mgmt. Domain
XML-RPC Marshal Parameters
XML-RPC Demarshal Parameters
Network Data Mgmt. Domain
Receive Data
Send Data
Peek Data
Object Registration Domain
Register Object
Remove Object
Get Object
IIOP Protocol Processing Domain
IIOP Encode Request
IIOP Decode Request
IIOP Encode Reply
IIOP Decode Reply
IIOP Decode Header
XML-RPC Protocol Processing Domain
XML-RPC Encode Request
XML-RPC Decode Request
XML-RPC Encode Reply
XML-RPC Decode Reply
XML-RPC Decode Header
Object Invocation Domain
Prepare Method Invocation
Invoke Method
Protocol Detection Domain
Detect Protocol
TCP Outgoing Connection Mgmt. Domain
Connect
Return Communication Point
TCP Incoming Connection Mgmt. Domain
Init
Accept
URI Object Reference Mgmt. Domain
Reference to Object

Figure 4.5: ExORB Structure

The functions of the 11 domains are detailed briefly below.

- CDR Parameter Management Domain: marshals and demarshals parameters using CORBA's default presentation format – the Common Data Representation Format (CDR). It has 2 MBBs as shown in the diagram to provide the marshalling and demarshalling functionality.
- XML-RPC Parameter Management Domain: marshals and demarshals parameters that are encoded according to the XML-RPC protocol.
- IIOP Protocol Processing Domain: a collection of MMBs that provide functionality to encode and decode messages that conform to the IIOP protocol.
- XML-RPC Protocol Processing Domain: equivalent to the IIOP Processing Domain but provides functionality to process requests and replies that conform to the XML-RPC protocol.
- Network Data Management Domain: processes incoming and outgoing network traffic.
- Object Invocation Domain: Uses Java language reflection capabilities to automate server method invocation. Hence developers only register their server objects and the system obtains all the information it needs instead of developers having to build the skeletons for these objects.

- TCP Incoming Connection Management Domain: processes incoming TCP network connections.
- TCP Outgoing Connection Management Domain: processes TCP connection establishment with remote peers.
- Object Registration Domain: Manages all server objects using three MBBs; Register Object, Remove Object, and Get Object.
- Protocol Detection Domain: identifies the communication protocol that incoming requests conform to (either IIOP or XML-RPC). This supports ExORB's multi protocol behaviour.
- URI Object Reference Management Domain: parses a remote object URI reference and extracts all required information to send the requests to the remote object.

The total size of ExORB's current Java implementation is 70 KB. It exports four actions: send request used on the client side, and receive request, init and register object which are for the server side. Each MBB explicitly specifies its state dependencies whose definition is provided in terms of a name and value. These tuples are stored in a storage area in the MBB domain. The state of the software is given by the union of all MBB state attributes. Hence the state of ExORB at any instance is the sum of all state attributes that are defined by all 28 MBBs.

Current work is on reduction of the performance overhead that the use of ExORB introduces. Future work will involve incorporation of functionality to guard against structural and logical semantic errors as inevitably, architectural externalization raises system integrity issues.

Evaluation

The ExORB project's main aim is to contribute towards construction of configurable, updateable and upgradeable middleware services. The externalization technique the middleware uses has potential benefits to software developers and domain experts since it offers support to phone evolution. Essentially, it makes it possible for multiple configurations, updates and upgrades to be done at runtime.

The following functional components are supported by ExORB: *Mobility and Location Awareness* and *Code Updater*. Targeting the mobile phone industry, which involves changes of location, the middleware explicitly address *mobility*. *Code Updater* functionality is provided by allowing updating and replacement of the Micro Building Blocks (MBBs) operating within the middleware. However, there is no support for a mechanism for automatically performing the updates and replacement of MBBs, but human intervention is required for this purpose.

ExORB addresses the non-functional requirements of *heterogeneity*, *openness* and *adaptability*. ExORB provides partial support for *heterogeneity* through its multi-protocol (IIOP and XML-RPC) support paradigm. The requirement of *openness* is addressed by allowing run-time updates and replacement of the small functional units (Micro Building Blocks) that comprise the middleware. Although it does not incorporate dynamic response by the middleware to changing application needs without developer-initiated actions, ExORB's provision of the ability to change software configuration at run-time implies it has great potential to offer support for *adaptability.*

4.1.3 WSAMI

This work is part of the IST Ozone project [OZONE], which investigates the design and implementation of a generic framework enabling consumer-oriented ambient intelligence applications. The main goal of the work is providing middleware support for development of ambient intelligence systems based on web services. Enabling ambient intelligent means that consumers will be provided with universal and immediate access to available content and services regardless of location and device capability. Focusing on software systems development aspect, this means that actual implementation of any ambient intelligence application requested by user can only be resolved at run time according to users' specific context. The system achieves this by having a base declarative language and associated core middleware, which supports the abstract specification of ambient intelligence applications, together with their dynamic composition according to the environment.

The solution is based on web services architecture whose pervasiveness enables both service availability in most environments and specification of applications supporting automated retrieval and composition. The work focuses on enabling seamless access to content and services any time and any where. The key feature relates to enabling the dynamic composition, possibly distributed, of requested service according to the mobile users' situation. The web service based solution enables the solution to be pervasive enough and the consistent specification of composable services. Currently, the main constituents of web services architecture are:

1. WSDL (web services description language) that is a declarative language for specifying the interface of web services.
2. SOAP (simple object access protocol) that defines a light weight mechanism for information exchange.
3. UDDI (universal description, discovery and integration) for registering web services and locating web services.

The middleware solution builds upon the web services architecture by having WSAMI (Web Services for AMbient Intelligence). The specific requirements imposed by ambient intelligence systems on the web services architecture relate to supporting the dynamic selection and composition of web services. This must further be realised in a way that enforces quality of service, while accounting for mobile and resource constraints of wireless devices. For example, significant part of the demonstrator scenario of the middleware relates to accessing public services from a wireless device such as scheduling and booking trips, restaurants, hotel, theatre, city guides and multi modal transport services. These services may be implemented via a composition of more primitive web services which may be accessed via the internet. However they may be also accessed via WLAN using adequate service discovery in the local area. Some of the services may be hosted by mobile nodes as well. The above example of composition focuses on realising application related services out of existing web services. This is straightforward to achieve in a stationary environment with internet access using the available web services technology. However, it should be noted that web serves can be hosted in stationary and mobile nodes and the correct composition needs to be achieved so as to offer quality of service to the mobile users according to his/her situation [Issarny].

Effective situation sensitive composition of web services is realized in the WSAMI environment by supporting: 1) a composition process that is distributed over the nodes hosting the component services, 2) connectors customization to enforce quality of service, and 3) exploitation of today's WLAN for both local and wide area service discovery.

In the following, we will describe key characteristics of the WSAMI environment.

Distributed composition of services: Assisting the composition of web services in stationary environment has been an active research area over the last years. The proposed solutions relates to offering an XML language for describing the overall composition process. The resulting description primarily serves as a composition design, since the implementation of the associated composite service relies on the development using specific platforms. Dynamic integration of web services is further supported by using UDDI that defines a middleware related trading service for retrieving service instances that match a given service specification (eg. WSDL specification). The above provides adequate solutions towards developing composite services whose component services may possibly be retrieved dynamically. However, the dynamic integration of services shall be embedded within the implementation of individual services and is left under the responsibility of the service developer. Dynamic selection and integration of web services needs to be automated in the ambient intelligence context. It must be supported by the middleware as opposed to handled by the application developer. The WSAMI language enriches the specification of web services with the abstract specification of services with which interaction is required The WSAMI middleware further supports retrieval of matching service instances in the environment.

Enforcing quality of service via connector customization: The solution to the systematic customization of connectors lies in the WSAMI support for: 1) abstractly specifying non-functional properties to be enforced over connectors; and 2) customisers that define middleware related web services to be integrated for enforcing a given non functional property.

Combining local and wide area service composition: The WSAMI core middleware allows for service composition both in the local and wide area over WLAN, via its naming and discovery service, which is the only middleware related service that is introduced in addition to the SOAP based core broker of the web service architecture.

The WSAMI Language: The WSAMI language enables the specification of web services so that they can be dynamically composed according to the environment in which services are requested, while enforcing quality of service. The WSAMI specification of a service relates to the service's abstract interface with possible extension of associated non functional properties. The description of web services is enriched by WSAMI with the specification of required services and non functional properties to allow for the situation sensitive composition of services as supported by WSAMI middleware.

WSAMI Middleware: The core middleware associated with web services lies in the provision of a SOAP-based core broker, including SOAP containers that are able to deploy web services and to manage RPCs from SOAP clients and dispatch them to services. In the WSAMI middleware, web services may be deployed on resource constrained wireless terminals (e.g., PDA-like devices). Deployment of web services on mobile platforms is not considered to be a major issue given the base run-time platforms being developed for such terminals, such as the Java2 Micro edition. This section focuses on the design of the WSAMI naming and discovery (ND) middleware service, which supports situation sensitive composition of services, given the services' WSAMI specification. The ad-hoc mode based and infrastructure mode based operation of WLAN is considered to support enhanced connectivity and hence enhanced service availability. The naming & discovery support for dynamically locating requested services lies in: 1) the management of repositories of services' abstract interfaces and instances, 2) locating instances of services that are reachable both in the local and wide area. The prototype implementation of the middleware is a Java based prototype of the WSAMI core middleware. The IEEE 802.11b is used as the underlying WLAN. The demonstrator application has used this middleware. The services are implemented as web services on top of WSAMI which supports for the dynamic deployment and dynamic discovery and composition of web services in mobile environments. WSAMI relieves the developers from dealing with mobility management. The WSAMI core middleware prototype subdivides into: 1) the WSAMI SOAP based core broker including the CSOAP SOAP container for wireless resource constrained devices, 2) the naming & discovery service including support for connector customization. Figure 4.6 depicts the main components of the WSAMI middleware prototype implementation on top of which web services executes. The grey components denote the available components that were reused and the components developed are bold faced. The components developed as part of the WSAMI core broker belong to any web service platform; an implementation is provided so as to allow for execution on resource constrained devices.

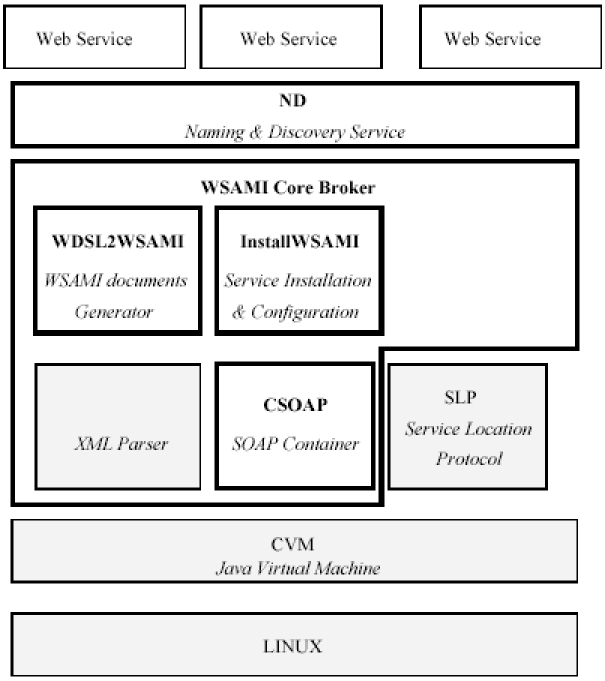


Figure 4.6: WSAMI core middleware prototype

The WSAMI core broker contains an XML parser, a SOAP container to deploy web services and manage RPC, the WSDL2WSAMI tool that generates the WSAMI documents that describe WSAMI enabled web services and InstallWSAMI tool to install and configure services on the WSAMI middleware. From the developer's point of view, the development of WSAMI-enabled web services adds minimal overhead compared to the base web services. The developer has to specify services and non-functional properties required by the service being developed using the WSAMI language.

Evaluation

By defining a minimal middleware infrastructure for the actual dynamic composition of services, ie. naming and discovery service in addition to SOAP, the middleware allows for wide deployment, and also incurs minimal overhead in terms of resource consumption. The middleware solution is similar to a service discovery platform, but most service discovery platforms operate in local area networks, while here composition of services may retrieved both in local and wide area. As it's based on web services, the availability of the service is promoted. Compared to the state of the art web services architecture, this work differs by addressing the dynamic discovery and composition of web services in the mobile context. The solution also addresses infrastructure-based and ad-hoc mode based operation.

The embedded devices that can be supported by this middleware are limited to 32-bit microprocessor/controller that has more than 2MB of total memory for the storage of VM and libraries. The middleware is not suitable for real-time applications. Since its performance is comparable to web services, it suffers from high processing overhead inherent in XML-based solutions in embedded devices.

WSAMI supports the following functionality: *Mobility and Location Awareness* and *Service Discovery. Mobility* of devices is supported within an infrastructure-based wireless network and wireless ad-hoc networks. *Service Discovery* is realized with the implementation of the Naming and Discovery middleware service.

Non functional requirements addressed by this middleware are *heterogeneity and security.* Since the middleware is based the on web services architecture, it addresses *heterogeneity* and especially, interoperability between different platforms and services (that are based on CORBA, DCOM, Java

RMI, etc.). This means that consumers will be provided with universal access to the available content and services. The ambient intelligence requirement is addressed by enabling anytime, anywhere access to applications from any terminal by binding related service instances at run-time. The WSAMI language supports dynamically retrieving instances of services matching a requested application.

4.1.4 CORTEX

The CORTEX (CO-operating Real-time senTient objects: architecture and EXperimental evaluation) project [CORTEX] addresses the emergence of a new class of applications that operate independently of human control. Key characteristics of these applications include sentience, autonomy, large scale, time and safety criticality, geographical dispersion, mobility and evolution. The key objective of CORTEX is to explore the fundamental theoretical and engineering issues necessary to support the use of sentient objects to construct large-scale proactive applications and thereby validate the use of sentient objects as a viable approach to the construction of such applications [Verı́ssimo 02]. In the CORTEX approach, applications are composed of collections of ‘sentient objects’ - mobile intelligent software components (not mobile code) that accept input from a variety of different sensors allowing them to sense the environment in which they operate before deciding how to react and affect the environmental objects. Sentient objects are able to discover and interact with each other and with the physical world in ways that demand predictable and sometimes guaranteed quality of service (QoS), encompassing both timeliness and reliability guarantees. Achieving predictability is made difficult by the characteristics of the dynamic environment in which these objects operate, including an unstable and mobile object population, unpredictable network load, varying connectivity, and the presence of failed system components. Thus, the construction of applications from sentient objects takes account of the fundamental trade-off between the existence of a dynamic environment and the need for predictable operation. To date, no comprehensive technology appropriate to the design and implementation of such applications exists. The objective of CORTEX is to explore the fundamental theoretical and engineering issues that must be addressed to deliver such technology.

In order model and design applications based on the sentient objects paradigm, the following aspects were treated as first class entities, i.e. programming model, interaction model and systems architecture.

Sentient Object Programming Model

In the sentient object programming model [Wu], software entities are categorized into sensors, actuators, and sentient objects. Sensors are defined as entities that produce software events in reaction to a stimulus detected by some real-world hardware device. An actuator is defined as an entity that consumes software events and reacts by attempting to change the state of the real world in some way via some hardware device. Both of these may be a software abstraction of actual physical devices. A sentient object is then defined as an entity that can both consume and produce software events, and lies in some control path between at least one sensor and one actuator.

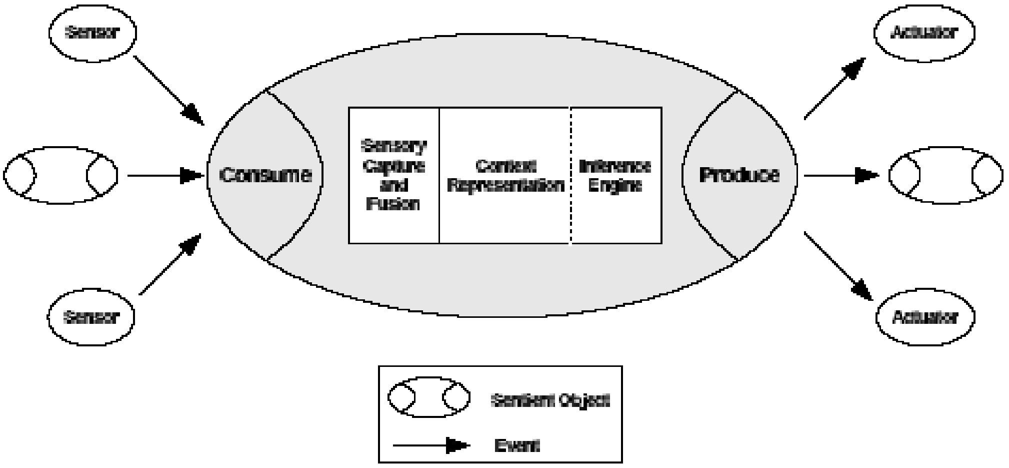


Figure 4.7: The sentient object model

Sentient objects are cooperative and communicate with each other and with sensors and actuators via an anonymous event-based communication paradigm, permitting loose coupling between sentient objects and sensors and actuators. A sentient object and its internals are illustrated in Figure 4.7. The novel event-based communication mechanism incorporated in the model is specifically designed for mobile ad-hoc wireless environments, such as those requiring spontaneous device interactions and networking, found in embedded systems (e.g., mobile computing, sensor networking, cooperative vehicles, etc.).

Interaction Model

The interaction model between sentient objects is based on an event-based or a publish-subscribe communication model. Typically, mobile sentient objects meet each other in a spontaneous and ad-hoc manner; therefore the event-based communication model needs to be operational over mobile ad-hoc networks; breaking the traditional assumptions of relying on fixed infrastructure based services. The STEAM [Meier02] definitions describe the design considerations in architecting fully distributed even- based communication middleware for proximity-based mobile ad-hoc networks. Since sentient objects perform real time interactions, the underlying event channels are QoS-aware, specifically with respect to reliability and timeliness. The Timely Computing Base [Casimiro] provides the basic support to construct QoS-aware distributed event channels.

The CORTEX middleware supports diverse application domains such as cooperating sentient vehicles [Sivaharan] and smart living environments.

CORTEX Middleware Architecture and Platform

The key research challenges that were addressed in CORTEX's architecture are: communication model, routing protocol, context-awareness, end-to-end QoS (Quality-of-Service) and fail-safety. These challenges are addressed and solutions are provided as different component frameworks (CF) [Sivaharan]. The middleware platform consists of: Publish-Subscribe CF, Group communication CF and Context CF. Component frameworks (CF) enforce the functional and non-functional properties of the system, and keep consistency across adaptations triggered by applications. A particular configuration of the middleware for mobile ad-hoc networks is shown in Figure 4.8. This configuration was targeted towards the cooperating sentient vehicles application, where context-aware autonomous vehicles travel from a given source to destinations and cooperate with other vehicles to avoid collisions, obey road side traffic lights and give way to pedestrians.

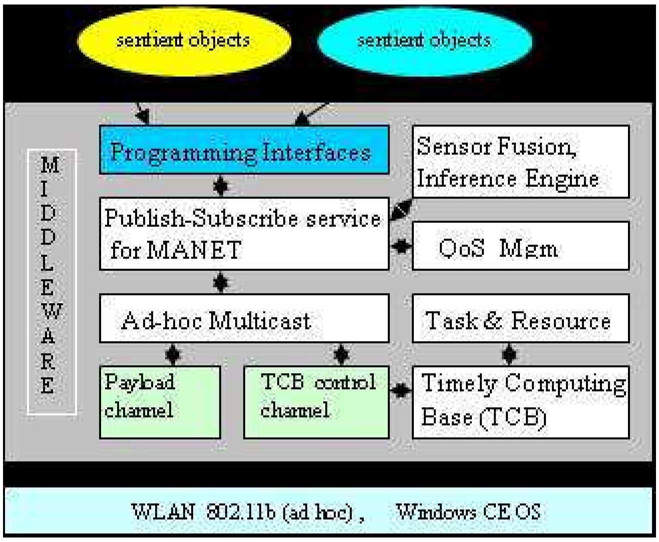


Figure 4.8: Middleware Platform for MANET

Communication model: A key challenge that needs to be addressed by the cooperating sentient vehicles application, is the suitable communication model. To address this, a loosely-coupled, asynchronous, anonymous and a fully decentralized communication model were implemented based on the Publish-Subscribe communication model. However, most of the state of the art publish-subscribe or event-based middleware are based on centralized event brokers. The CORTEX middleware takes the approach of having a publish-subscribe communication model, with all the aforementioned required properties and was especially designed for MANETs [Sivaharan].

Routing Protocol: Routing in mobile ad-hoc networks is a challenging issue because of frequent topological changes in networks. Publishers and subscribers move frequently, posing a challenge for routing of events in wireless ad-hoc networks. Multicast routing based on proactive and reactive ad-hoc routing, using shared state kept in the form of routes and adjacent information, is useful in environments with low node mobility. However, in scenarios with high node mobility such protocols are unsuitable as shared state and topology information can quickly become outdated. For this reason, as part of the Group communication CF, a Probabilistic Multicast Protocol for Wireless Ad Hoc Networks was implemented. The protocol specifically targets proximity based ad hoc environments with high node mobility and a frequently changing topology of group members. Generally different overlays are required to meet application specific requirements.

Context-awareness: Another challenge is context-awareness in highly dynamic physical environments. The fundamental challenge is that it is not possible to construct an exact 'image' (perception) of the surrounding environment. Therefore, there is a risk of wrong decisions being made based on inaccurate information. The context awareness is modeled using sentient objects. The sentient objects are objects that consume events from variety of different sources including sensors and event channels, fuse them to derive higher level contexts, reason about using an inference engine, and produce output events whereby they actuate on the environment or interact with other objects. The inference engine in the middleware is powered by a C Language Integrated Production System (CLIPS) inference engine). The Context CF provides the facility for supporting a range of inference engines and sensor fusion algorithms (that may be selected at runtime).

End-to-End QoS Management and fail safety: In cooperating sentient vehicle applications, timely event delivery and awareness of the QoS of the event channels used for inter-vehicle communication are crucial for fail-safety. Dealing with highly dynamic interactions and continuously changing environments, at the same time, with needs of predictable operations is a major challenge. The key issue in operating in uncertain environments is that timing bounds for distributed actions may be violated because of timing failures. Therefore when executing in uncertain environments, distributed operations with timeliness requirements must be able to deal with timing failures. CORTEX middleware assumes in the architecture, that it can model the uncertainty of wireless communication using a dependable timing failure detection service for distributed operations. In the middleware, this is provided by University of Lisboa's Timely Computing Base (TCB) [Casimiro]. The TCB provides the facility to monitor timeliness of event delivery on distributed event channels, thus providing estimations and awareness of timing failure probability for a given required coverage

The middleware platform shown in Figure 4.8 was implemented for the windows CE based PDA using C/C++. The OpenCOM [Blair et al, 01] reflective component technology underpins the implementations.

Evaluation

The CORTEX project proposes a novel sentient object model to address the emergence of a new class of applications that operate independently of human control. It clearly identifies the key components required for mobile networked embedded systems applications, such as the communication model, the routing protocol, context awareness etc.

The support for *mobility and location awareness* is addressed, where infrastructure-based and ad-hoc based wireless environments are considered. Applicability of the middleware includes application domains, such as smart spaces and embedded devices in mobile wireless ad-hoc networks.

The support for *heterogeneity* is limited in the middleware. The issue of *adaptability* in the environment is addressed by providing highly configurable (during deployment and at run-time) middleware, which is adaptable to environmental dynamics.

4.1.5 AURA

Today, a major source of user distraction arises from the need for users to manage their computing resources in each new environment, and from the fact that the resources in a particular environment may change dynamically and frequently. In the Project Aura [Garlan02, Sousa02] at Carnegie Mellon University, researchers have proposed a new solution to this problem. This solution is based on the concept of the personal Aura. The intuition behind a personal Aura is that it acts as a proxy for the mobile user it represents: when a user enters a new environment, their Aura marshals the appropriate resources to support the user's task. Furthermore, an Aura captures constraints that the physical context around the user imposes on tasks.

To enable the action of such personal Aura, an architectural framework is needed that clarifies which new features and interfaces are required at both the system and the application-level. The framework must have also defined placeholders for capturing the nature of the user's tasks, personal preferences, and intentions. This knowledge is the key to configure and monitor the environment, thus shielding the user from the heterogeneity of computing environments as well as from the variability of resources.

Figure 4.9 shows a bird's-eye view of the proposed architectural framework. There are four component types: first, the *Task Manager*, called *Prism*, embodies the concept of personal Aura. Second, the *Context Observer* provides information on the physical context and reports relevant events in the physical context back to *Prism* and the *Environment Manager*. Third, the *Environment Manager* embodies the gateway to the environment; and fourth, *Suppliers* the abstract services that tasks are composed of: text editing, video playing, etc.

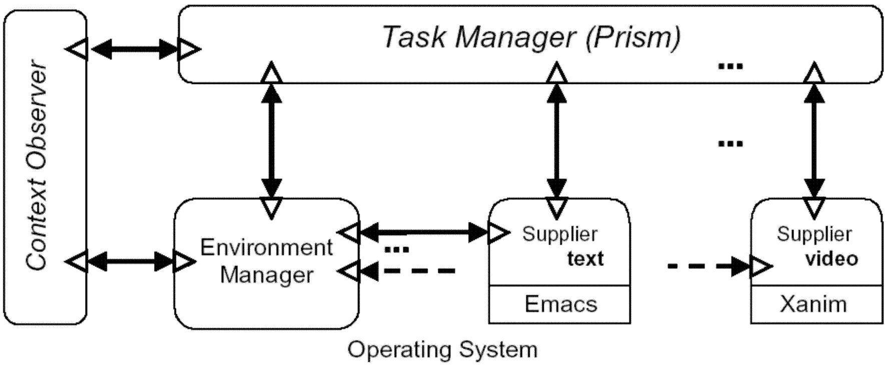


Figure 4.9: Aura's architecture

The environments are not defined by the physical boundaries of boxes or by network connectivity; they are only of administrative nature. For simplicity, the developers of the framework consider that each environment has one running instance of each of the following types: *Environment Manager*, *Context Observer* and *Task Manager*. Naturally, components of these types cooperate with the corresponding components in other environments. One environment has several service *Supplier*s: the more it has, the richer the environment is considered.

The *Task Manager* has to handle the changes of the environment and the context. It must minimize the user distractions in four fields:

- *user migration between two environments*: if a user moves into a new environment, the Task Manager has to coordinate the migration all the data the user task uses and negotiate the task support with the new Environment Manager.
- *changes in environments*: if quality of service (QoS) information provided by components becomes incompatible with the requirements of the current task, Prism has to query the Environment Manager to find a new configuration to support the task. The same operation has to be done if a monitored component dies.
- *changes in tasks*: if Prism notices that the user interrupts his current task or switches to a new task, the Task Manager coordinates saving the state of the interrupted task and instantiates the intended new task. To find out the user's intentions Prism monitors explicit indications from the user and receives event announcements from the Context Observer.
- *changes in contexts*: when constraints on the context included in the task description are not met, Prism restricts some operations. For example, when a user works with sensitive data and a second person whom is not allowed to see the information enters the room, then the display will be automatically hidden by the Task Manager.

The key idea behind Prism is the platform-independent description of user tasks. While earlier research in this area treated task as a cohesive collection of applications, Aura is describing task as a coalition of abstract services, such as "edit text" and "play video". This way, tasks can be successfully instantiated in different environments using different supporting applications.

Context Observers provide information on the physical context and report relevant events in the physical context back to *Prism* and the *Environment Manager*. There can be different degrees of complexity in each environment, depending on the number and capabilities of the sensors located in the environment.

Aura introduces a *Contextual Information Service* (CIS) [Judd03] that provides applications with contextual information through a virtual database. The information is stored or collected on demand by a distributed infrastructure of contextual information providers. The applications issue queries using an SQL-like query interface, the Contextual Service Interface (CSInt). These queries are decomposed by the Query Synthesizer to one or more lower-level queries, which are then forwarded to individual information providers. Results are synthesized and returned to the client application (see Figure 4.10).

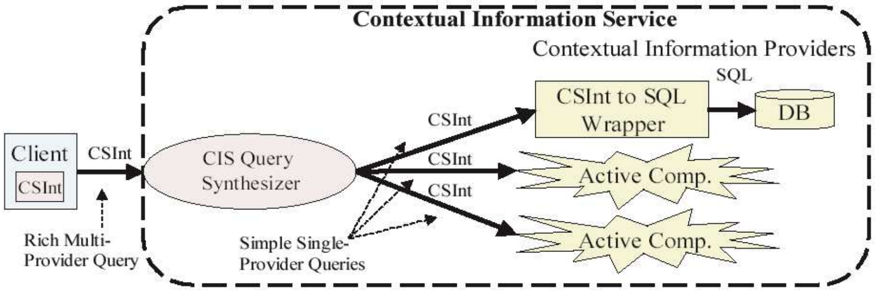

Contextual Information Service
Contextual Information Providers
SQL
CSInt to SQL Wrapper
DB
Client
CSInt
CSInt
CIS Query Synthesizer
CSInt
CSInt
CSInt
Active Comp.
Active Comp.
Rich Multi-Provider Query
Simple Single-Provider Queries

Figure 4.10: CIS architecture

The *Environment Manager* is the gateway to the environment: it is aware of which components are available to supply which services, and where they can be employed. It also encapsulates the mechanism for distributed file access. Every Supplier installed in an environment is registered with the local Environment Manager, so when a new task is initiating and needs an abstract service, the discovery mechanisms only have to look at this registry.

Suppliers provide the abstract services that tasks are composed of. Two suppliers for the same type of service can be different depending how complex they are. For instance a supplier for text editing can be more powerful than another, if the former has a built-in spell checking function. For task migration purposes, service suppliers have to possess a function to extract and map service status information to and from the Task Manager and also use a universal representation of service status. For the latter one a mark-up representation is used which contains a vocabulary of tags and the corresponding interpretation. Each service type is characterized by a distinct vocabulary of tags corresponding to the information relevant for the service, although there are commonalities across service types.

All the component types in Aura's architecture have standard interfaces, or ports (represented by triangles in 4.9). These ports only support local method calls. When Prism migrates a task from one environment to another, the deployment of the suppliers across devices may be very different. To enable dynamic reconfiguration in a transparent way to the involved components and to hide the variation of low-level interaction mechanisms from one environment to the next, Aura uses a feature called *connector*. Connectors are autonomous pieces of code assuring the interconnection between a local and a remote port. There are four types of connectors in the Aura architectural framework: one type between Prism and an arbitrary supplier, one type between Prism and the Environment Manager, and two other types connecting the Context Observer to Prism and to the Environment Manager. Each of these connector types is defined by an interaction protocol appropriate to the component type it connects.

An experimental system has been implemented based on the architecture presented above.

Evaluation

The following functional components are supported by Aura: *Event Notification*, *Mobility and Location Awareness* and *Service Discovery*. As we described earlier, events are notified to the middleware by the deployed within the system Content Observer components. The functionality of mobility and location awareness is provided by Aura’s Task Manager components, able to adapt tasks based on changes of the environment and the current context. Finally, services can be dynamically discovered by querying the Environment Manager components, holding information about the services that can be supplied to a particular task.

Non functional requirements addressed by this middleware are the requirements of *adaptability, security* and *feasibility*. **Aura’s** adaptive behavior relies on the fact that it can migrate tasks when users enter a new environment, find new configurations for supporting a task when the required QoS is not received by a particular task, and save the state an interrupted task in order to instantiate a new task. However, the middleware is limited to provide adaptation only at the task-level, ie. at the service-level. No support is provided for adaptation at the network-level, for example there are no mechanisms for distributed management of network resources according to the current network status. The requirement of *security* is also addressed with use of a SPKI/SDSI (Simple Public Key Infrastructure / Simple Distributed Security Infrastructure) mechanism for access control of users' location information.

Aura addresses the non-functional requirement of *feasibility* by providing a mechanism for ensuring that when constraints on the context (such constraints are included in the task description) are violated, then the middleware restricts some of its operations in order to stop the occurrence of the violation. However, it is not clear if the framework supports mechanisms for ensuring that generic real-time constraints, such as the available computational, memory and network resources can be included in the task description, so that the mechanism implemented in Aura can be used to evaluate these types of constraints.

4.1.6 Oxygen

MIT’s Project *Oxygen* [Oxygen] enables pervasive, human-centred computing through a combination of specific user and system technologies. It directly addresses human needs using speech and vision technologies that enable the user to communicate with Oxygen as if the user were interacting with another person. Automation, individualized knowledge access, and collaboration technologies help users to live in a comfortable world, where they can do lot of things easier or at all.

The devices, networks and applications used by Oxygen (Figure 4.11) extend the range of a user by delivering the technologies to users at home, at work or on the go. The researchers have designed two types of devices: the so called *Enviro21* computational devices (*E21s*) placed in homes, offices and cars to sense and affect the users’ immediate environment, and the *Handy21* handheld devices (*H21s*) to provide communication and computing support to the users independently of their location. Both mobile and stationary devices are universal communication and computation appliances. They are also anonymous: they do not store configurations that are customized to any particular user. The primary difference between them lies in the amount of energy they supply. The researchers also invented special software (*O2s*) running on E21 and H21 devices which are able to adapt to changes in the environment or in user requirements. To locate devices, access resources, people or services dynamic, self-configuring networks, the *N21s* networks were designed.

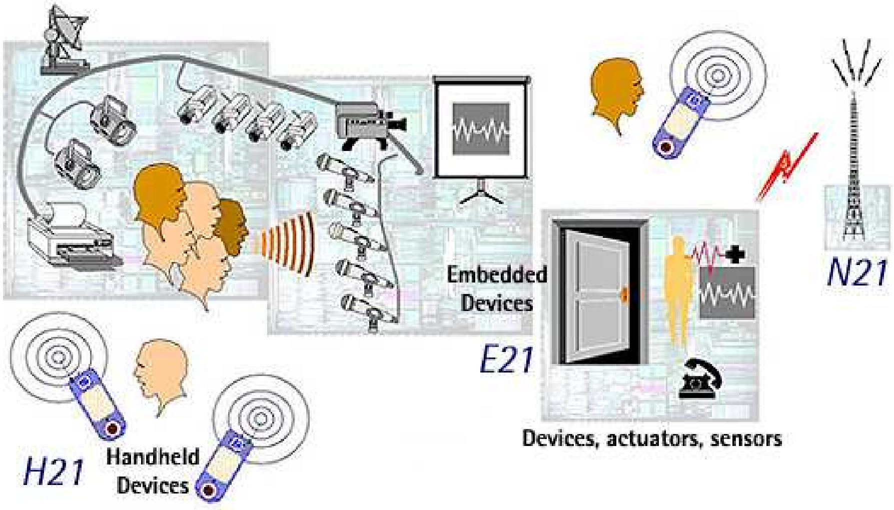


Figure 4.11: Overview of Oxygen

Collections of the E21 embedded devices create *intelligent spaces* inside offices, buildings, homes, and vehicles (see Figure 4.11). E21s provide large amounts of embedded computation, as well as interfaces to camera and microphone arrays, large area displays, and other devices. Users communicate naturally in the spaces created by the E21s, using speech and vision, without being aware of any particular point of interaction.

H21 handheld devices provide mobile access points for users both within and without the intelligent spaces controlled by E21s. H21s accept speech and visual input, and they can reconfigure themselves to support multiple communication protocols or to perform a wide variety of useful functions (e.g., to serve as cellular phones, beepers, radios, televisions, geographical positioning systems, cameras, or personal digital assistants). H21s can conserve power by offloading communication and computation onto nearby E21s.

N21 networks connect dynamically changing configurations of self-identifying mobile and stationary devices. They integrate different wireless, terrestrial, and satellite networks into one global seamless network. Through algorithms, protocols, and middleware, N21 networks realize four purposes.

First, N21s *automatically configure collaborative regions*, in which reside dynamically self-organizing collections of computers that share some degree of trust. In addition, N21s create topologies and adapt them to mobility and change.

Second, the networks used by Oxygen *provide automatic resource and location discovery* enabling the applications to use *intentional names* and location discovery through proximity to named physical objects (e.g., transmitting radio frequency beacons). With intentional names not only statically named resources can be found, but entities too that are characterized by their feature or functionality. The researchers of Oxygen describe some examples. For instance, using this solution, a full soda machine or a surveillance camera that have recently detected suspicious activity can easily be found.

Third, N21 networks *provide secure, authenticated and private access to networked resources*. The base of the security is the collaborative region, in which the devices are instructed by their owners to trust each other to a specified degree. Rules are defined, which are specifying what is allowed and what is forbidden. For instance, in the collaborative region of a meeting, guests are not allowed to use the local printer. Resource and location discovery systems address privacy issues by giving users the control over how much to reveal.

Fourth, N21s *adapt to changing network conditions*, including congestion, transmission errors, latency variations, and heterogeneous traffic, by balancing bandwidth, latency, energy consumption, and application requirements. They allow devices to use multiple communication protocols, as well vertical handoffs among these different protocols. N21s provide interfaces to monitoring and control mechanisms, which enable applications to use their own settings by a connection and vary them upon the current situation.

N21 networks use two types of routing protocols: intra-space and wide-area. Intra-space routing protocols perform resolution and forwarding based on queries that express the characteristics of the desired data or resources in a collaborative region. Wide-area routing uses a scalable resolver architecture; techniques for soft state and caching provide scalability and fault tolerance. There are also two types of name resolution solutions: early and late binding between names and addresses (at delivery time). The preceding supports high bandwidth streams and anycast, the later one mobility and multicast.

Oxygen integrates many existing protocols and solutions. For routing, it uses a routing protocol called *Grid*. It was designed for ad-hoc mobile networks, is self-configuring, requires none fixed infrastructure, is robust in the face of node failures and intrinsically supports mobile hosts. For the maintenance of an energy-efficient, ad hoc wireless network's topology Oxygen has chosen the *Span* protocol. Span nodes save power by turning off their radio receivers most of the time. A *Resilient Overlay Network* (*RON*) allows distributed Internet applications over the network to detect and recover from path outages and periods of degraded performance within several seconds. RON nodes monitor the functioning and quality of the Internet paths among themselves, and route packets according to this information. *Cord*, a scalable distributed lookup protocol for peer-to-peer networks is also integrated in Oxygen. It maps keys to nodes, adapting efficiently as nodes join and leave the system. The *Cooperative File System* (*CFS*) is based on Cord and provides highly available, read-only storage to a group of cooperating users.

For resource discovery purposes, Oxygen uses the *Intentional Naming System* (INS). It supports scalable, dynamic resource discovery and message delivery. As mentioned above, INS describes application intent in the form of properties and attributes. The resources looked for by a user or service can be either public or protected by SPKI/SDSI (Simple Public Key Infrastructure / Simple Distributed Security Infrastructure) *access control lists* (ACLs). In case of protected resources, software proxies (*K21s*) for resources and users present authorization information as an answer for an intentional query requested by an INS. The INS then compares this information to resource-supplied ACLs, and if user has access to the requested resource, INS adds it to the list of available requested resources.

For managing session-specific application state across changes in network attachment points and during periods of disconnectivity, Oxygen uses a solution, called *Migrate*. The *Self-Certifying File System* (*SFS*), a secure decentralized global file system, is needed to enable users access their data from any location.

The O2 software used by Oxygen was built considering the frequent changes of the environment. There can be many reasons for alterations, for example it can be occasioned by anonymous devices customizing to users, by explicit user requests, by the needs of applications and their components, by current operating conditions, by the availability of new software and upgrades or by failures. Oxygen's software architecture relies on control and planning abstractions that provide mechanisms for change, on specifications that support putting these mechanisms to use, and on persistent object stores with transactional semantics to provide operational support for change.

Oxygen software infrastructure consists of Pebbles, the MetaGlue, CORE, Click, SUDS and IOA [OxygenSoftwareTech]. Pebbles are platform-independent software components, capable of being assembled dynamically by the GOALS planning mechanism in response to evolving system requirements. Each Pebble's description contains a mix of formal interface specifications (method signatures, etc.), informal descriptions (of the sort found in user manuals), and arbitrary other potentially useful information, including code for test cases and demonstrations.

MetaGlue provides computational glue for large groups of software agents, such as those used in the Intelligent Room. MetaGlue clearly separates software that acts on behalf of users from software controlling spaces, provides wide-scale communication and discovery services, enables users to interact (subject to access control) with software and data from any space, and arbitrates among applications competing for resources. MetaGlue is implemented in Java, replacing the remote method invocation (RMI) mechanism with one that allows dynamic reconnection, so that agents can invisibly resume previously established, but broken connections.

CORE is a communication-oriented routing environment for pervasive computing. It structures an application as a graph of interconnected components along with a set of event-based rules. CORE supports application debugging by making all actions reversible, thereby enabling developers to test new agents or devices until something goes wrong, and then to rewind the system so as to observe the events leading up to the failure.

Click is an architecture for constructing network routers using software running on standard PC hardware. Click routers can perform a wide variety of complex tasks efficiently, including network address translation, encryption, filtering, and traffic prioritization. Conventional routers built from special-purposed hardware are not easily adaptable to these tasks. Click improves on other software routers by making it easy to configure and control packet-forwarding paths.

SUDS (Software Upgrades in Distributed Systems) is a mechanism for automatically upgrading code for objects in a distributed object-oriented database (OODB) to correct software errors, improve performance, or support new features without disrupting the service. Upgrades run just-in-time as transactions serialized with respect to all other (application and upgrade) transactions. SUDS is implemented using Thor, a large-scale distributed OODB that provides reliable and highly available persistent storage.

IOA is a language and set of tools for developing reliable distributed systems. The language enables system designers to express designs at different levels of abstraction, starting with a high-level specification of required global behaviour and ending with a low-level version that can be translated easily into code. IOA tools allow designers to simulate and reason about properties of designs at all levels of abstraction and about relationships between different levels. A code-generation tool, currently under development, will connect verified low-level designs to distributed code, thereby avoiding errors that often occur when manually transcribing designs into code.

Evaluation

Project Oxygen emphasizes the use of speech and vision technologies. It reckons the devices as intelligent resources with which the users can communicate as they would interact with another person. The concept defines fixed and mobiles devices, a software running on them, furthermore an intelligent network. These solutions provide platform-independency, routing and other networking functions, automatic code upgrade on components, support software agents and facilitate the development of reliable distributed systems.

Oxygen supports the following functional components: *Mobility and Location Awareness*, *Service Discovery* and *Code Updater*. *Mobility and Location Awareness* is offered with the deployment of intelligent networks supporting stationary and mobile (H21) devices. Mechanisms are implemented within the network in order to create at run-time network topologies and adapt these topologies according to changes in devices' location. *Service Discovery* is supported using the *Intentional Naming System* (INS) that describes the services that an application demands in terms of their properties and a list of attributes. This information is used to find dynamically the corresponding services that operate within the system. As we described earlier, *Code Updater* functionality is provided by the *Software Upgrades in Distributed Systems* (SUDS) mechanism that is used for automatically upgrading code for objects in a distributed object-oriented database.

The Oxygen project addresses several non-functional requirements. *Adaptability* is provided within the network and the operating system, while *openness* is provided by allowing adding of new software and performing software upgrades within the operating system. Support is also provided for the requirement of *security* by allowing specification of network rules, specifying which sets of users are allowed to use particular resources.

4.1.7 CARISMA

CARISMA [Capra03] exploits the principles of reflection and metadata to support the construction of adaptive and context-aware mobile applications. CARISMA define primitives to be used by applications to describe how context changes should be handled using policies. These policies may conflict. CARISMA has components to handle policy conflicts, based on micro-economic techniques. CARISMA makes use of reflection mechanisms to provide context awareness and adaptation. If more than one application is specifying context changes, CARISMA is able to handle conflicts.

Evaluation

The main aim of CARISMA is to address *adaptation* and *openness* requirements. Adaptation is achieved through reflection techniques, for inspection of behaviour over middleware and application. Openness is allowed through the use of metadata exchanged between the application and the middleware, with well defined semantics.

4.1.8 LIME

Lime [Picco99, Murphy00] is a model and middleware that extends and adapts the Linda model to the mobile environment.

Linda and Tuple Spaces. Linda [Gelernter85] is a shared memory model where the data is represented by elementary data structures called *tuples* and the memory is a multiset of tuples called a *tuple space*. Each tuple is a sequence of typed fields, such as *<"foo", 9, 27.5>* and coordination among processes occurs through the writing and reading of tuples. Conceptually all processes have a handle to the tuple space and can add tuples by performing an **out**(t) operation and remove tuples by executing **in**(p) which specifies a pattern, *p* for the desired data. The pattern itself is a tuple whose fields contain either *actuals* or *formals*. Actuals are values; the fields of the previous tuple are all actuals, while the last two fields of *<"foo", ?integer, ?float>* are formals. Formals act like ``wild cards", and are matched against actuals when selecting a tuple from the tuple space. For instance, the template above matches the tuple defined earlier. If multiple tuples match a template, the one returned by **in** is selected non-deterministically. Tuples can also be read from the tuple space using the non-destructive **rd**(p) operation.

Both **in** and **rd** are blocking, i.e., if no matching tuple is available in the tuple space the process performing the operation is suspended until a matching tuple appears. A typical extension to this synchronous model is the provision of a pair of asynchronous primitives **inp** and **rdp**, which return null if no matching tuple exists in the tuple space. Processes interact by inserting tuples into the tuple space with the **out** operation and issuing **rd** and **in** operations to read and remove data from the space.

Lime: Linda in a Mobile Environment. Communication in Linda is decoupled in time and space, i.e., senders and receivers do not need to be available at the same time, and mutual knowledge of their identity or location is not necessary for data exchange. This decoupling makes the model ideal for the mobile ad hoc environment where the parties involved in communication change dynamically due to their movement through space. At the same time, however, the global nature of the tuple space cannot be maintained in such an environment, i.e., there is no single location to place the tuple space so that all mobile components can access it at all times.

To support mobility, the Lime model breaks up the Linda tuple space into multiple tuple spaces each permanently attached to a mobile component, and defines rules for the sharing of their content when components are able to communicate. In a sense, the static global tuple space of Linda is reshaped by Lime into one that is dynamically changing according to connectivity. For example, consider a group of professors carrying PDAs, and imagine each of them inserting a business card tuple into their local tuple space, referred to in Lime as the *Interface Tuple Space* (ITS). When all professors are in the same room (or, according to Lime rules, within transitive communication), Lime's transient sharing of tuple spaces provides a view where it is as if all business card tuples were in the same tuple space, and accessible to all professors. However, when one professor leaves the room her business card is no longer accessible to the others, but it remains accessible to her. As shown in the figure below, the Lime model encompasses mobile software agents and physical mobile hosts. Agents are permanently assigned an ITS, which is brought along during migration, and reside on the mobile hosts. Co-located agents are considered connected. The union of all the tuple spaces, based on connectivity, yields a dynamically changing *federated tuple space*.

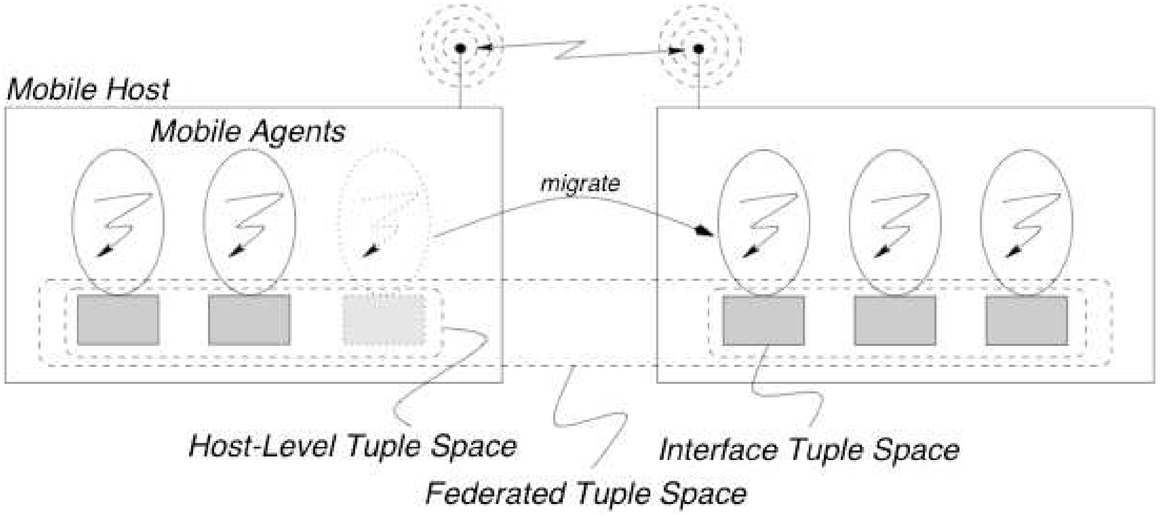


Figure 4.12: Lime's tuple spaces

Access to the federated tuple space remains very similar to Linda, with each agent issuing Linda operations on its own ITS. The semantics of the operations, however, is as if they were executed over a single tuple space containing the tuples of all connected components. In the previous example, a professor could issue a **rd** operation and retrieve, non-deterministically, the business card of any of the professors in the room.

Besides transient sharing, Lime adds two new notions to Linda: tuple locations and reactions. Although tuples are accessible to all connected agents, they only exist at a single point in the system, i.e., with one of the agents. When a tuple is output by an agent it remains in the corresponding ITS, and the tuple location reflects this. Lime also allows for tuples to be shipped to another agent by extending the **out** operation to include a destination. The notion of location is also used to restrict the scope of the **rd** and **in** operations, effectively issuing the operation only over the portion of the federated tuple space owned by a given agent or residing on a given host. For example, a professor can read the business card of another by specifying the host identifier of her colleague as part of the **rd** query.

Reactions allow an agent to register a code fragment---a listener---to be executed whenever a tuple matching a particular pattern is found anywhere in the federated tuple space. This is particularly useful in the highly dynamic mobile environment where the set of connected components changes frequently. Continuing the example above, now a professor (say, Dr. Doe) registers a reaction for business card tuples and associates a listener for displaying the card contents. As soon as the reaction is registered, it would fire immediately for each professor, since the business cards are already in the tuple space, and trigger the display of each card content on Dr. Doe's screen. Similarly, if a new professor walks in the room with a card in her tuple space, the reaction would immediately cause its display on Dr. Doe's screen. Like queries, also reactions can be restricted in scope to a particular host or agent. Nevertheless, the ability to monitor changes across the whole system by installing reactions on the federated tuple space has been shown to be one of the most useful features of Lime.

Additional information, including API documentation and source code, is available at `http://lime.sourceforge.net`.

Evaluation

Lime provides an integration of data sharing and event notification, therefore providing under a single model, both a proactive and reactive model of communication. Also, it unifies the logical mobility of agents and the physical mobility of hosts using a single programming framework. No security is provided. Lime tuple spaces have also been used to store code [Picco02], context-aware data [Murphy04], as well as to provide a middleware for sensor networks, as described later [Curino05].

4.1.9 REDS

Publish-subscribe middleware is gaining popularity because the asynchronous, implicit, multi-point, and peer-to-peer communication style it fosters is well-suited for modern distributed computing applications. While the majority of deployed systems is centralized, commercial and academic efforts are focusing on achieving better scalability by exploiting a distributed event dispatching architecture.

Beyond scalability, the next challenge for pub/sub middleware is dynamic reconfiguration of the topology of the distributed dispatching infrastructure. Companies are frequently undergoing administrative and organizational changes, and so is the logical and physical network enabling their information systems. Mobility is increasingly becoming part of mainstream computing. Peer-to-peer networks are defining very fluid application-level networks for information sharing and dissemination. The very characteristics of the pub/sub *model*, most prominently the sharp decoupling between communication parties, make it amenable to these and other highly dynamic environments. However, this is true in practice only if the pub/sub *system* is itself able of dealing with reconfiguration. In particular, all the aforementioned sources of reconfiguration affect the topology of the event dispatching network, forcing the middleware to reconfigure its behavior accordingly.

The REDS (Reconfigurable Event Dispatching System) system, currently developed at Politecnico di Milano, addresses the issue of topological reconfigurations by building on published results that propose innovative algorithms for content-based pub/sub.

The problem of dynamically reconfiguring a pub/sub system can be regarded as composed of three sub-problems that involve the:

1. Reconfiguration of the dispatching tree, to retain connectivity among dispatchers without creating loops;
2. Reconfiguration of the subscription information held by each dispatcher, to keep it consistent with the changes in the tree without interfering with the normal processing of subscriptions and unsubscriptions;
3. Minimization of event loss during reconfiguration.

The first problem is currently dealt with by specialized solutions developed for fixed large-scale networks, and smaller-scale mobile ad hoc networks. The second problem can be solved as described in [Picco03, Cugola04], by obtaining remarkable overhead reductions with respect to the solutions available in the literature. Finally, the third problem is dealt in REDS using the solution described in [Costa04], which exploits epidemic algorithms.

Finally, the very architecture of REDS is highly flexible, enabling the selection of different transport layers, event matching algorithms, and reconfiguration strategies.

Evaluation

REDS mostly addresses *fault-tolerance* and *scalability*, by enabling distributed content-based pub/sub even in scenarios where topology undergoes very frequent changes. The content-based pub/sub paradigm per se addresses *openness*, in that it enables the development of highly decoupled architectures.

4.1.10 SATIN

There is growing interest, both in research and practise in self-organising systems, systems that can adapt to accommodate a new set of requirements. Mobile systems are an extreme instance of highly dynamic distributed systems; mobile applications are typically hosted by resource-constrained environments and may have to dynamically reorganise in response to unforeseeable changes of user needs, to the heterogeneity and connectivity challenges from the computational environment, as well as to changes to the execution context and physical environment. In the SATIN (Self-Adaptation Targeting Integrated Networks) project, they argue that mobile computing systems can benefit from the use of self-organisation primitives. In particular, they argue that the application of logical mobility primitives as well as the componentisation of the system assist in building self-organising mobile systems. A component model is required by the application of logical mobility primitives for self-organisation, and discuss one. The primitives are made available to application programmers using a light-weight component-based middleware.

SATIN offers the ability to move code and data from device to device, possibly transparently to the application. Different logical mobility paradigms can be used to make this relocation happen, depending on what is most suitable. SATIN is component based, which allows the middleware to range from pretty small configuration to larger ones, but also to reconfigure dynamically based on context.

Evaluation

SATIN has components for code update and movement. It addresses requirements such as heterogeneity handling and adaptation, through its ability to reconfigure the logical behaviour or both the application and the middleware.

4.1.11 STEAM

STEAM [Meier02] is an event service, specifically designed for mobile ad-hoc networks (MANET). STEAM utilises the Implicit publish subscribe model thus does not require any separate dedicated fixed cluster of event servers for the p/s to operate. Significantly the implicit publish subscribe model allows the subscribers to subscribe to particular event types and the publishing entities to publish events of some type. The entities in STEAM are therefore fully anonymous. STEAM is fully distributed and the event filtering and event routing functionalities are fully distributed among all the publishing and subscribing devices. In addition, this organisation has the potential to avoid single points of failure making it applicable to ad-hoc networks.

STEAM uses a proximity based group communication service [Meier02] as the underlying transport mechanism for the entities to communicate. For entities to participate in group communication they must be in the same proximity and be interested in the particular type of event. The proximity based group mechanism utilised by STEAM enables mobile devices to discover each other once they are within the same geographical proximity. The proximity group communication mechanism uses a style that is based on flooding the geographical area concerned with messages at the low level in order to provide reliable message delivery. The proximity of the group defines the scope within which messages propagated are valid thereby limiting the propagation of messages beyond its geographical area. The draw back of proximity based group communication is entities can only participate in group communication once they are in close geographical proximity. STEAM supports three different types of event filters, which are Subject filter, Proximity filter and Content filters. The usage of content filters enables subscribing entities to express sophisticated queries, which enable fine grain filtering of events. The subject and proximity files are utilised to address scalability of the system. The proximity filters specify the scope within which events are propagated; specifically it describes the geographical area within which events of specific types are valid. In STEAM subject and proximity filters are applied on publisher side. Events are only routed to subscribers if both filters match. On the other hand content filters are deployed at subscriber's side and utilised when an instance of an event is received at the subscriber side to determine whether or not to deliver the event to the application.

Evaluation

STEAM is limited for publish/subscribe service operational over proximity based ad-hoc networks only. STEAM uses its own proximity group communication protocol as its underlying transport mechanism and it is tightly coupled to STEAM. The subscription language of STEAM allows filter expressions which can be used to match against the values of the set of parameters that comprise a notification. Therefore STEAM supports content filtering and thus has high expressive power. However here each device must be able to do complex content filtering and this may not be possible for resource constrained devices. STEAM utilises the proximity group communication as its underlying transport mechanism and proximity filters. The underlying transport uses a flooding mechanism to communicate therefore the entities of a given distributed application which use the middleware are preferred to be in close proximity, otherwise scalability problems can arise. This significantly limits the usefulness of the STEAM to specific application domains, where entities are in close proximity. The applicability of STEAM is limited only to entities in close proximity as it's is tightly coupled to the underlying proximity based group communication protocol. This static implementation suits only mobile ad-hoc networks where entities are in close geographical proximity.

4.2 Middleware for Embedded Systems

It is not the purpose of this survey to report on all the past work on embedded systems as this is not strictly of interest of the RUNES project. We found interesting however to report on some of the effort that has been spent in merging the effort some real time requirements middleware and Quality of Service. We report on one middleware which fulfils these functionalities.

4.2.1 ZEN

The main aim of the ZEN project [ZEN] at University of California, Irvine is to make development of distributed, real-time, & embedded (DRE) systems easier, faster and more portable; and to provide open-source real-time CORBA ORB written in real-time Java to enhance international middleware research and developments efforts. It aims to address many challenges of mission-critical distributed applications which require real-time QoS guarantees, for example combat, online trading and telecommunication systems. Building QoS-enabled applications manually is tedious, error-prone and expensive. Conventional middleware does not support real-time QoS requirements effectively. The middleware is based on CORBA specification and follows the client-server model. CORBA defines interfaces, not implementations. It simplifies development of distributed applications by automating and encapsulating Object location, Connection and memory management, Parameter (de)marshalling, event and request demultiplexing, error handling, fault tolerance, object/server activation, Concurrency and Security. CORBA shields applications from heterogeneous platform dependencies for example languages, operating systems, networking protocols and hardware.

Real-time CORBA adds QoS control to regular CORBA to improve application predictability, by bounding priority inversions and managing resources end-to-end. The policies and mechanisms for resource configuration and control in real-time CORBA include a) Processor Resources: thread pools, priority models, portable priorities, b) Communication resources: Protocol policies, explicit binding, c) Memory Resources: request buffering. These capabilities address some important distributed real-time and embedded application development challenges.

The ORB is based on Java. The motivations for using Java are easier, faster development, large programmer base and recent real-time Java specification. ZEN is the 5th generation of ORB design. The first generation was static monolithic ORB (e.g., original implementation of TAO), the second generation was monolithic ORB with compile-time configuration flags (e.g., second generation of TAO), the third generation was Dynamic micro-ORB which had small kernel of required functionality and various components are linked/loaded on-demand (e.g., newest version of TAO & ZEN, GOPI), the fourth generation was Dynamic reflective micro-ORB here the application description is used to "prime'' the ORB, loading required components at initialization- & run-time (e.g., dynamicTAO), and finally; the fifth generation is static reflective micro-ORB, here application configuration and needs are learned by dynamic reflective micro-ORB and a model-based generator builds a custom-ORB for each application, which can then be compiled and placed into ROM. The trend in new generations of ORB are clear shift from monolithic, high memory footprint ORB to low memory footprint micro-ORB kernel where core ORB features whose behaviour may vary are factored out from the kernel and provided as pluggable alternatives.

The main goal of ZEN is to provide highly configurable real time Java ORB for distributed, real-time and embedded (DRE) systems applications. The features of the middleware that enable to achieve the goals are flexible middleware configuration, easy extensibility of middleware and real-time performance. Flexible Configuration allows small footprint, load classes only as needed on-demand or at initialization time. Extensibility of the middleware allows to code and compile new alternatives and dynamically "plug" them in/out of the ORB for example new transport protocols, object Adapters, IOR formats, etc. The real-time performance is achieved by having bounded jitter for ORB/POA operations, eliminating sources of priority inversion, enabling access to real-time Java features in ORB by applications and having low start-up latency.

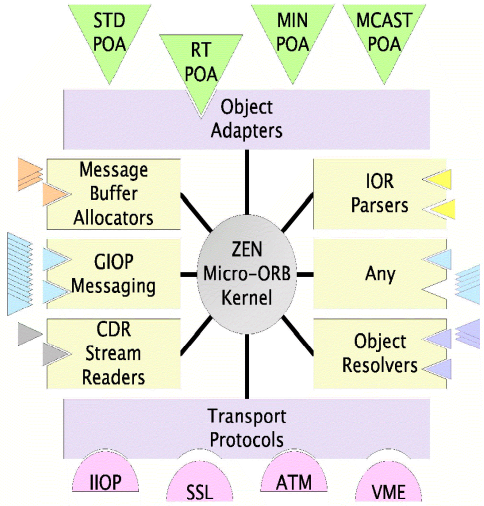


Figure 4.13: Real time CORBA with ZEN [ZEN]

The above figure shows the architecture of the ORB where core services are factored out as pluggable alternatives. As shown in the Figure 4.12, core components such as transport protocols, object adapters have pluggable alternatives. Moreover the optional features such as IOR parsers, GIOP messaging, etc., also have pluggable alternatives. This makes it possible to custom build a low memory footprint ORB depending on applications requirements.

Evaluation

ZEN is essentially a CORBA client-server based middleware for networked embedded systems. The middleware's target environment is essentially static wired networked embedded systems and the middleware crucially provides the functional requirement of *real-time* support. Since ZEN is based on CORBA, it supports multiple languages (for example C++ and Java). By using the concept of late demarshalling, the middleware is able to support multiple languages without memory or performance penalty. The middleware is highly configurable and most of the functionality of the middleware can be plugged in and out depending on application profile at deployment time. The middleware does not address the requirement of adaptability. It addresses *heterogeneity* at many levels including support for multiple languages, which is inherited by its compliance to CORBA standard. However it does not address heterogeneous wireless networks.

4.3 Middleware for Sensor Systems

This section describes some middlewares for sensor networks.

4.3.1 MiLAN: Middleware Linking Applications and Networks

Amongst the issues that current research in distributed computing has a focus on are high-level concerns such as the use of replication to improve fault tolerance and techniques that enable high-level communication abstractions e.g., remote invocation. The end-product is a set of middleware platforms whose main purpose is make lower-level functionality of the system e.g., network connectivity transparent to the application and provide a high-level coordination interface to the application programmer. Since sensor networks have characteristics such as inherent distribution, dynamic availability of data sources, constrained application QoS demands, co-operation and resource constraints, simply responding to a changing operating environment is not enough if the required QoS is to be achieved over an application's lifetime. This is because more often than not, the reactive approach leads to application having to sacrifice the quality they need over time. MiLAN argues for a proactive approach that is mainly characterised by applications actively having an effect on the entire network.

Middleware Linking Applications and Networks (MiLAN) [Wendi04] is a project at Rochdale University whose aim is to bridge the gap between what current middleware platforms offer and the need for 'proactivity' i.e. the capability of the middleware platform to enable applications have an effect on the network and the sensors themselves. It does this by making it possible for sensor network applications to specify their quality needs and subsequently makes an adjustment on the sensor network's properties to meet the applications' quality needs. It targets a new class of applications called sensor network applications. These applications are:

- data-driven: they collect and perform an analysis of data from the environment and since this data's reliability is affected by noise, redundancy and the sensors' properties, they could possibly assign a quality level on the data.
- state-based: being dynamic in nature, the applications needs regarding sensor data are bound to change over time.

MiLAN uses three sources of information namely:

1. the applications: these issue their QoS needs and ways in which these requirements can be met using different combinations of sensors to MiLAN.
2. the overall system and the user: this concerns the relative importance of different applications that run on the middleware and the required interaction between the applications.
3. the sensor network: this relates to available resources and sensors e.g., sensor energy and channel bandwidth.

Armed with the above information, MiLAN dynamically adapts the sensor network configuration to optimise the system's functionality by proactively specifying which sensors ought to send data and what roles (e.g., routers in multi-hop networks) each sensor should play. Figure 4.14 presents a high-level diagram of a system that uses MiLAN.

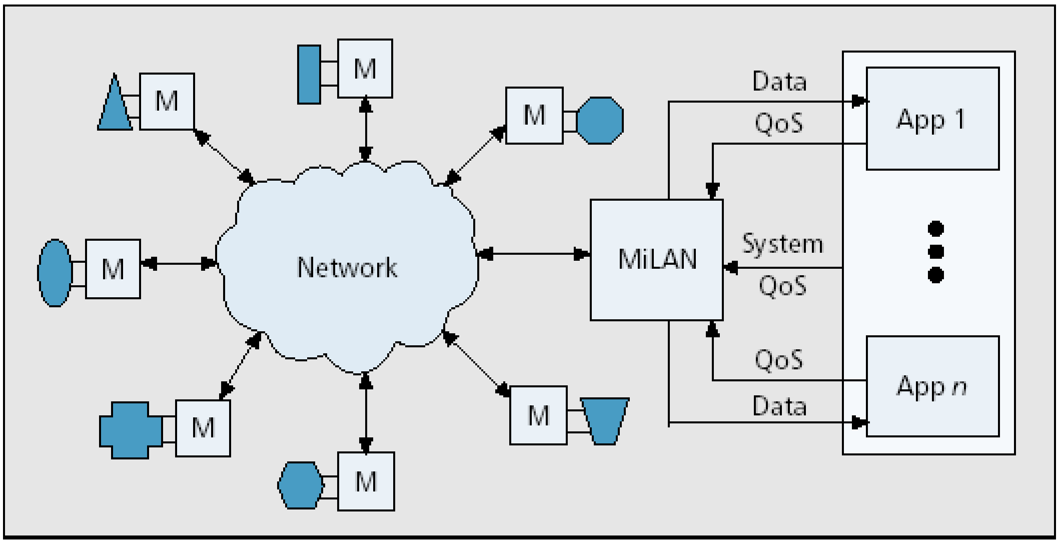


Figure 4.14: High-level diagram of a system that uses MiLAN [Wendi04]

As shown in the diagram, each sensor runs a (possibly scaled down) version of MiLAN. MiLAN receives application requirements, monitors prevailing conditions of the sensor network and dynamically optimizes sensor and network configurations to make application lifetime as long as possible. Application needs are represented by use of specialized graphs which incorporate state-based changes in application needs. Traditional middleware sits between the application and the operating system but MiLAN's architecture extends into the network protocol stack as shown in Figure 4.15 below.

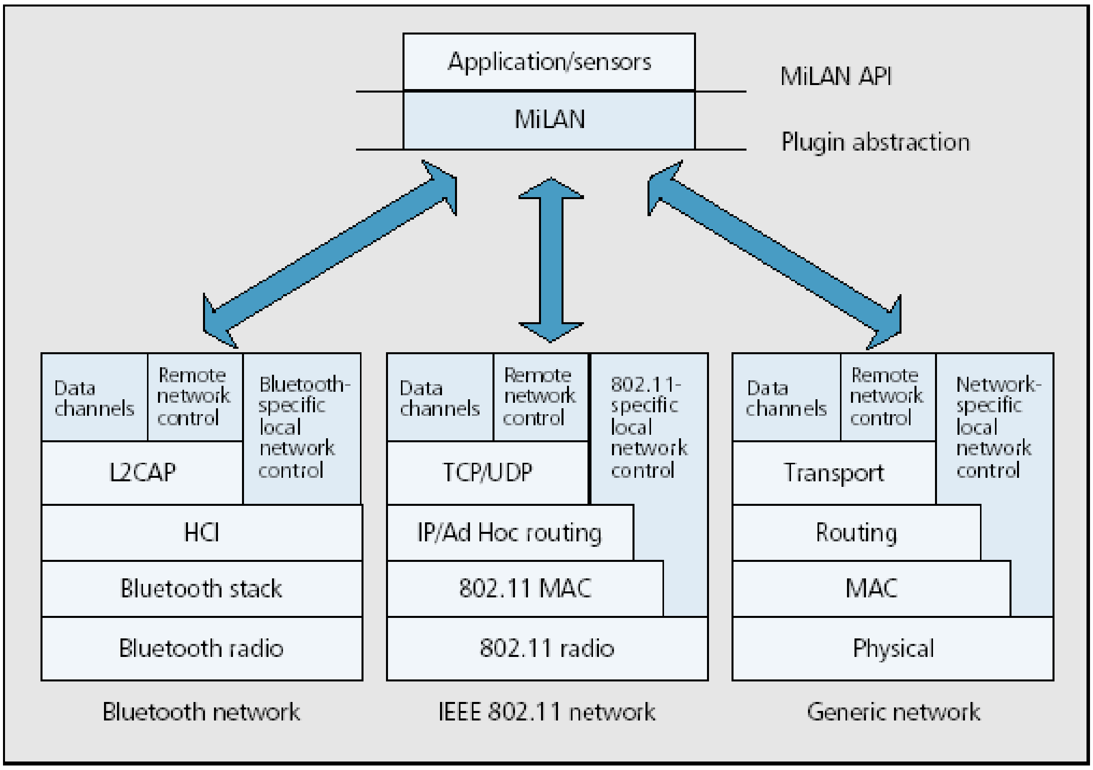
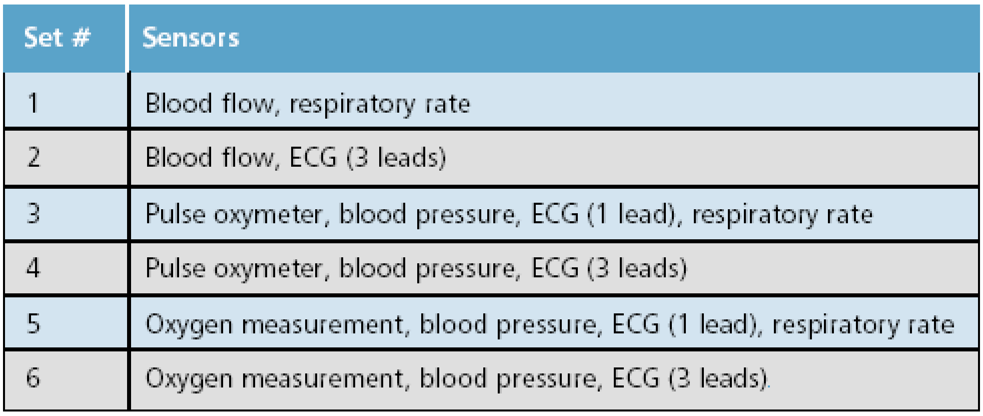


Figure 4.15: MiLAN components (shaded)

It sits on top of multiple physical networks and presents an API via which the application presents its requirements. There also exists an abstraction from the network-level functionality through which MiLAN issues commands to determine available sensors and configure the sensor network. This abstraction layer also makes it possible for network-specific plugins to perform conversion of MiLAN commands to protocol-specific commands which are subsequently passed through the network protocol stack.

Many sensor network applications receive data from multiple sources and are expected to adapt as the number of available sensors at any one time fluctuates. The assumption is that the performance of applications can be expressed in terms of the QoS parameters in different variables. To test application performance, MiLAN uses a personal health monitor which comprises variables such as blood pressure sensor and provides a quality of 1.0 (quality is mapped to a specific reliability in determining the variable from the sensor's data, with 1.0 corresponding to 100% reliability) to determine this variable. Additionally, this sensor could measure other variables, such as heart rate with a quality that is less that 1.0, in determining this variable. The quality of data providing measurement of the heart rate could be improved using high-level fusion of blood pressure measurements alongside data from other sensors such as a blood flow sensor. In order to serve the health monitor application, MiLAN receives information during initialization regarding variable that interest it, the required QoS of each variable it is interested in and the level of QoS that data from each sensor or sensor-set can provide for each variable. Figure 4.16 below shows the health monitor sensors.



Set #	Sensors
1	Blood flow, respiratory rate
2	Blood flow, ECG (3 leads)
3	Pulse oxymeter, blood pressure, ECG (1 lead), respiratory rate
4	Pulse oxymeter, blood pressure, ECG (3 leads)
5	Oxygen measurement, blood pressure, ECG (1 lead), respiratory rate
6	Oxygen measurement, blood pressure, ECG (3 leads)

Figure 4.16: Health monitor sensors [Wendi04]

All these parameters are bound to change as the application runs. The application uses 'State-based Variable Requirements' and 'Sensor-QoS' graphs to pass this information to MiLAN. An abstract State-based Variable requirements graph has the required QoS for every variable of interest to the application based on the current system state and the variables on interest. The state of these variables is influenced by an analysis by the application of previously received data. A state comprises a system state and a variable state and for every state the State-based Variable requirements graph defines the required QoS for each variable. MiLAN extracts the maximum QoS for each selected variable to satisfy requirements of all variable states since variables exist for many variable states. The State-based Variable requirements graph makes a specification of the application's minimum acceptable QoS for each variable to MiLAN. These values vary depending on the state of the environment being monitored at any one instance. MiLAN uses the information provided by the graph as well as the current application state to determine which sets of sensors meet the applications QoS needs for each variable.

Evaluation

MiLAN is an example of an "proactive" middleware. Both "proactive" and "reactive" middleware systems are categories of adaptive middleware, operating in a dynamic environment, such as a wireless sensor network. Compared to "reactive "middleware systems, which react only by themselves when changes are occurring within the network, "proactive" middleware systems enable applications to actively participate in the process of configuring the network where the middleware operates. Such a reactive behavior is supported by the MiLAN middleware, by allowing applications to specify their QoS requirements and also the policies that dictate how the middleware should react when the applications requirements are not met in a given network instance. The middleware uses this input from applications and trades off applications' performance with the network cost. At the same time, the middleware retains the separation between the policy that specifies how the middleware should react (the policy is provided by the application) and the mechanisms that implement the policy (these mechanisms are implemented in the middleware).

Given the dynamic nature of a wireless sensor network, MiLAN uses services discovery protocols, such as SDP [Avancha02] or SLP [SLP]. This is needed to discover new nodes and be aware when nodes become inaccessible (either as a result of mobility or loss of battery power). The discovery protocols must return the data type that a discovered node can provide and modes in which it can operate, the node's transmission power levels and residual energy level. Using this information from each currently available node, the network plug-in in MiLAN's architecture determines which sets of nodes can be supported by the network. This mechanism aims at satisfying the requirement of feasibility. More specifically, in a given network instance, only the subset of nodes that can be supported by the network are considered by the middleware as candidate nodes that could be used to support the application's requirements.

4.3.2 Impala

Wireless sensor networks typically comprise a number of geographically dispersed and as a result, they present a new domain of parallel/distributed computing which is gaining considerable interest amongst researchers in academia and industry. Even though sensors have little computational and communication resources, they are expected to sense their environmental parameters often for long periods of time and communicate it via the network to other nodes or to a base station. This uncertainty in sensor networks (communication and computation-wise) presents the need for a highly-tuned operation in which implemented mechanisms are able to handle a range of possible parameter values. Incorporation of adaptation in sensor network systems presents a panacea to the plethora of issues that result from the modesty in computation/communication resources in wireless sensor networks.

Impala [Liu03] is a middleware system that was designed as part of the ZebraNet mobile sensor network. Its inspiration was out of the observation that sensor networks are long-running and autonomous. They therefore have a requirement for a system that manages and fine-tunes applications on each node to ensure reliability and ease of upgrades during the application's entire lifetime. It proposes a system that serves the role of a lightweight event and device manager for each mobile sensor node. The ZebraNet project comprises a mobile sensor network system whose objective was to explore ways to improve wildlife tracking technology by use of energy-efficient tracking sensors and peer-to-peer communication amongst nodes. Sensors are positioned on free-ranging wildlife to help perform long term observation of migratory tendencies in animals. It also handles connectivity issues in mobile nodes by enabling interoperability of a set heterogeneous protocols.

As shown in Figure 4.17 below, Impala adopts a layered approach in which the upper layer comprises all the application protocols and programs. Their task is to gather environmental data and subsequently route it to the base station in a peer-to-peer network. Impala's lower layer comprises three middleware agents:

- the Application Adapter: adapts the application protocols at run-time to the changing environmental conditions.
- the Application Updater: receives and transmits updates wirelessly, installing them on the node.
- the Event Filter: does capture and dispatch of events to the above two layers.

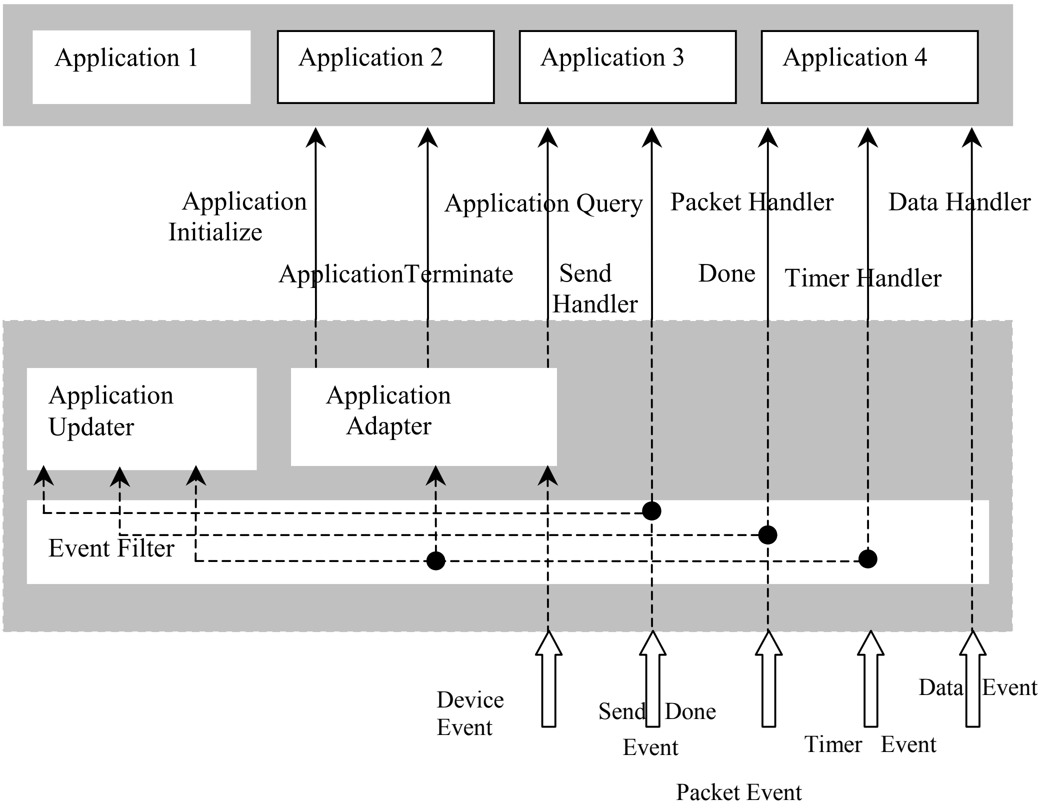


Figure 4.17: Impala's Layered Architecture

Impala classifies the events it handles into five categories: i) **Timer Event** which signals that a timer has gone off, ii) **Packet Event** which signals that a packet has arrived, iii) **Send Done Event** which signals that a packet has been sent or there as been a failure sending it, iv) **Data Event** which signals when a data sample from the sensor is ready to read, v) **Data Event** which signals that a sensor failure has occurred.

As illustrated in Figure 4.17 above, Impala has an event-based programming model in which the applications, the Application Adapter and the Application Updater are programmed into event handlers that are invoked by the Event Filter when packets are received. The user library comprises some programming utilities that are accessible to the adapter and updater. The Networking utilities make it possible for applications to send asynchronous packets after which they generate **send done** events to the event filter. Timer utilities give applications the ability to set up timers e.g., to send packets ate regular intervals. Device utilities on the other hand give applications control over hardware e.g., turning the transceiver on and off. Impala defines a uniform storage image that comprises a list of data generated from the local sensor and a list of data received from other sensors. It also maintains a list of data that has been routed to the base station successfully.

In the following, we will describe in more detail the main functional components of Impala, the *Application Adapter* and the *Application Updater* components.

Application Adapter: Impala's event-based application programming model uses this agent to respond to the events mentioned above. Whenever events such as timer events signal that some action has taken place, the Application Adapter may query the system or application states to determine if adaptation should occur. Although programming the applications to make adaptation decisions would be more flexible, Impala's Application Adapter makes adaptation decisions as unlike applications, it has a global overview of the entire system state. It defines a set of application parameters (only known to a particular application) alongside another set of Impala parameters which represent runtime states. The Application Adapter performs two kinds of adaptations. In the parameter-based adaptation, each application tracks a subset of application parameters and reports their changing values. An Application Parameter Table in the Application Adapter agent maintains a list of which application tracks what parameters. The adapter periodically queries the active application for parameter values that it tracks, gets the system parameters at that instant and examines the rules that activate a switch. Should any rule be met, the adapter triggers a switch. In order to prevent the periodical querying operation from competing for network resources with the nodes, application queries occur at the end of the network activity period.

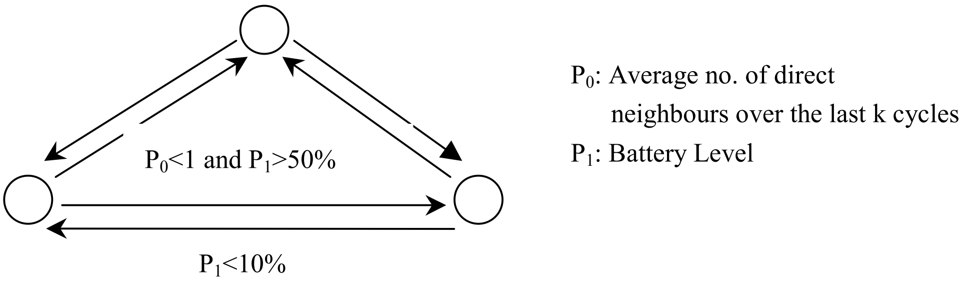

P_0<1 and P_1>50%
P_1<10%
P_0: Average no. of direct neighbours over the last k cycles
P_1: Battery Level

Figure 4.18: Adaptation Finite State Machine

The adapter makes adaptation decisions by examining the Adaptation Finite State Machine (AFSM) which has states that suit different applications as shown in Figure 4.18 above. The arrows represent adaptive transitions while the parameter expressions above each arrow are the conditions under which adaptation occurs.

The device-based adaptation occurs when the Application Adapter performs periodic queries and discovers a piece of hardware that does not respond. An Application Device Table has records of which applications use which device. Impala tracks up to eight hardware devices although the ZebraNet project only has three - the GPS transceiver, a short-range radio and a long-range radio.

Application Updater: Impala's updater is designed to meet a unique set characteristics that were observable in the ZebraNet project (indeed these characteristics also exist in other sensor networks). These include i) High Mobility - sometimes moving in clustered patterns, ii) Constrained Network Bandwidth as a host of sensors collect data from the environment and transmit it back to a base station, iii) Wide Range of Updates ranging from bug fixes and application enhancement code to addition/deletion of applications. The updater's role is to use a mechanism that achieves updates for mobile wireless sensor networks that have resource constraints. To manage software, it stores complete and incomplete update versions in the code memory. The complete versions are logged awaiting execution while incomplete versions are logged awaiting resumption should the update be available again. An on-demand strategy is used to transmit updates and sensor nodes periodically share their version information with others and only exchange code upon request.

Evaluation

Impala exploits mobile code mechanisms to adapt the functional aspects of the middleware that runs at sensor nodes. These sensor nodes are implemented in a modular way facilitating updates that are small and that introduce little transmission overhead. The novelty in its contribution stems from the fact that that it explores a software architecture which suits minimal performance and energy impact for code that runs on sensor nodes that are typically energy constrained. Furthermore, it adopts a middleware layer that is able to update and adapt applications at run-time via new protocols being plugged in. Currently, Impala's adaptation is based on local states of local sensors although there are plans to include the entire sensor network.

ZebraNet's first version is rather simplistic as it has the application running only a communication protocol which transmits data wirelessly to the base station. In the absence of a more complex application software that incorporates event filtering, noise filtration and sensor data fusion, it is difficult to judge Impala's performance as a sensor network middleware.

Adaptability is also provided by the adapter agent which disables protocols that need to use a failed device should a sensor fail and switches to an application that does not need a failed sensor device should an active application be disabled.

Impala does not address *heterogeneity* issues since its implementation is on a single hardware platform in which a hardware node comprises a GPS, a transceiver and a micro-controller CPU with 1-8 MB of volatile storage. It is being prototyped for the HP/Compaq iPAQ Pocket PC handheld which runs the Linux operating system.

4.3.3 AutoSeC

Automatic Service Composition, AutoSeC [Han01] is a middleware framework whose focus is on provision of support for dynamic service brokering for effective utilization of resources within a distributed environment. Distributed applications have QoS requirements that can be translated into the underlying system-level resources and in this respect, AutoSeC performs resource management within a sensor network by granting access control to applications in order to meet QoS requirements on a per-sensor basis. Crucially, meeting these requirements entails the AutoSeC middleware dynamically choosing a combination of information collection and resource provisioning policies from a given set which is based on the users' needs and the system state. Since the choice of policies is done by AutoSeC, the application developer or system administrator is relieved from the tedious task of having to make a choice from the set of policies that are available.

AutoSeC comprises a middleware infrastructure that offers support for adaptive selection of optimal information collection and resource provisioning service combinations. This is done in a transparent manner, ensuring that the application is not involved in the complex decision-making process. Figure 4.19 below presents an overview of the AutoSeC framework.

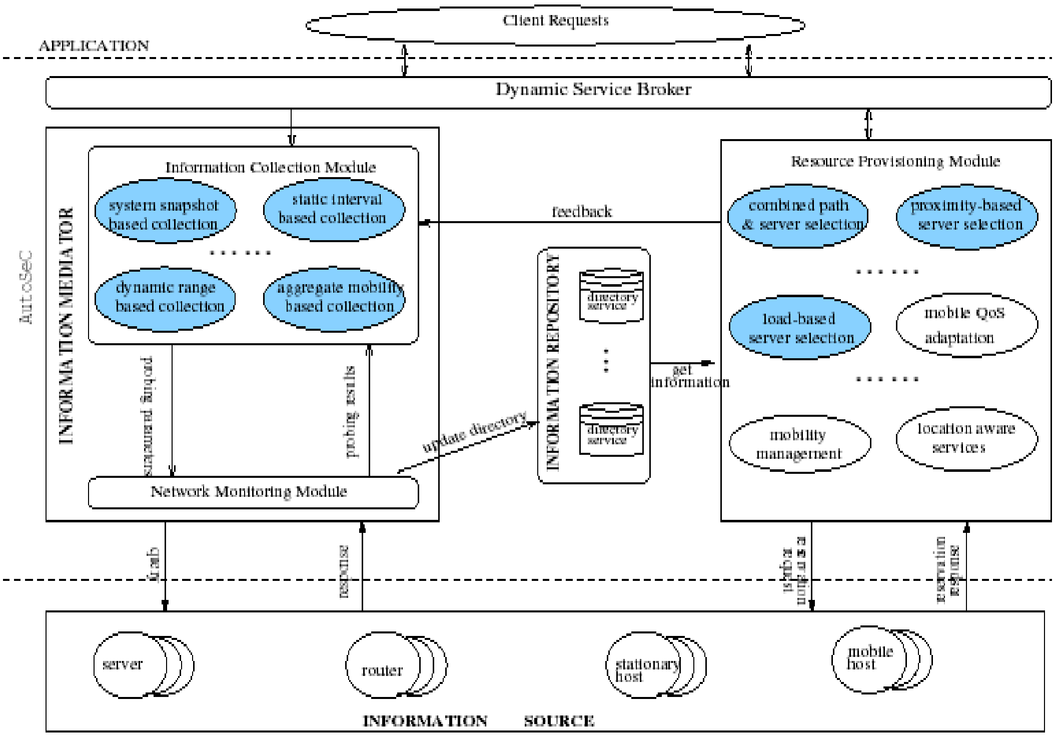


Figure 4.19: The AutoSeC Dynamic Service Broker Framework

AutoSeC's *directory service* is centralized and stores system state information concerning three categories of parameters: i) network parameters, such as bandwidth and end-to-end delay, ii) server parameters, e.g., buffer capacity, CPU utilization, disk space, and iii) client parameters, e.g., client connectivity, capacity etc. Each of these parameters can be represented by either one value or a range of values with an upper and lower value.

The *information collection module* determines whether to update the information repository with the current system image. This is influenced by the rate at which samples are collected. The higher the sampling interval, the higher the accuracy of the *directory service's* information, and the higher the overhead introduced to the system. Hence the *information collection module* has to maintain a balance between the information accuracy and the overhead introduced by the directory service maintenance.

The *resource provisioning module* uses information received from the directory service regarding the current system state to perform resource allocation. It uses intelligent mechanisms to choose appropriate resources, e.g., servers, routing paths to service QoS based requests. It takes into account the current network and server utilization parameters provided by the directory service to allocate resources in a way that optimizes overall system performance. Once new clients are assigned a path and/or server, they set up a connection to the assigned server on the assigned path. Routers and servers along the assigned path check their residual capacity and could perform either of two actions; admit the connection and reserve resources or reject the request. Once the connection of a client to a server terminates, the client makes a termination request and the resources along the formerly assigned path are reclaimed by the *resource provisioning module.*

The *network monitoring modules* are distributed with each module monitoring parts of the entire network. Each module probes routers and servers to collect system state information. These probes consolidate the sample values collected by the routers and servers before they forward them to the monitor. The *network monitoring modules* finally transmit the information to the *information collection module* which performs updates on the directory service.

AutoSeC's *dynamic service broker* performs all the decision-making functions regarding what combinations of resource provisioning and information collection policies are those that satisfy user requests and match current system conditions. It also decides when to switch these policies at run-time.

The AutoSeC system implements a number of strategies for information collection and resource provisioning. These mechanisms are discussed below.

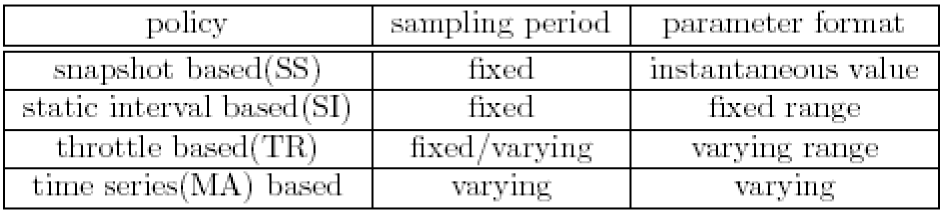
Network and Server Information Collection Policies:

These include the:

- System Snapshot Based Information Collection in which all the information about the residual capacity of the network nodes and server nodes is based on an absolute value obtained from a periodic snapshot.
- Static Interval Based Information Collection in which the residual capacity information is collected using a lower bound L, and an upper bound H with the actual value assumed to be uniformly distributed across this range. Hence the expected value is computed using (H/L)/2 9 [Apostolopoulos98].
- Throttle Based Information Collection in which the *directory service* has a range of upper and lower bounds that vary dynamically for the parameter that is monitored. In this policy, should a sampled value fall within the current range for a specific period of time, the range is tightened by a pre-defined ratio. On the other hand, should the current sampled value fall outside the range for a specific period of time, the range is relaxed.
- Time Series Based Information Collection which implements a two-phase information collection procedure. This policy uses simple statistical analysis based on time-series. The first phase derives a range of values so that the deviation of the predicted values from the observed samples is within a certain confidence level. Depending on the size of the range and the associated confidence level, this policy determines a bound on the rate at which

sampling occurs. The second phase involves a dynamic adjustment of the range and the rate of sampling by the information collection process. This is entirely based on burstiness of incoming traffic.

Table 4.1 below provides a summary of the Information Collection Policies implemented in the AutoSeC framework.



policy	sampling period	parameter format
snapshot based(SS)	fixed	instantaneous value
static interval based(SI)	fixed	fixed range
throttle based(TR)	fixed/varying	varying range
time series(MA) based	varying	varying

Table 4.1 Family of Information Collection Policies [Liu03]

Resource Provisioning Policies:

This type of policies helps in utilizing network and server resources, such that the QoS requirements of applications running on top of AutoSeC are met. Policies of this type are the following:

- Server Selection (SVRS) which implement mechanisms that enable clients to discover an optimal replica for a given service/document that is replicated across distributed servers. Two server selection policies are evaluated in AutoSeC: i) the Least Utilization Factor Policy (SVRS-UF) which selects the server that has minimal utilization in terms of four observable parameters – CPU cycles, memory buffers, I/O bandwidth and network transfer bandwidth, and ii) the Shortest Hop Policy (SVRS-HOP) which chooses the nearest server in terms of the number of hops between the replication source and destination. In case of a tie, the SVRS-UF policy randomly picks one server while the SVRS-HOP policy selects the least loaded server.
- Combined Path and Server Selection (CPSS) which are based on the premise that in order to achieve better system-wide utilization, applications that are sensitive to QoS parameters need high-level provisioning strategies which address route selection and server selection in a unified way. Therefore, given a client that makes a request with QoS requirements, its algorithm chooses a server and links which maximize resource utilization. Load balancing is therefore possible between replicated servers and among network links.

Simulations have been carried out to assess AutoSeC's performance and to determine the most optimal combination of information collection policies with resource provisioning policies under a range of system loads.

Evaluation

AutoSeC, as MiLAn, is another example of "proactive" QoS-aware middleware. Having as input the applications' requirements and the network state, the middleware implements a decision-making process in order to choose which information collection and resource provisioning policies should be enforced. Compared to MiLAN, where policies are specified by the application, AutoSec's policies are only defined a priori within the middleware and in consequence only the middleware can perform resource allocation. However, it in not clear if and how the set of information collection and resource provisioning policies can be updated. This would be required in cases where the existing set of policies can no longer cater for the applications' requirements.

Resource management is realized through the *resource provisioning module's* run-time allocation of resources to new clients and reclaiming of resources once a new client terminates a connection to a particular server.

Future research directions in AutoSeC include the provision of a set of network management middleware services targeted at applications that are network-aware. Essentially, this would involve adaptation of current network management mechanisms for provision of a dynamic and relevant set of information to the AutoSeC middleware. Additionally, an extension of the current centralized directory services to incorporate distribution would enhance scalability of the framework.

4.3.4 DSWare

Despite the fact that sensor networks are characterised by limited computing resources, they must be operational for long periods of time. This implies that efficiency in power consumption is very important in order to prolong a sensor network system's life-time. Apart from being unreliable, wireless communication introduces the most significant overhead in power consumption and in order to optimise data transmission from sensor nodes, sensor networks initiate data collection and transmission via queries and subscriptions. Sensors also transmit a large amount of data and to minimize unnecessary transmission of data, nodes sometimes work together or use intermediate sensor nodes to do filtration of data before it gets to its destination i.e. the base station. The argument that forms a basis for the design of DSWare is that addressing these issues requires an approach that 'builds trust' on a group of sensor nodes as opposed to any single node.

Data Service Middleware (DSWare) [Li03] is a specialized layer that does an integration of various real-time data services for sensor networks and in so doing, provides a data-base like abstraction to sensor network applications. It offers support for group-based decision making and reliable storage to improve real-time system performance, reliability of aggregated results and reduction in communication overhead. DSWare's design as a specialized layer provides an abstraction of data services to applications. This makes it possible for application programmers to avoid re-implementing the common data service part of applications. It also hides come characteristics of sensor networks such as unreliability in sensors and their communication, complexities such as group coordination and data fusion and the large volumes of data by providing a generic data service to sensor network application programmers.

As shown in Figure 4.20, DSWare provides a data service abstraction to the applications that run on it.

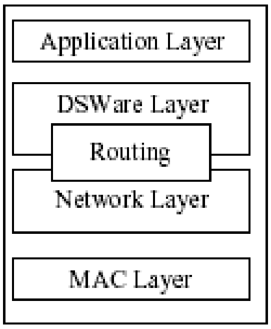


Figure 4.20: Software Architecture in Sensor Networks

Its architecture separates routing operations from it and the network layer because its group management and scheduling components are used to enhance power and real-time-awareness of the routing protocols. Figure 4.21 below, illustrates DSWare's framework.

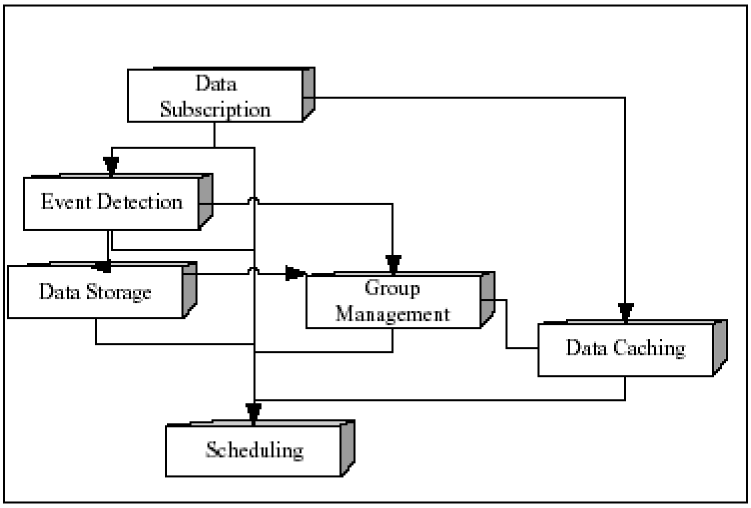


Figure 4.21: The DSWare Framework

As can be seen in the above figure, DSWare is comprised of several components each of which serves a well-defined need. Each of the components is described below.

Data Storage: DSWare's Data Storage component stores data according to the semantics that is associated with the data. It has a data look-up operation and offers support for robustness should there be node failures. It also has operations via which data that is correlated can be stored in geographically adjacent regions. This has two advantages: it enables data aggregation and also makes it possible for the system to perform in-network processing.

To facilitate data look-up, DSWare maps data to physical storage using two levels of hash functions. At the first level, the hash function maps a key (a unique identifier which each data type has) to a logical storage node in the overlay network. As a result of this operation, storage nodes form a hierarchy at this level. The second level involves mapping a single logical node to multiple physical nodes such that a base station performing a query operation has the data fetched from one of the physical locations.

There is a big risk in mapping a given data type to a single node as this data could be lost as a result of node failure. Furthermore, mapping data to a single node in the sensor network causes bursts of traffic to the node which causes collision and a higher rate power consumption. DSWare uses an approach in which data is replicated in multiple physical sensor nodes which can be mapped onto a single logical node. Load balancing occurs since queries can be directed to any one of the physical nodes and the lifetime of individual nodes is prolonged since power consumption is substantially reduced. With replication of data amongst multiple nodes come consistency issues. DSWare adopts 'weak consistency' to avoid peak time traffic since only the newest data amongst nodes is bound to lack consistency. This new data is propagated to other nodes and the size of inconsistent data is bounded so that replication occurs when the workload in individual nodes is low.

Data Caching: The function of this component is to provide multiple copies of data that is most requested. DSWare spreads cached data out over the network so that availability is high and query execution is faster. A feedback control scheme is used to dynamically decide whether or not copies should reside in frequently queried nodes. This control scheme uses inputs such as proportion of periodic queries, average response time from data source etc to guide nodes in making decisions about whether or not a copy should be kept. This component also monitors the usage of the copies to decide whether to increase/reduce the number of copies or move them to a new location.

Group Management: This component has the task of ensuring that there is cooperation between the functionality of various nodes. When nodes in a group provide a value that is agreeable to most of them, this value has a higher confidence that when there is disagreement between nodes. Secondly, Group Management also makes it possible for the system to notice nodes that provide suspicious values and exclude values received from them in the performance of computations. Some computations e.g., those that involve speed and movement require the use of more than just a single sensor to accurately compute. The Group Management component also makes it possible for energy-saving actions such as putting some nodes to sleep. DSWare forms groups as a query is sent out and dissolves them at the expiry of the task. To accomplish this, a group formation criterion is sent out to a queried area and nodes join the group by checking whether or not they match the criterion. Groups could either be fairly stable once nodes join it e.g., those that measure temperature. Others could be dynamic e.g., those that track a vehicle's motion and in this case, changed criterion gets broadcast persistently in a small location and nodes can join or leave the group as the target is in motion.

Event Detection: DSWare uses SQL-like statements to register and cancel events as this approach provides applications with a simple interface. Applications can insert events to the sensor network without modifications in the code. This is beneficial to applications which need event detection services regardless of the data service middleware that provides the service. This approach may not be the most optimal in certain cases as it inevitably introduces a parsing overhead (parsing consumes memory and processing power) which DSWare has to perform once the SQL-like statements are issued. A more energy saving approach would be to provide method signatures to applications as sensors have inherent limitations in processing power and memory capacities. DSWare's argument for the use of SQL-like statements is that they are expressive and provide the desired flexibility to satisfy many event notifications.

Data Subscription: Data subscription queries provide sensor networks with their own characteristics such as data feeding paths, stable traffic nodes for the paths and possible merges for the feeding paths. When many base stations make subscriptions for data from the same sensor node, the Data Subscription service in DSWare puts copies of the date at intermediate nodes in order to save on communication costs between the node and several subscribing base stations. It also changes the data feeding paths dynamically e.g., when it detects the proximity of two paths, it merges them by placing a copy of the data at an intermediate node which then transmits data to the subscribers at their respective requesting intervals.

Scheduling: This component plays the role of scheduling services to all DSWare components. It provides two scheduling options: energy-aware and real-time scheduling. The default scheduling mechanism is the real-time option (EDF, EDDF, with or without admission control) since most queries in sensor networks are real-time in nature. When requirements for real-time scheduling have been satisfied, DSWare can also apply the energy-aware mechanism. Importantly, the scheduling schema is specified by individual applications based on their most primary concerns.

Evaluation

DSWare aims at providing an abstraction of data services to applications since current sensor networks applications have to implement the entire stack of application-specific data services. It frees applications from low-level tasks and enables them to use data from sensor nodes using interfaces that are similar to conventional databases. Its implementation is similar to AutoSec's data service abstraction but the subtle difference is that DSWare uses a group of geographically close sensors to provide data.

DSWare handles the run-time change in data sources by using the *Group Management* component to handle stable and dynamic groups of nodes which either leave or join groups.

Future work in DSWare involves an extension of the Data Storage component's design to enable mapping of data to regions that are within the immediate geographical vicinity. This capability in DSWare has huge potential to promote data aggregation and in-network processing.

4.3.5 "Adaptive Middleware for Distributed Sensor Environments" [X.Yu03]

Another adaptive middleware for sensor networks is described in [X.Yu03]. The basic idea here is that resource/quality trade-offs can be exploited dynamically in order to reduce energy consumption in the context of information collection in distributed sensor environments. Furthermore, the proposed middleware exploits the predictability of sensor readings, stemming from the predictability of the real-world phenomena the sensors monitor, in order to further reduce communication overhead and thus energy consumption.

Two main architectural components within the middleware provide its adaptive behavior. The first component, the *adaptive precision setting* component has an input data from a translation module, which is used to translate a given application quality (AQ) requirement to corresponding quality requirements on data (DQ) collected by sensors. The *adaptive precision setting* component then expresses the data quality obtained from the AQ-DQ translation as a tolerable measurement error and communicates this information to the sensors. In their turn, on the basis of the error tolerance communicated by the adaptive-precision-setting process on the middleware, sensors implement new filters on sensor-generated data.

In addition to the *adaptive precision setting* component just described, the framework includes a second component, the *prediction module,* that executes in parallel with the precision-based module. In prediction-based adaptation, the middleware compares a sensor's current reading with its previous reading to determine whether an update is needed. If the sensor and server sides of the middleware agree that the two readings are the same, then a message exchange isn't needed. In general, more-complicated prediction models can exploit sensor reading predictability to further reduce communication cost. The prediction module implements prediction models that both the sensor and server agree on in order to enable adaptation. More information about on how sensor's data history values can be used to select which prediction model to use (against a set of predefined ones) can be found in [X.Yu03].

Evaluation

This middleware provides adaptability through the implementation two specific adaptation mechanisms. The first mechanism adapts the filters on sensor-generated data based on the precision settings demanded by the application, while the second mechanism tries to predict the sensors' data values in order to reduce the communication cost of sending messages containing updated values of sensors' readings.

Although both mechanisms can be proven useful for data-driven applications running within a wireless sensor network environment, the middleware does not provide support for other types of QoS adaptation which may be needed for other types of applications, other than those expressing their QoS requirements in terms of their precision settings. For example, real-time applications could require QoS guarantees in terms of bounds of delay and packet loss within the network. No support is provided to address additional non-functional requirements, such as scalability and heterogeneity.

4.3.6 "Issues in Designing Middleware for Wireless Sensor Networks" [Y.Yu03b]

State-of-the-art design of network protocols and applications in wireless sensor networks are usually closely coupled or combined as a monolithic procedure. This indeed limits the reuse and the extensibility of such application and it arises the need for a different approach that separates application semantics from the underlying infrastructure.

The paper firstly highlights amenable features that should belong to a middleware for sensor networks, namely data-centric communication, application knowledge, localized algorithms, lightweightness and the ability of trading the QoS of various applications against each other. Then, it introduce the purposed approach: a cluster-based architecture that includes a cluster layer responsible of creating and maintaining cluster according to different policies (e.g., data accessibility, node capability or network connectivity) and a resource layer in charge of resource allocation within the clusters. Each cluster contains a set of spatially adjacent sensor nodes that cooperate together to provide a specific functionality. Application tasks are distributed over different nodes according to the QoS required by the application and the cluster policy adopted.

Simulation results are provided to evaluate the performance of this approach.

Architecture

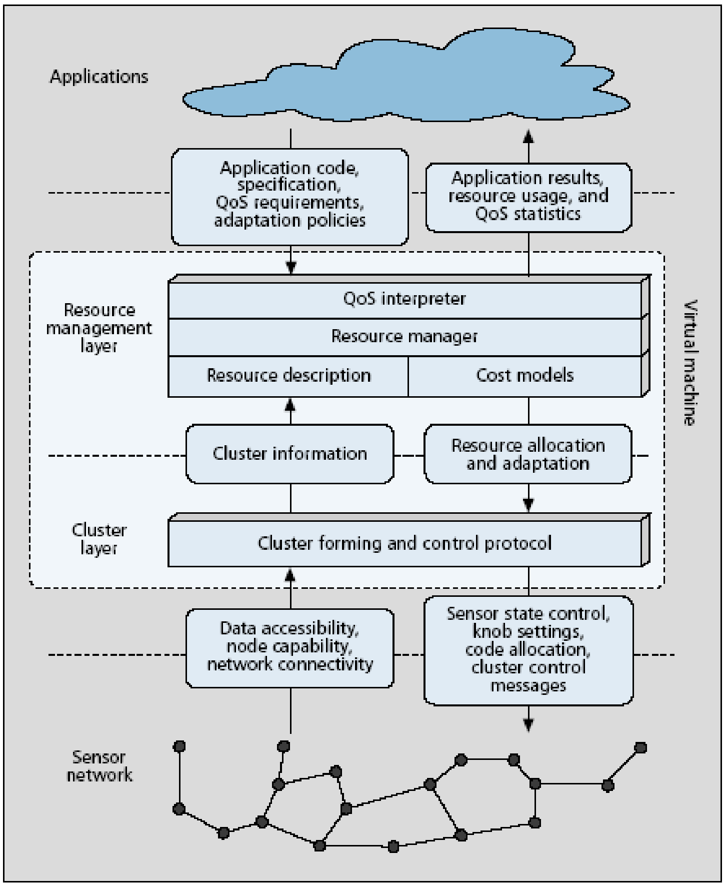

Applications
Application code, specification, QoS requirements, adaptation policies
Application results, resource usage, and QoS statistics
Resource management layer
QoS interpreter
Resource manager
Resource description
Cost models
Virtual machine
Cluster information
Resource allocation and adaptation
Cluster layer
Cluster forming and control protocol
Data accessibility, node capability, network connectivity
Sensor state control, knob settings, code allocation, cluster control messages
Sensor network

Figure 4.22: System architecture

The Cluster Layer is responsible for forming a cluster from a pool of sensor nodes that are around the target phenomena. Different criteria may be adopted to partition the network, e.g., network connectivity, physical proximity and node capability. These demands for on the fly self-configuring distributed clustering mechanism, capable of dynamically determine the membership of nodes as the phenomena move. Authors do not address this issue specifically but suggest some references.

The other component is the Resource Management Layer, which controls the allocation and adaptation of resources. In particular, resource allocation concerns the mechanisms for finding an initial viable solution when the cluster is formed, while resource adaptation commands the runtime behaviour of the cluster.

Inputs from the application include application structure, data/control flow, resource requirements, and performance constraints. It may also include policies for fidelity adaptation and trade-off with other applications. A critical open problem is the development of a formalized general specification for representing the above information.

The allocation phase is driven by a three-phase heuristic, described in a previous paper, is presented. It may be roughly summarized in the following steps:

- Tasks are partitioned into task groups, which are groups of tasks to be assigned on the same sensor nodes. The goal is to minimize the overall execution time of the application. A simple FIFO policy is adopted.
- A greedy policy is used to find an assignment of the tasks group obtained previously onto the actual sensor nodes within the cluster to minimize the maximal energy dissipation among all nodes
- The voltage and the modulation settings of tasks or communication activities are adjusted in an iterative fashion.

Routing mechanisms are not described since the authors assume to leverage off existing network protocols.

Evaluation

The main focus of this paper is on **adaptability**, especially with respect to application needs. All the work is fostered by the need of providing a middleware, able to modify its behaviour according to what specified by the application. Authors projected their cluster layer to deal with changes in the network. As for **fault-tolerance** they cite an existing work in literature where a clustering algorithm, able to face node crashes or disconnection, is presented. Similarly, **energy-awareness** is not addressed directly but it is delegated to the adoption of techniques found in literature.

The main weakness of the paper, with respect to the non-functional dimension we identified, regards its **feasibility** since it stresses mainly the design of the architecture and the heuristic for resource allocation, while it does not spend many words in discussing issues related to energy-consumption or fault-tolerance by redirecting the reader to the included bibliography.

4.3.7 Design and Implementation of a Framework for Programmable and Efficient Sensor Networks (SensorWare)

In this paper the authors argue that the development of a framework based on a mobile agent abstraction will make the sensor networks programmable and open to external users and systems, keeping at the same time the efficiency that distributed proactive algorithms have. This work introduces and describes a new framework, Sensor Ware, which defines, creates, dynamically deploys and supports such agents.

Mobile agents are TCL scripts migrating from node to node. The scripts are made mobile using extend language commands and directives. Its programming paradigm should allows the implementation of almost arbitrary distributed algorithms, since the collaboration structures among sensor nodes are up to the programmer.

The sensing, communication, and signal-processing resources of a node are exposed to the control scripts that orchestrate the dataflow to assemble custom protocol and signal processing stacks. A script can replicate or migrate ("populate") its code and data to other nodes, directly affecting their behaviour.

The wireless sensor network is seen as *a whole, an aggregate* so that a user connected to the network may inject a (possibly distributed task) over the network.

The approach to WSN programmability that is used by SensorWare, as opposed to the traditional distributed-database view, is the active sensor approach. The term indicates a family of frameworks that try to task sensor nodes in a custom fashion, much like active networks task network nodes. The main difference is that active sensors have to react also to events other than network ones, such as sensing events or timeout.

Architecture

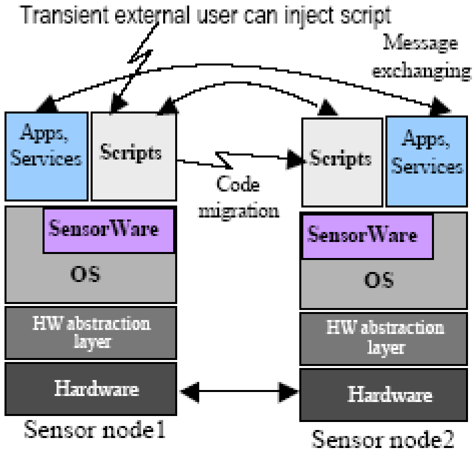


Figure 4.23: SensorWare architecture

The overall architecture is depicted in Figure 4.23. The lower layers are the raw hardware and the abstractions (e.g., device driver) needed to communicate with it. OS is sitting on top of lower layers, providing services and functions for a multithread virtual machine as required by upper layers. SensorWare is integrated into OS and supplies the support for the control scripts. Standard applications coexist with mobile scripts, exploiting the functionalities offered by the OS as well as abstractions provided by SensorWare.

SensorWare provides a set of extension to standard TCL, notably functionalities for moving and relocating code. The general programming paradigm that is adapted is event based. Basically, an event is described and tied with the definition of an event handler. The event handler, according to the current state will do some processing and possibly create some new events and/or alter the current state.

Two different task classes are present: fixed tasks and platform-specific ones. Fixed tasks are always included in every SensorWare implementation and handle system functions such as radio transmitting, tasking allocation and so on. Conversely, platform-specific depends on the specific hardware configuration. Examples falling in these categories are sensor abstractions that enable communication among OS and sensing devices.

Portability is addressed to by creating abstract wrapper functions to clearly separate middleware code from the OS and hardware specific code.

The system has been implemented on the iPAQ 3670 and it was deeply analyzed with respect to memory size, delay and energy consumptions.

Evaluation

The aim of the paper is to supply a flexible yet effective way to enable sensor networks to modify dynamically their behaviour according to the needs of multiple transient users (**openness**). **Heterogeneity** is well addressed since through the architectural design based on wrapper functions, the system is able to run on different hardware and platforms. **Energy-efficiency** is also considered and the authors provide the result of their experiments, aimed to show the low energy consumption. Nevertheless, it is not clear if and how it could be implemented on motes (**feasibility**), since the platform chosen for the implementation, the iPAQ, is provided with completely different hardware. A more interesting test would have been a real implementation on a sensor platform to verify that SensorWare satisfied the high constraint requirements as imposed by sensor networks.

4.3.8 TinyLime: Bridging Mobile and Sensor Networks through Middleware

This work aims to exploit the transiently shared tuple space metaphor as provided by LIME (an extension of LINDA to supply a tuplespace abstraction in a mobile environment) in the context of sensor networks. The resulting model and middleware, called TinyLime makes sensor data available through a tuple space interface, providing a data sharing abstraction among applications and sensors.

In LIME (context) data provided by each mobile unit is transparently shared based on connectivity and data can be accessed proactively (queries) or reactively (reactions/events). In TinyLime the programmer still sees a transiently shared tuple space, which however contains also sensor data. However, considering a mote just like another Lime host would be impractical. Instead, a mote is part of the system only if currently connected through same base station and it is “seen” as if it were another agent on the base station’s host. This clearly implies a different implementation where no tuple space is available on the mote but query operations on the federated tuple space span not only hosts and agents, but also motes. Nevertheless, sensor data in the tuple space is read-only and reactions are extended with a notion of freshness.

The operational settings targeted by TinyLime removes the usual assumptions of a central collection point for sensor data. Instead the sensors are sparsely distributed in an environment, not necessarily able to communicate among themselves, and a set of clients move through space accessing the data of sensors nearby, yielding a system which naturally provides context relevant information to client applications. We further assume the clients are wirelessly networked and share locally accessed data. This scenario is relevant, for example, when relief workers access the information in their work area and share this information with other workers.

Data is cooperatively collected by mobile monitors interconnected through a MANET, which can access only those sensors that are directly available to them. Communication between the mobile monitors is entirely handled through LIME through a new layer that is built entirely on top of it to interact efficiently with the specialized components deployed on the sensors. It does not require multi-hop communication among sensors and places little computation and communication demands on the motes.

TinyLime was entirely developed in nesC on XBOW Mica2 motes and code is available for download at http://www.leet.it/pwp-motes/static/WebSummary.html

Architecture

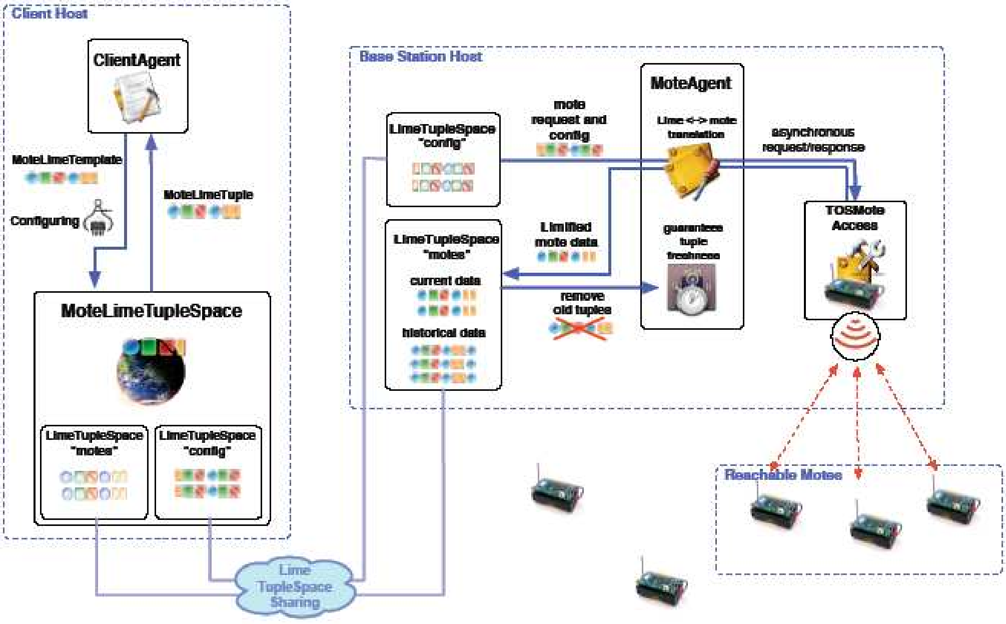


Figure 4.24: TinyLime architecture

In TinyLime's model clients interact with motes through by `MoteLimeTupleSpace`, who refines the normal Lime tuple space interface. Queries are in the form `rd[cur][dest][moteid][freshness](p)` where `p` is a pattern with the following format: `<SensorType,SensedValue, Epoch, Date>.` Reactions work as in Lime, but are extended with the ability to match against inequality and on value ranges. Additional methods are provided to directly perform actions on one mote, or on all those in range (e.g., setBuzzer, setDutyCycle, setSensingTimeout, setRadioPower).

`MoteLimeTupleSpace` provides the illusion of a single tuple space containing sensor data. Inside it, two base Lime shared tuple spaces are used:

- The `config` tuple space is used to communicate requests.
- The `motes` tuplespace stores the actual data.

Request processing "emulates" (asynchronously) a client-server paradigm using the shared tuple space. A read request generates a tuple in the client config tuple space and simultaneously registers a reaction on the motes tuple space. Thanks to sharing, the agent on the base station is able to react and trigger processing on the motes to fetch results.

Sensed data is stored in the motes tuple space, where it triggers the client reaction which delivers the data.

Data are stored in the tuple space only on demand i.e., when a request is issued towards a mote. Subsequent requests are answered by returning the cached data, and do not involve the mote anymore until the data becomes stale, i.e., no longer fresh. Freshness is a parameter configurable on a per-invocation basis. This is required to save power on the motes. On each base station, operation processing as well as management of the sensed data is performed by an instance of `MoteAgent` that installs reactions, processes requests, and purges stale data.

`TOSMoteAccess` translates the high-level requests of `MoteAgent` into messages towards sensors (read, reaction, stop, set). It is highly decoupled and since it does not contain anything specific to

Lime, it can be reused in other projects. Read operations simply involve a request-reply pair but timeout may cause retransmission by the base station

Reactions are performed without storing state on motes (the base station queries the mote once per epoch) because otherwise would occupy resources, and complicate handling of disconnection.

Motes are not always on, to save power. However, power is not an issue for the base station, which therefore can safely broadcast a message repeatedly. TinyLime do not explicitly control when each mote wakes up. In principle several motes may end up competing for the channel.

An algorithm (wakeup scattering) to adjust mote wakeup-times to prevent transmission collisions is provided:

1. At the beginning of the first round, the process is initiated by a special packet sent by one of the motes or an external agent.
2. After receiving this packet, each mote randomly places its wakeup time within the epoch and, when this time arrives, sends a scatter notify packet.
3. During the entire epoch, each mote records the arrival times of the notify packets of the others. Only two such packets are of interests, the one received immediately before the transmission of the mote s own scatter notify, and the one received immediately after.
4. At the end of the epoch, the mote finds the midpoint between these two transmissions, and moves its own wakeup time closer, but not exactly, to this point.
5. The process of detecting the scatter notify packets and moving the wakeup time can be repeated any number of times, iteratively refining distribution. The current implementation of TinyLime works on the Crossbow mote platform (TinyOS) and is written in nesC.

Evaluation

The operational setting as proposed by TinyLime, in contrast with mainstream approaches, does not assume a centralized data collector. This results in a more flexible system, able to deal with changes in network topology, geared towards highly dynamic and mobile systems (**fault tolerance** and **adaptability**). Further, the abstraction provided takes advantage of Lime’s content-based, **transparent** context access, freeing the user from being aware of physical data location, since she can access them simply performing a read on tuple-space. Finally, it keeps into account **power consumption** by exploiting a distributed protocol to wake up motes, so saving battery and increasing network life system. A kind of **heterogeneity** is addressed as well, since the communication takes place among mobile station, each of which may communicate with different sensor, provided with the appropriate network interfaces. Conversely, it shows some drawbacks, due to its early development: the query language is rather simple (though accommodated by the last version of the tuple space engine, LighTS) and, currently it does not provide any multihop access to motes.

4.3.9 EnviroTrack: Towards an Environmental Computing Paradigm for Distributed Sensor Networks

The work reported in this paper is motivated by the increasing importance of distributed sensor networks as a platform for a number of applications such as habitat monitoring intrusion detection, defence, and scientific exploration. These applications perform activities that are often a direct consequence of particular events that take place in the physical environment, and as a consequence these systems aim to track the location of entities in the external environment, yet this task is often addressed in ways that are specific to the application at hand.

Authors think that new tracking-related abstractions should be provided to the programmer of distributed applications. EnviroTrack is in fact a distributed system that implements such abstractions in middleware that runs on sensor devices.

It differentiates itself from traditional localization systems that assume cooperative users who, for example, can wear beaconing devices that interact with location services in the infrastructure for the purposes of localization and tracking; the authors focus instead on situations where no cooperation is assumed from the tracked entity.

Envirotrack main goal is to provide a new abstraction based on context labels and tracking objects. In the architecture context labels are active elements that do not only provide a mechanism for addressing nodes that sense specific environmental conditions, but also can host context-specific computation that tracks the target entity in the environment.

As the tracked entity moves, the identity and location of the sensor nodes in its neighbourhood change, but the tracking object representing it remains the same. The programmer thus interacts with a changing group of sensor nodes through a simple object interface. Simple language support was developed to declare tracking objects.

System Architecture

Envirotrack programming model is shown in the figure below.

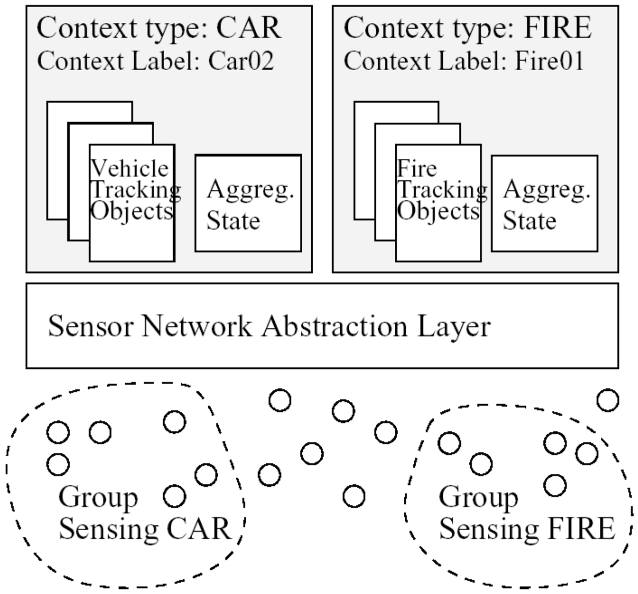


Figure 4.25: Envirotrack programming model

As we can see sensors that sense a given entity in their proximity form a group. This group is dynamic as it changes following the entity moves in the physical world. The network abstraction layer associates each group with a context label which serves as an identifier for the tracked entity. Applications declare a new context type specifying an activation condition called *sense()* which is used by sensors to join and leave the group.

The second part of a context type definition is composed of a given number of context variables that define what is called aggregate state. This state is an aggregation made up of readings taken by the various sensors of the group about the sensed entity. Example of this variable could be the average location for cars or maximum temperature for fires. For each context variable applications define an aggregation function and two constants: a critical mass Ne and a freshness Le. An aggregate variable value is significant only if is computed from at least Ne sensor readings at most Te seconds old.

As shown in Figure 4.25 one or more tracking objects can be attached to each context type constituting the third and last component that applications set with the goal of performing context-specific computation. These tracking objects can access the aggregate state and take local action (using actuators if available) or report some information to a base station. The activation of this computation can be time-triggered, invoked remotely or based on a predicate on the aggregate state.

The architecture of the system is made up of the following elements:

- *EnviroTrack pre-processor*: This pre-processor emits NesC code that initialize structures to track context labels and periodically sense the *sense()* functions to allow entity discovery. It also translates definition of context types, for example replacing aggregate variable references with functions that access the computed state.
- *Group management protocol*: This protocol manages join and leaves of sensors sensing the activation condition. In particular it ensures that there is always at least one leader for the group using heartbeat from the leader and also flood the group surroundings to avoid creating a new context label as the entity activates new sensors.
- *Aggregate state computation*: The leader of a group collects readings from sensors of the group, aggregating values from the last L seconds and marking the result as valid iff it is a result of at least N readings. Group members send their data to the leader every P seconds where P=L-d and d is the expected transmission delay.
- *Object naming and directory services*: Context types are hashed to an (x,y) location which serves as a directory for that type. Whenever new context labels are created they register on the directory service their position to allow being found by other objects. As entities move in the field they leave temporary forwarding pointers behind and occasionally update their position on the directory service.
- *Communication and transport services*: Tracking objects can communicate between each other using context labels as logical addresses which are managed by the group leader of that context sensor group. A remote method invocation engages the transport protocol for communication between leaders of the source and destination objects leader. This leader uses a protocol to identify the location of the remote context using the already mentioned directory lookup and communicates the message to the remote context leader.

Evaluation

Envirotrack albeit limited to tracking functionality, thus not addressing the querying and monitoring (wide) application area, it realizes a sound and readily available piece of software.

The design seems to be thought of with ***energy efficiency*** in mind, for example all communication to track the entities are local and energy consumptions seems to be related to maximum speed of the tracked entity. There is possibly some room for improvement as for example in dense networks the current scheme does not provide a method to put unnecessary nodes into sleep.

Long range communication is used is for remote procedure call and data reporting. These communications are based on a sort of directory service built using DHT.

The system exhibits a good degree of *fault tolerance*: in particular the algorithms are designed to resist message loss, leader failures, and for example the aggregate state is loosely synchronized to account for communication failures.

Also the *feasibility* of the system is demonstrated by the available implementation of the whole system.

A possible limitation can be spotted in the area of *openness* and *heterogeneity*; in fact the system is implemented on motes at NesC level. This unfortunately prevents the runtime definition and deployment of new context types, limiting the run-time flexibility of the implementation.

4.3.10 Sensor Information Networking Architecture and Applications (SINA)

Issues concerning how information collected by and stored within a sensor network could be queried and accessed and how concurrent sensing tasks could be executed internally and programmed by external users are of particular importance. In this article the authors describe a sensor information networking architecture, called SINA, which facilitates querying, monitoring, and tasking of sensor networks.

The sensor network is modelled as a collection of massively distributed objects. SINA plays the role of middleware, allowing sensor applications to issue queries and command tasks into, collect replies and results from, and monitor changes within the networks.

In contrast to conventional distributed databases in which information is distributed across several sites, the number of sites in a sensor network equals the number of sensors, and the information collected by each sensor becomes an inherent part (or attributes) of that node.

To support energy-efficient and scalable operations, sensor nodes are autonomously clustered. Furthermore, the data-centric nature of sensor information makes it more effectively accessible via an attribute-based naming approach instead of explicit addresses.

Architecture

SINA architecture consists of the following functional components:

- *Hierarchical Clustering*: Sensors autonomously forms clusters according to power level and proximity. This clustering scheme can be applied recursively to form a hierarchy of clusters. Cluster heads are responsible for filtering fusion and aggregation of information. Appropriate measures must be taken in order to reorganize the cluster in the case a leader runs low on battery power or fails unexpectedly.
- *Attribute based naming*: Since sensors are deployed in high numbers it is often not useful to address a single node in the network by its identifier. It's a common for application to address sensors using attributes to specify the target of queries or tasks. For example common operations might be regarding which area(s) has temperature higher than 100°F or what is the average temperature in the southeast quadrant, rather than the temperature at sensor ID#101. SINA offers support for attribute-based naming
- *Location Awareness*: Due to the fact that sensor nodes operate in physical environments, knowledge about their own physical locations is crucial. This information can be acquired directly by GPS receivers for absolute positioning or using optical trackers or other self-localization mechanisms for computing offset wrt GPS equipped hardware.
- *The SINA programming model*: In SINA, a sensor network is viewed as a collection of datasheets; each datasheet contains a collection of attributes of each sensor node. Each attribute is referred to as a cell, and the collection of datasheets of the network present the abstraction of an associative spreadsheet, where cells are referred to via attribute-based names. Initially the datasheet on each node is empty. A node creates a new cell when it receives a request from other nodes, the value in a cell can be obtained in one of the following ways:
  - referring directly from one or more cells
  - invoking a system-defined function
  - aggregating values from other sensors' spreadsheets

SINA offers two choices with respect to the language available to program the network: the first is a declarative language which is based on SQL. Examples of the language shown in figure are used to define new cells in the spreadsheet. As an alternative SIENA support SQTL (Sensor Querying and Tasking Language) which is a light OO procedural language with some kind of support to location awareness for event handling to facilitate asynchronous programming. Both SQL and SQTL code are handled by an interpreter albeit they are not presented in the papers.

SQL and SQTL programs are shipped in the network by means of an XML wrapper containing forwarding information (such as ALL_NODES, NEIGHBORS, etc.) and predicates identifying target sensors by means of condition on attributes (e.g., [temperature > 30 and area= N-W]). It is runtime environment's job to discard non relevant messages.

Authors also discuss three general approaches to avoid the response implosion problem affecting information gathering in sensor networks:

- reduction of nodes involved in a query for dense networks
- random delay of responses to avoid collisions at the mac layer
- in-network aggregation of information which also reduces network traffic

Evaluation

SINA offers a composite middleware platform that is intended to provide applications with an infrastructure for sensor networks centred on associative spreadsheets. It also defines a new programming language SQTL with its execution environment.

The main concerns about the whole system are its feasibility and performance as the execution environment is not detailed at the algorithm level and no actual implementation of the system is available. As an example the use of associative broadcast raises concerns about the energy efficiency of such a scheme that are not addressed in the papers.

SINA main achievements are in the area of openness and adaptability, as SQTL code can be shipped around and can specify coordination of the sensor at a fine granularity to perform a given goal.

5 Critical Analysis and Conclusions

Chapter 4 of this document presented a number of middleware systems targeting networked embedded systems. In this section, we want to summarize our findings. Table 5.1 indicates, for each of the middleware presented, a concise indication of its main purpose and the list of the functional and non functional requirements addressed by the middleware. The table also reports on the type of system the middleware is currently targeting.

Middleware	Purpose	Functional requirements addressed	Non-functional requirements addressed	Type of system targeted
GAIA	Context-aware environments	Event Notification, Mobility and Location Awareness, Addressing, Service Discovery, Code Updater	Openness	Mobile Hybrid
ExORB	Runtime reconfigurability	Mobility and Location Awareness, Code Updater	Heterogeneity Openness Adaptability	Mobile Phones
WSAMI	Interoperability in pervasive environments	Mobility and Location Awareness, Service Discovery	Heterogeneity Security	Mobile Hybrid (not truly ad hoc)
CORTEX	Reconfigurable for different platforms. Asynchronous communication based	Mobility and Location Awareness	Adaptability	Mobile Hybrid (not truly ad hoc)
AURA	Context-aware through task migration	Event Notification. Mobility and Location Awareness, Service Discovery	Adaptability Security Adaptability	Mobile Hybrid (not truly ad hoc)
Oxygen	User driven adaptation framework	Mobility and Location Awareness, Service Discovery, Code Updater	Adaptability Openness Security	Mobile Hybrid
CARISMA	Reflective approach for context awareness	Mobility and Location Awareness	Adaptability Openness	Mobile Ad Hoc (or Hybrid)
LIME	Data sharing in mobile environments	Event Notification, Mobility and Location Awareness	Adaptability Openness Failure Handling	Mobile Ad Hoc (or Hybrid)
REDS	Content-based pub/sub on a dynamic network topology	Event-Notification, Mobility and Location Awareness	Fault-tolerance Scalability	Mobile Ad Hoc (or Hybrid)
SATIN	Using logical mobility for adaptation	Mobility and Location Awareness, Code Updater	Heterogeneity Adaptability	Mobile Ad Hoc (or Hybrid)

STEAM	Publish subscribe communication	Event Notification Mobility and Location Awareness		Mobile Ad Hoc
ZEN	Real time support	Real time	Heterogeneity	Fixed Network Embedded
MiLAN	Runtime adaptation to application requirements	Cross-layer communication	Adaptability Openness Performance (energy-efficiency)	Sensor Networks
Impala	Runtime adaptation to application requirements, dynamic reprogramming	Cross-layer communication, Code Updater	Adaptability Openness Performance (energy-efficiency)	Sensor Networks
AutoSec	Dynamic service brokering in a distributed environment	Cross-layer communication, Service Discovery	Adaptability Performance	Sensor Networks
DSWare	Data-base like abstraction	Event notification, Distributed storage and lookup	Adaptability Performance	Sensor Networks
Adaptive Middleware for Distributed Sensor Environments	Runtime adaptation to application requirements	Cross-layer communication	Adaptability Performance (energy-efficiency)	Sensor Networks
EnviroTrack	Entity tracking in sensor network	Multi-hop routing, Time Synchronization, Clustering	Performance (energy-efficiency) Failure handling Feasibility	Sensor Networks
Sina	Data gathering and tasking for sensor networks	Distributed Query-like interface	Openness Adaptability	Sensor Networks
[Y.Yu03b]	Application task allocation	Distributed Task Scheduler, Clustering	Adaptability Fault-tolerance Performance (energy-efficiency)	Sensor Networks
SensorWare	Mobile Agent Support in sensor networks	Code Updater	Openness Heterogeneity Performance (energy-efficiency)	Sensor Networks

TinyLime	Data sharing among applications and sensors	Wake-up Coordination, Distributed Tuple Space	Fault tolerance Adaptability Transparency Heterogeneity Performance (energy-efficiency)	Sensor Networks

Table 5.1: Assessment of overviewed middleware platforms

As we can see from Table 5.1, most of the effort has been put on addressing the requirement of adaptability in networked embedded systems. These efforts have yielded encouraging results and a number of well recognized approaches have been developed including service discovery protocols [SLP], [Avancha02] as used in MiLAN and run-time adaptation protocols as used in Impala and AutoSeC. This effort is explained due to the dynamic nature of networked embedded systems: wireless networks with changing topologies, devices that constantly join/leave the formed networks, devices that run out of power, new applications and users continuously updating their QoS requirements are some examples of factors that potentially lead to a high degree of dynamism in a networked embedded environment.

However, although a certain form of adaptability is offered by most of the systems included in our survey, the adaptation mechanism of the middleware remains static during its lifetime. This means that it is not possible to use an alternative adaptation mechanism when different adaptive behavior is required to be performed by the middleware. For example, it is possible that during the lifetime of a networked embedded system, a new application is introduced demanding a different type of adaptation than the one offered by the middleware. For this purpose, a new adaptation mechanism should be selected and configured within the middleware in order to provide the new type of adaptation required by the application. The problem of selecting and configuring the new adaptation mechanism becomes more complex considering the demand for autonomous operation of the middleware. Since it not possible to assume human intervention for management of the middleware operating within a networked embedded system, intelligent mechanisms such as expert systems should be studied to see if they can operate within the middleware, such as the middleware itself has the capabilities to dynamically update its adaptation mechanisms.

Another problem that we can see observing Table II is that although most of the middleware have addressed the requirement for adaptability, no single middleware exists addressing all the non-functional requirements of our list. This is due to the assumptions about the application or the types of embedded devices that the middleware designers have made. For example, some of the work on sensor networks assumes the existence of a single type of sensors embedded in the environment, thus not addressing the requirement for heterogeneity. However, in the most general case of a networked embedded system consisting of many types of devices other than single-type sensors, it is important that the requirement for heterogeneity is addressed by the middleware.

Another remark is that the requirement of feasibility has not been adequately addressed by existing middleware systems. However, considering the resources limitations (in hardware and in network resources) that are potentially present in a networked embedded system, new mechanisms should be implemented to ensure that the functions or services that the middleware offers to the application are feasible to be performed in a given system/network instance.

Based on the lessons we learned during this survey in RUNES we will create a middleware solution for networked embedded system, which fulfils the identified requirements and solves the problems we have found in existing approaches. The result of this work will be presented in the following deliverables of WP5.

References

[Akyildiz02] I.F. Akyildiz et al., "A Survey on Sensor Networks," IEEE Communications Magazine, August 2002, pp. 102-114.

[Apostolopoulos98] Apostolopoulos, G., Guerin, R., Kamat, S., Tripathi, S.K., "Quality of Service based Routing: A Performance Perspective". In proceedings of ACM SIGCOMM, 1998.

[Avancha02] Avancha, S., Joshi, A., Finin, T., "Enhanced Service Discovery in Bluetooth", IEEE Comp., Volume 35, Issue No. 6, June 2002, pp. 96 – 99.

[Bharathidasan] A. Bharathidasan and V. A. S. Ponduru. "Sensor Networks: An Overview". Available from http://wwwcsif.cs.ucdavis.edu/~bharathi/

[Blair et al, 01] Blair, G.S., Coulson, G., Andersen, A., Blair, L., Clarke, M., Costa, F., Duran-Limon, H., Fitzpatrick, T., Johnston, L., Moreira, R., Parlavantzas, N., Saikoski, K., "The Design and Implementation of OpenORB v2", To appear in IEEE DS Online, Special Issue on Reflective Middleware, 2001.

[Bonnet00] Bonnet, P., Gehrke, J., Seshadri, P.,"Querying the Physical World", IEEE Pers. Commun., Volume 7, Issue No. 10 – 15, October 2000.

[Boulis03] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava, "Design and Implementation of a Framework for Efficient and Programmable Sensor Networks", MobiSys 2003

[Brumit00] B. Brumit, B. Meyers, J. Krumm, A. Kern, S. Shafer: *"EasyLiving: Technologies for Intelligent Environments"*, Handheld and Ubiquitous Computing, September 2000.

[Capra03] L. Capra, W. Emmerich and C. Mascolo (2003). CARISMA: Context-Aware Reflective mIddleware System for Mobile Applications. IEEE Transactions on Software Engineering, 29(10):929-945.

[Carzaniga01] A. Carzaniga, D. Rosenblum and A. Wolf, "Design and Evaluation of a Wide-Area Event Notification Service", ACM Transactions on Computer Systems, 19(3), pp. 332-383, 2001.

[Casimiro] António Casimiro and Paulo Veríssimo. "Using the Timely Computing Base for Dependable QoS Adaptation". Proceedings of the 20th IEEE Symposium on Reliable Distributed Systems, New Orleans, USA, October 2001.

[CORBA] Object Management Group, The Common Object Request Broker: Architecture and Specification, Revision 2.0, July 1995.

[CORTEX] CORTEX Project: CO-operating Real-time senTient objects: architecture and EXperimental evaluation - http://cortex.di.fc.ul.pt/index.htm

[Costa04] Paolo Costa, Matteo Migliavacca, Gian Pietro Picco, and Gianpaolo Cugola. Epidemic Algorithms for Reliable Content-Based Publish-Subscribe: An Evaluation. Proceedings of the 24th

International Conference on Distributed Computing Systems (ICDCS04), pages 552-561, March 2004, T.H. Lai and K. Okada eds., IEEE Computer Society Press.

[Coulouris] Distributed Systems: Concepts and Design', second edition, George Coulouris, Jean Dollimore, and Tim Kindberg, Addison-Wesley 1994.

[Cugola01] G. Cugola, E. Di Nitto and A. Fuggetta, “The JEDI event-based infrastructure and its application to the development of the OPSS WFMS”, IEEE Transactions on Software Engineering, 9(27), pp. 827-850, September 2001.

[Cugola04] Gianpaolo Cugola, Davide Frey, Amy L. Murphy, and Gian Pietro Picco. Minimizing the Reconfiguration Overhead in Content-Based Publish-Subscribe. Proceedings of the 19th ACM Symposium on Applied Computing (SAC04), pages 1134-1140, March 2004, A. Omicini et al. eds., ACM Press.

[Curino05] Carlo Curino, Matteo Giani, Marco Giorgetta, Alessandro Giusti, Amy L. Murphy, and Gian Pietro Picco. TinyLIME: Bridging Mobile and Sensor Networks through Middleware. To appear in Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom 2005), Kauai Island (Hawaii), March 8-12, 2005 **.**

[DCOM] Microsoft Corporation, Microsoft COM technologies –DCOM. http://www.microsoft.com/dcom.asp

[EmbeddedJava] Sun Microsystems, Embedded Application Environment. http://java.sun.com/products/embeddedjava

[Emmerich00] W. Emmerich, “Software Engineering and Middleware: a roadmap”. In Proceedings of the Conference on the Future of Software Engineering, pp. 117-129, 2000.

[Esler99] M. Esler, J. Hightower, T. Anderson, G. Borriello: *"Next Century Challenges: Data-Centric Networking for Invisible Computing: The Portolano Project at the University of Washington"*, MobiCom '99

[FRIENDS] J. C. Fabre and T. Perennou, “A Metaobject architecture for fault-tolerant distributed systems: The FRIENDS approach”, IEEE Transactions on Computers, vol. 47. no. 1, pp. 78-95. 1998.

[Gaia] Gaia Project: Active Spaces for Ubiquitous computing. http://gaia.cs.uiuc.edu/index.html

[Garlan02] D. Garlan, D. P. Siewiorek, A. Smailagic, P. Steenkiste: "*Aura: Toward Distraction-Free Pervasive Computing"*, IEEE Pervasive Computing, 2002

[Gelernter 1985] D. Gelernter. Generative Communication in Linda. ACM Computing Surveys, 7(1):80-112, January 1985.

[Grimm00] R. Grimm, T. Anderson, B. Bershad, D. Wetherall: *"A system architecture for pervasive computing"*, appeared in the Proceedings of the 9th ACM SIGOPS European Workshop, pages 177-182, Kolding, Denmark, September 2000.

[Grimm01] R. Grimm, J. Davis, B. Hendrickson, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, D. Wetherall: *"Systems directions for pervasive computing"*, appeared in the Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), pages 147-151, Elmau, Germany, May 2001

[Grimm03] R. Grimm, J. Davis, E. Lemar, A. MacBeth, S. Swanson, T. Anderson, B. Bershad, G. Borriello, S. Gribble, D. Wetherall: *"Programming for pervasive computing environments"*, submitted for publication, 2003

[Han01] Han, Q., Venkatasubramanian, N.,"AutoSeC: An Integrated Middleware Framework for Dynamic Service Brokering", IEEE Distributed Systems Online, Volume 2, Issue No. 7, 2001.

[Heinzelman04] W. Heinzelman, A. Murphy, H. Carvalho and M. Perillo, "Middleware to support sensor network applications". IEEE Network Magazine, 18(1):6--14, 2004.

[Java-RMI] Java Soft, Java Remote Invocation specification, revision 1.5, JDK1.2 edition, October 1998.

[Judd03] G. Judd, P. Steenkiste: *"Providing Contextual Information to Pervasive Computing Applications"*, PerCom 2003, Dallas, March 23-25, 2003.

[ICE] ICE: http://www.zeroc.com

[IRL] R. Baldoni et al. "Active Software Replication through a Three-tier Approach". In Proceedings of the 22th IEEE International Symposium on Reliable Distributed Systems (SRDS02), pp. 109-118, Osaca, Japan, October 2002.

[Issarny] Valerie Issarny, Daniele Sacchetti, Ferda Tartanoglu, Francoise Sailhan, Rafik Chibout, Nicole Levy, Angel Talamona. "Developing Ambient Intelligence Systems: A Solution based on Web Services". In Journal of Automated Software Engineering. 2004.

[Jaikaeo00] Jaikaeo, C., Srisathapornphat, C., Shen, C.-C.,"Querying and Tasking in Sensor Networks", SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V), Orlando, FL, April 2000.

[Li03] Li, S., Son, Stankovic, J.,"Event Detection Services Using Data Service Middleware in Distributed Sensor Networks", In Proceedings of the 2nd International Workshop for Information Processing in Sensor Networks, April 2003.

[Liu03] Liu, T., Martonosi, M., "Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems", ACM SIGPLAN Symposium, Principles and Practise of Parallel Programming, June 2003.

[Mascolo02] Cecilia Mascolo, Licia Capra , and Wolfgang Emmerich. "Middleware for Mobile Computing (A Survey)". In Advanced Lectures in Networking. Editors E. Gregori, G. Anastasi, S. Basagni. Springer. LNCS 2497. 2002.

[Meier02] R. Meier and V. Cahill, "STEAM: Event-Based Middleware for Wireless Ad Hoc Networks", Proceedings of the International Workshop on Distributed Event-Based Systems (ICDCS/DEBS'02), pp. 639-644, Vienna, Austria, 2002.

[Murphy00] A.L. Murphy and G.P. Picco and G.-C. Roman. Lime: A Middleware for Physical and Logical Mobility. Proc. of the 21st Int. Conf. on Distributed Computing Systems (ICDCS-21). F. Golshani and P. Dasgupta and W. Zhao eds. May 2001, pp. 524-533.

[Murphy04] Amy L. Murphy and Gian Pietro Picco. Using Coordination Middleware for Location-Aware Computing: A Lime Case Study. Proceedings of the 6th International Conference on Coordination Models and Languages (COORD04), pages 263-278, R. De Nicola, G. Ferrari, and G. Meredith eds., Springer LNCS 2949, February 2004.

[Oxygen] MIT Oxygen project (http://oxygen.lcs.mit.edu/)

[OxygenSoftwareTech] Oxygen Software Technologies, http://oxygen.lcs.mit.edu/Software.html

[OZONE] OZONE Project New Technologies and Services for Emerging Nomadic Societies http://www.extra.research.philips.com/euprojects/ozone/

[Picco99] G.P. Picco and A.L. Murphy and G.-C. Roman. Lime: Linda Meets Mobility. Proc. of the 21st Int. Conf. on Software Engineering. D. Garlan ed. May 1999, pp. 368-377.

[Picco02] Gian Pietro Picco and Marco L. Buschini. Exploiting Transiently Shared Tuple Spaces for Location Transparent Code Mobility. Proceedings of the 5th International Conference on Coordination Models and Languages, pages 258-273, April 2002, F.Arbab and C. Talcott eds., Springer LNCS 2315.

[Picco03] Gian Pietro Picco, Gianpaolo Cugola, and Amy L. Murphy. Efficient Content-Based Event Dispatching in Presence of Topological Reconfigurations. Proceedings of the 23RD International Conference on Distributed Computing Systems (ICDCS03), P. McKinley and S. Shatz eds. May 2003, ACM Press, pages 234—243.

[Portolano] *"Portolano/Workscape: Charting the new territory of invisible computing for knowledge work"*, Online Documentation, http://portolano.cs.washington.edu/proposal/

[QuO] J.A. Zinky et al. “Architectural support for quality of service for CORBA objects”, Theory and Practice of Object Systems, vol 3, no. 1, 1997.

[Römer02] Kay Römer, Oliver Kasten and Friedmann Mattern. “Middleware challenges for wireless sensor networks”. ACM SIGMOBILE Mobile Computing and Communications Review, Volume 6, Issue 4, pp. 59-61, October 2002.

[Sadjadi] S.M. Sadjadi, “A Survey of Adaptive Middleware, Michigan State University.

[Schmidt02] D.C.Schmidt, “Middleware for real-time and embedded systems”, Communications of the ACM, vol. 45, June 2002.

[Sivaharan] Sivaharan, T., Blair, G.S., Friday, A., Wu, M., Duran-Limon, H., Okanda, P., Sørensen, C.F., "Cooperating Sentient Vehicles for Next Generation Automobiles", ACM MobiSys 2004 workshop on Applications of Mobile Embedded Systems (WAMES 2004), Boston MA, June 2004

[SLP] Service Location Protocol (SLP), http://ietf.org/html.charters/svrloc-charter.html

[Sousa02] Joao Pedro Sousa, David Garlan: *"Aura: an Architectural Framework for User Mobility in Ubiquitous Computing Environments"*, 3rd Working IEEE/IFIP Conference on Software Architecture, Montreal, 2002.

[Sun 98] Sun Microsystems. Java Distributed Event Specification. Sun Microsystems, Inc., July 1998. http://www.javasoft.com/products/javaspaces/specs.

[Tilak02] Sameer Tilak, Nael B. Abu-Ghazaleh and Wendi Heinzelman. “A taxonomy of Wireless Micro-Sensor Network Models”. ACM Mobile Computing and Communications Review (MC2R), Volume 6, Number 2, April 2002.

[Tubaishat03] M. Tubaishat, S. Madria. “Sensor Networks : An Overview”. IEEE Potentials, April/May 2003.

[Veríssimo02] P. Veríssimo, V. Cahill, A. Casimiro, K. Cheverst, A. Friday and J. Kaiser. "CORTEX: Towards Supporting Autonomous and Cooperating Sentient Entities". Proceedings of European Wireless 2002, Florence, Italy, February 2002.

[Wang00] Z. Wang, D.Garlan: *"Task-Driven Computing"*, Carnegie Mellon University Technical Report CMU-CS-00-154, http://reports-archive.adm.cs.cmu.edu/cs2000.html, May 2000.

[Wendi04] Wendi, B., Heinzelman, Amy, L., Murphy, Hervaldo, S., Carvalho, Mark A., Perillo, “Middleware to Support Sensor Network Applications”, Network IEEE, Volume 18, Issue No. 1, Jan/Feb 2004, pp. 6 – 14.

[Wolenetz04] M. Wolenetz, R. Kumar, J. Shin, and U. Ramachandran. Middleware Guidelines for Future Sensor Networks. In Proceedings of Fist Annual Workshop on Broadband Advanced Sensor Networks (Broadnets 2004), San Hose, CA, USA, 2004.

[WSAMI] WSAMI: A Middleware Infrastructure for Ambient Intelligence based on Web Services - http://www-rocq.inria.fr/arles/work/wsami.html

[Wu] Maomao Wu, Adrian Friday, Gordon Blair, Thirunavukkarasu Sivaharan, Paul Okanda, Hector Duran Limon, Carl-Fredrik Sørensen, Gregory Biegel and René Meier "Novel Component Middleware for Building Dependable Sentient Computing Applications.", Workshop on Component-oriented approaches to context-aware systems, Oslo, Norway, June 2004. Affiliated with 18th European Conference on Object-Oriented Programming (ECOOP 2004).

[X.Yu03] Xingbo Yu, Koushik Niyogi, Sharad Mehrotra, and Nalini Venkatasubramanian. “Adaptive Middleware for Distributed Sensor Environments”. IEEE Distributed Systems Online, http://dsonline.computer.org, May 2003.

[Y.Yu03a] Y. Yu and V. Prasanna. “Energy-balanced task allocation for collaborative processing in networked embedded systems”. In Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tools for embedded systems, San Diego, California, USA, pp. 265 – 274, 2003.

[Y.Yu03b] Y. Yu, B. Krishnamachari and V. Prasanna. "Issues in Designing Middleware for Wireless Sensor Networks". IEEE Network Magazine, 2003.

[Zachariadis04] S. Zachariadis, C. Mascolo and W. Emmerich (2004). SATIN: A Component Model for Mobile Self-Organisation. In R. Meersman and Z. Tari et al. (eds), On the Move to Meaningful Internet Systems 2004: Proc. of CoopIS, DOA and ODBASE, Agia Napa, Cyprus. Lecture Notes in Computer Science. Vol. 3291. pp. 1303-1321.

[ZEN] ZEN**:** Highly Configurable Real-time Object Request Broker, http://zen.ece.uci.edu/zen/

A. Appendix - Security and location services

A.1. Cryptographically Generated Addresses

Middleware used on embedded systems focuses on providing a homogeneous set of functions to the upper layers on different distributed nodes, which helps to abstract the heterogeneity of the node's underlying architecture. The benefit of a middleware therefore is to limit the need of adapting upper layers to the specific environment found on a certain node. In order to do so information needs to be exchanged between the instantiations of the middleware on these different nodes.

Depending on the functionality of the middleware this information exchange needs to happen in a more or less secured manner. A first step of such a security can be the authentication of the exchanged information, that is, the receiving instantiation of a middleware can be really sure that the information is coming form a certain sending instantiation of the middleware. In a second step the exchanged information additionally can be encrypted in order to provide confidentiality.

A.1.1. Drawbacks of public / private key authentication

The idea behind Cryptographically Generated Addresses (CGAs) is to have a lightweight mechanism for authenticating IPv6 packets.

Doing such an authentication today quite often is based on the usage of public/private key mechanisms. For this purpose the sender of an IPv6 packet needs to generate a public/ private key pair first, and to use the private key in order to sign the IPv6 packet. In order to verify this signature the receiver of the IPv6 packets has to know the public key of the sender. For this purpose the sender could easily send its public key to the receiver, however, if an attacker replaces this public key on the fly by its own, and sends a faked IPv6 packet signed by its private key, the attacker can hijack a complete session. Therefore it is absolutely important, that the receiver can be really sure he received the correct public key from the sender. This guarantee usually is provided by certificates, that is, a certificate authority trusted by the sender and the receiver will issue a certificate for the sender which certifies, that a certain public key really belongs to the sender.

Deploying such certificate authorities doesn't come without costs. For example every node involved in such a secure communication needs to get into contact with the certificate authority to claim a certificate for its own source address. This means that all communication partners have to trust the same authority providing the certificate authority. Then the communication partners have to exchange their certificates at the beginning of communications. Lastly the nodes have to verify the validity of the certificates and use then the public key contained within them for verifying the messages.

Looking on embedded systems the use of such Public Key Infrastructures (PKIs) certainly has some drawbacks:

- Embedded systems are required to contact a certificate authority first in order to apply for a certificate for the own IPv6 address. Sensor nodes often will be installed plug & play like, that is, any additional configuration effort such as generating certificates may not be possible.
- Transmission of large size certificates could be problematic on narrowband links, as they are often used to connect sensor node.
- Validating the certificates and using the public key in order to verify the received IPv6 packets may consume too many resources in order to be doable on a sensor node.

CGAs provides an alternative mechanism for authenticating IPv6 packets which relaxes some of the drawbacks described above.

A.1.2. Overview of CGAs

CGAs are IPv6 addresses, which allow for a secure association of an IPv6 address, the CGA, with a public key. While this kind of association otherwise is mainly done using certificates, and therefore requires the deployment of Public Key Infrastructures (PKIs), the CGA approach doesn't require any infrastructure at all.

In principle CGAs are generated like all IPv6 addresses by concatenating a 64 bit long subnet prefix with a 64 bit long identifier. However, in CGAs the identifier additionally reflects the public key belonging to the CGA, more precisely, the identifier is a hash value formed from a CGA parameter set, including among others the public key. Knowing these CGA parameters, any receiver of IPv6 packets with a CGA as source address can re-calculate the hash value, and verify if it matches the one contained in the 64 bit identifier of the packet's source address. Figure A.1 provides an overview of the structure of CGAs.

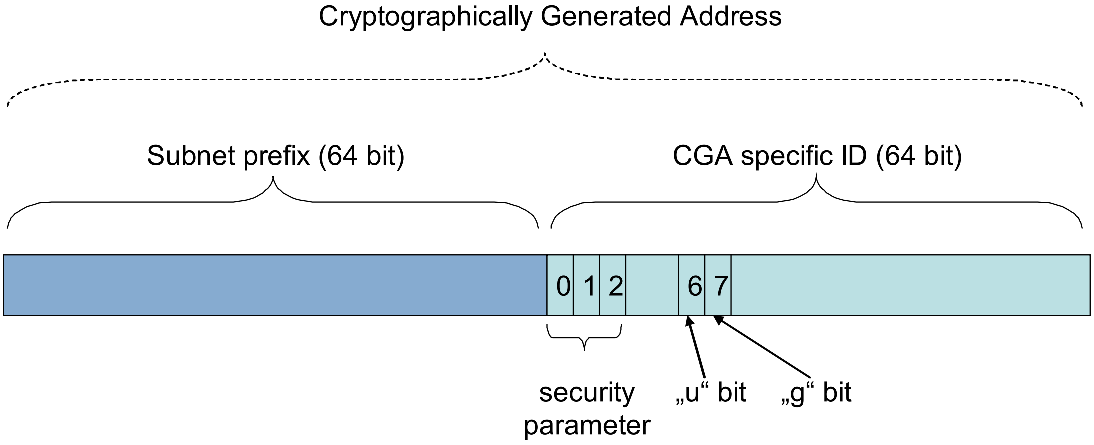


Figure A.1: Structure of CGAs

The CGA parameters mentioned above, which are used for calculating the CGA, consist of the following:

- a 16 octet long modifier, which can be chosen arbitrarily,
- a 8 octet long subnet prefix, which is equal to the subnet prefix of the CGA itself,
- a 1 octet long collision count, as well as
- the public key itself, which can have a variable length.

In order to allow the receiver to verify a CGA, it needs to have the CGA parameters as well as the CGA itself. The latter one is implicitly provided to the receiver in case it is used as source address in IPv6 packets. In principal there can be many ways for exchanging CGA parameter, the IETF send WG for example specified one alternative used for securing the neighbour discovery process.

After a successful verification, the receiver can securely assign a certain public key to an IPv6 address, information, which is usually provided by certificates. With this information the owner of a CGA can use its private key in order to sign messages, knowing, that the receiver will be able to associate the appropriate public key and use this for the verification of the message signature.

A.1.3. Use of CGAs for embedded systems

CGAs above certainly cannot only be used for authenticating middleware traffic exchanged between different embedded systems; CGAs can be used more generic in order to authenticate any information being exchange within IPv6 packets.

CGAs solve some of the issues of public / private key authentication in the area of embedded systems. For example it is no longer required to deploy PKIs and to first contact a certificate authority in order to apply for a certificate. Furthermore certificates need no longer to be exchanged over narrowband links. Instead of this the CGA parameter set needs to be provided to a receiver of CGA packets. However, currently this exchange is currently only defined in the IETF SEND WG for the usage of CGAs for securing IPv6 neighbour discovery messages. Therefore a new exchange mechanism would need to be defined anyway for a more generic usage in the area of embedded systems, and this exchange could be defined in a less resource consuming way than certificates are exchanged. Also due to the lack of certificate there is no longer a need for their validation. However, still the verification of the signature of an IPv6 packet is required, which could still be quite resource consuming. Therefore this process, as well as the also resource consuming process of generating CGAs, may be delegated in some scenario to a more powerful node, like a gateway connecting a cloud of sensor to the backbone network.

Finally it should be noted, that a node using a CGA as source IPv6 address will face some privacy issues. These issues may be caused by the usage of a fixed CGA identifier in IPv6 source addresses, as well as by the transmission of the public key within the CGA parameter set.

The issue with the fixed CGA identifier can be addressed by computing several CGAs for the same public key, and switch between their usages according to the IPv6 privacy extensions specified in RFC 3041. Generating more CGAs for the same public key can be achieved by varying the modifier part of the CGA parameters. As the computation of CGAs could become computational expensive, one solution could be a pre-calculation of CGAs on a more powerful node.

The issue with the fixed public key transmitted within the CGA parameter set is more difficult. As the public key will be visible along the whole path CGA parameters are exchanged, the node owning the CGA will be traceable in this area.. The only chance to avoid this would be the change of the public key itself. However, if CGAs are used only in local environments, such as for securing neighbour discover, one may not be concerned with privacy as tracing a node could here be possible also by other means like following link layer information. Also using certificates instead of CGAs would cause the same privacy concerns.

A.2. Host Identity Protocol

As already mentioned above, middleware usually has several instantiations distributed over several nodes. As all these instantiations need to communicate with each other, each of these instantiations has to be uniquely identifiable. There identifiers will then be used in order to set up information exchange between these different instantiations.

A.2.1. Drawbacks with current naming scheme

Currently in the Internet two main name spaces are used, the Internet Protocol with its IPv4 or IPv6 addresses, and the Domain Name Service (DNS). The IP name space reflects the current point of attachment of a node to the Internet routing infrastructure, that is, an IP address has the functionality of a locator. Contrary the DNS name space assigns a globally unique name to each node, that is, a DNS name has the functionality of an identifier.

The IP address has been originally designed with the following properties:

1. Non-mutable, that is, an address used for sending a packet will be the address when receiving the packet,
2. Fixed, that is, an IP address of a node doesn't change during the time the node has a communication established with another node,
3. Reversible, that is, a packet can always be sent back to the sender by switching the source and destination address fields of the packet, and
4. Omniscient, that is, a node knows which address a communication partner could use in order to send packets to the node.

As embedded systems are often connected using wireless technologies, which also allow them to be mobile and to dynamically change their point of attachment to a backbone network, they frequently need to change their IP address, that is, the second property listed above doesn't any longer exist. Furthermore due to the huge number of mobile nodes it is no longer possible with IPv4 to assign a globally unique IP address to each of them. Therefore in order to still route their packets via the Internet, the IP addresses of these embedded systems will have to be translated at a so-called Network Address Translator (NAT) box into globally unique and therefore routable IP addresses. However, this causes also the first and the fourth property listed above to exist no longer.

Currently there become already new protocols available, which again introduce these missing properties of the original design, e.g., IPv6 will provide a sufficiently large address space for re-introducing property one and four, while Mobile IP provides an efficient support for mobile nodes and therefore re-introduces property two. However, there is a completely new approach in the process of standardisation, called the Host Identity Protocol (HIP), which will also re-introduce all lost properties, but which also re-introduces all the properties listed above, plus provides some additional features, such as

- an authentication service integrated in the name space,
- a clear separation of interworking layer from the higher layer, and therefore allow their independent further development, or
- no requirement for an administrative infrastructure to establish and maintain this name space, that is the names can be generated locally.

A.2.2. Overview of HIP

HIP focuses on the use of a clear separation of the identifier and the locator of a node. For the purpose of identification HIP assigns to each node a globally unique host identifier, which is generated from a cryptographically process. In more detail, each node has its own public / private key pair, from which the public key is used as the host identifier, that is, this kind of host identifier can be generated in a local manner, and can be used furthermore for authentication services.

For the locator part HIP uses whatever the underlying network is able to provide, e.g., for the Internet this will mainly be the IPv4 or IPv6 name space. Based on this principle HIP allows the upper layer and applications to make use of the static host identifier for their communication needs, that is, as these host identifiers will remain static, the upper layers and the applications can make use of a static name for their communications peers, which won't change during the mobility of a node.

Before sending packets via the underlying network, HIP has to map the host identifiers to the name space of the respective underlying network technology, that is, e.g., to IPv4 or IPv6 addresses. This design allows HIP to run on different underlying network technologies, and not only on the Internet. While this property sounds attractive, it currently is not clearly defined how this mapping may occur. The inclusion of the host identifier into the DNS is one option currently foreseen, however, in case the respective node is mobile and has to change its IP address, this change needs to be communicated to the communication peer. The options for doing so, those is, e.g., sending information directly to the communication partner, or otherwise for deploying rendezvous point storing the current mapping between host identifier and IP address, are described only very briefly so far.

In order to make use of its authentication feature, HIP specifies its own security negotiation mechanism, which results in setting up two unidirectional IPsec security associations between two communicating HIP nodes. This security negotiation mechanism is called HIP base exchange and is depicted in Figure A.2.

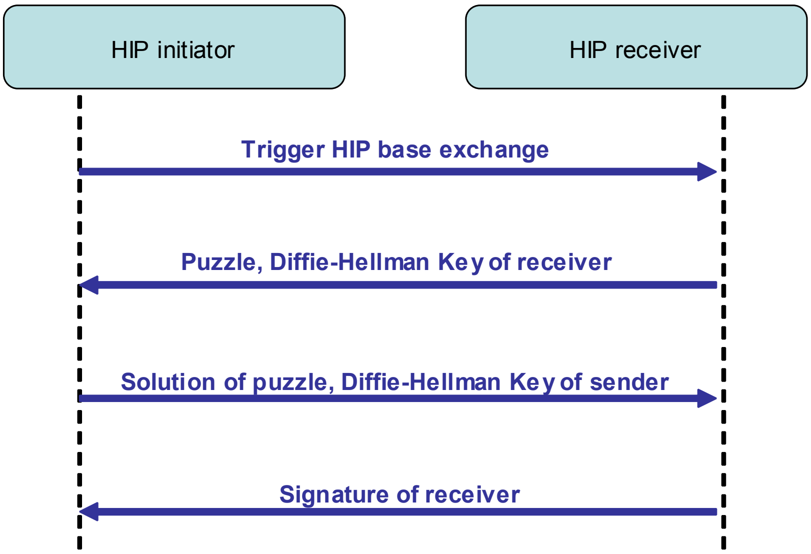

HIP initiator
HIP receiver
Trigger HIP base exchange
Puzzle, Diffie-Hellman Key of receiver
Solution of puzzle, Diffie-Hellman Key of sender
Signature of receiver

Figure A.2: Overview of HIP base exchange

After completing this HIP base exchange all the future HIP payload is exchanged in a protected manner using Encapsulated Security Payload (ESP) on the established IPsec security associations. Such a HIP payload e.g., could be the update of the communication peer with a new IP address recently obtained due to a new point of attachment to the Internet. However, it should be noted that the HIP specification clearly outlines that the use of public keys as host identifiers and their following use to set up IPsec security associations is not mandated, but strongly recommended. For using any other kind of identifiers HIP currently does not specify the relevant functionality.

A.2.3. Use of HIP for embedded systems

Theoretically, HIP can be used as a mechanism to separate locators and identifiers also on embedded systems. This would bring the tremendous advantage to embedded systems, that they can use the same identification scheme over different types of underlying internetworking technologies, such as a legacy IPv4 network, a new IPv6 network, a ZigBee based sensor network, or a vehicle bus system like the CAN bus. Additional advantages would be the support of the mobility of embedded systems, which would be completely transparent to the upper layers, plus the authentication and encryption functionality of HIP, which can be used in order to secure the communication between different instantiations of the middleware.

However, it also needs to be outlined that the HIP approach is still in the process of standardization, and therefore parts of it need still some major investigations. For example it needs more work on how to support mobility and multi-homing based on HIP. Concerning the use of HIP in embedded system environments also the appropriateness and overhead of the HIP security solution needs to be further investigated.