IMPLEMENTATION OF DISTRIBUTED COMPOSITION SERVICE FOR SELF-ORGANIZING SENSOR NETWORKS

Except where reference is made to the work of others, the work described in this thesis is my own or was done in collaboration with my advisory committee. This thesis does not include proprietary or classified information.

Udayan Naik

Certificate of Approval:

W. Homer Carlisle
Associate Professor
Department of Computer Science and Software Engineering

Alvin Lim, Chair
Associate Professor
Department of Computer Science and Software Engineering

Chung-wei Lee
Assistant Professor
Department of Computer Science and Software Engineering

Stephen L. McFarland
Acting Dean
Graduate School

IMPLEMENTATION OF DISTRIBUTED COMPOSITION SERVICE FOR SELF-ORGANIZING SENSOR NETWORKS

Udayan Naik

A Thesis

Submitted to

the Graduate Faculty of

Auburn University

in Partial Fulfillment of the

Requirements for the

Degree of

Master of Science

Auburn, Alabama
December 16, 2005

IMPLEMENTATION OF DISTRIBUTED COMPOSITION SERVICE FOR

SELF-ORGANIZING SENSOR NETWORKS

Udayan Naik

Signature of Author

Date of Graduation

THESIS ABSTRACT

IMPLEMENTATION OF DISTRIBUTED COMPOSITION SERVICE FOR SELF-ORGANIZING SENSOR NETWORKS

Udayan Naik

Master of Science, December 16, 2005
(B.E., Mumbai University, 1999)

114 typed pages

Directed by Alvin S. Lim

The heterogeneity and mobility inherent in large-scale ad-hoc sensor network accounts for the difficulty in developing applications in such environment. The difficulty primarily arises from weak and intermittent disconnection, dynamic reconfiguration, and limited power availability. The challenge in building a distributed service that simplifies adaptation to environmental change therefore lies in the degree to which communication and adaptation of these nodes can be automated or made transparent to applications.

Most applications in sensor networks involve group of sensor nodes coordinating to perform one task. Distributed composition service enables application to compose sensors to form a dynamic task group and maintains the group of sensors to achieve higher task availability. In this document, we describe the composition service and its implementation in detail and show how the process of re-organization can be automated via an adaptation server in collaboration with the composition service. Our testing and simulation validates the framework of the composition service and performance improvement in terms of service availability and energy preserving brought about by our adaptation algorithm, which is implemented with the support of the composition service.

ACKNOWLEDGEMENTS

The author would like to express special appreciation for his advisor, Dr. Alvin Lim, Associate Professor, Department of Computer Science and Software Engineering, for the opportunity to pursue this research and for the guidance provided. I would also like to thank the advisory committee members, Dr Chung-wei Lee and Dr. Homer Carlisle, and the professors and staff members in the Department of Computer Science and Software Engineering at Auburn University for their advice and help.

The author acknowledges all the individuals who have time and again put his interests before their own, notably Mark Ivester and Kelly Houts.

Above all, the author is grateful to his parents, without whose support and encouragement this would have not been possible.

Style manual or journal used: Guide to preparation and Submission of Thesis and Dissertation

Computer software used: MS Office 2003

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER 1: INTRODUCTION

The last few years have seen an explosion in the field of microelectronics. Devices are getting not just smaller but also more powerful in terms of their capabilities. A typical mobile phone today encompasses the functionality of a PDA, a camera and access to the Internet. The IT industry is moving fast towards the human/machine/network breakpoint – the point at which the number of networked interactive devices will surpass the number of people on the planet. However, according to Dr. Tennenhouse in [1], the bulk of the industry is still focused on office automation, e-commerce and the associated networking of these devices. It is believed that the distribution of new computers will be dominated by millions of new laptops, desktops and server nodes that will power the growth of interactive computation. Although these numbers are impressive, they are miniscule in comparison to the vast number of computational nodes that will be embedded in other objects. Rather than being in direct contact with humans, they will be in contact with their environments, able to sense and affect physical phenomena. These *sensor nodes* have a variety of applications; from sensing hostile vehicles in the battlefield to monitoring animal habitats. Dr. Tennenhouse envisions a future in which sensors will be integrated into our daily lives, working proactively and responding to external stimuli in a seamless manner.

Unlike traditional computer networks where connecting devices to the network use relatively static network configuration, in a sensor network, sensor nodes are networked in an ad-hoc manner with no fixed infrastructure. Although sensor nodes are equipped with a power supply and an embedded processor that makes them autonomous and self-aware, the functionality and capability of a single node is very limited. A large number of sensor devices must be deployed in most impromptu networks for each sensor device to organize itself in the overall community of sensors and perform coordinated activities with global objectives. In such a large network, users do not interact with any one device, but rather avail of the service offered by a set of sensors. Collaboration between nodes to offer certain services is thus essential to deliver relevant and useful data in a ubiquitous setting.

Till now, most of the research for sensor networks has concentrated on operating platforms for sensors or defining more efficient routing protocols. The results of these efforts are milestones such as Berkeley SmartDust [3], TinyOS [4] and routing protocols such as Directed Diffusion [5] and Dynamic Source routing [6]. Though advances in these areas are a great step forward in moving towards the vision of pervasive computing described in [1], the heterogeneous nature of the development environment is a big factor when it comes to implementing applications for a sensor network. Software is written for target platforms and cannot be easily ported to different hardware or operating systems. The blurry demarcation between software and hardware also means that developers have to take into account specific routing protocols along with other low-level system details and incorporate these into their applications. These factors are deterrents to implementing a sensor network that aims to be self-organizing via inter-sensor communication. A framework that provides developers with a conceptual view of the network and insulates them from the lower layers is needed.

In [2] Dr. Lim defines some of the characteristics for creating such self-organizing sensor networks as agility, self-awareness, self-configurability and autonomy. Nodes are aware of their own capabilities and those of others around them which may provide the services or resources they need. A group of these sensors may communicate and cooperate to deliver coordinated services while other nodes may be deployed or removed from the community spontaneously. However, building such networks is difficult for the following reasons. First, there are many different types of sensor with different capabilities which may be deployed with specialized and possibly non-interoperable network protocols and application requirements. Secondly, sensor nodes may be deployed incrementally with little or no pre-planning. The network must survive harsh environmental conditions, changes in sensor composition, task requirements, device failure and mobility of sensors.

The underlying principles that can overcome the above challenges are to provide the fundamental services upon which other networking services may be spontaneously specified and reconfigured. In [2] Dr. Lim describes three fundamental mechanisms, namely, the lookup service, the composition service and dynamic adaptation service.

The lookup service enables new system and network services to be made available to other sensor nodes. The composition service allows clusters of sensor nodes to be formed and managed. The adaptation service allows nodes and clusters to reconfigure dynamically as a result of node mobility, failure and deployment. All these services are designed to simplify and facilitate application development, insulating them from network dynamics.

Some of the applications being considered for sensor networks are multi-sensor fusion, collaborative target tracking and distributed query processing. In order for these applications to perform their tasks in an efficient manner, all the participating nodes need to set up and maintain connections among themselves. When dealing with such an environment, there are some issues that must be considered. Firstly, nodes may need to find and communicate with specific, and possibly remote, nodes for the application to give better results. Secondly, target nodes or intermediate nodes may fail, negatively affecting the application. Even if new nodes are available to replace the failed ones, existing nodes will have to be aware of their availability.

To ameliorate some of the above problems, applications should be able to depend on the distributed composition service providing the following services.

1. **Efficient Task Group Composition:** Multiple nodes of different types and specifications can contact the composition server to form a group of sensors participating in a common task, like target tracking.
2. **Automatic adaptation to failure:** The configuration data stored by the composition server can be used by some adaptation service to adapt to device and network failures.
3. **Optimize task performance:** The stored configuration data can also be used by the adaptation server to optimize and reconfigure all entities involved for better performance during runtime.

This research will aim to successfully undertake the following tasks

- To design and implement a usable API for group composition and communication. The API will provide software developers with a look and feel consistent to current network programming paradigms.
- To investigate the issues involved in applying the composition service to accomplish distributed task executing over a sensor network.

Chapter 2 presents some of the related technologies needed to construct a composition service. Chapter 3 discusses the architecture of the composition service and implementation of the application programming interface is presented in Chapter 4. The performance of the composition service under various scenarios is presented in Chapter 5. Chapter 6 concludes by giving a summary of this research and suggesting further work that could be done.

CHAPTER 2: RELATED TECHNOLOGIES

There are a number of separate areas of research on which this work is based, and we discuss related technologies in the following areas.

1. Directed Diffusion

Directed Diffusion [5] is a routing protocol for autonomous sensor networks developed at the University of Southern California. A departure from other routing protocols, data is named using attribute-value pairs which makes directed diffusion a data-centric routing protocol.

2. Distributed Services

As presented in the previous chapter, distributed services provide the foundation on which other application can be built. Some examples of distributed services are lookup service, adaptation service and composition service.

3. ISEE

ISEE is a runtime framework for the execution and monitoring of sensor network services, which makes allowance for simulated, emulated and real sensor network control and access. It allows for extensibility, scenario creation and experiment repeatability, providing a visibility to sensor network experiments.

4. Component Based Design Methodology

Being able to find, adapt and incorporate disparate components to form working, reliable applications is the goal of component based software engineering. This fits in well with the heterogeneous nature of a sensor network environment where nodes require special consideration in terms of their physical resources and computing power. Modeling sensor types, their interfaces and a specification of the patterns of interaction between sensor types will help to build an efficient service framework.

2.1 Directed Diffusion Protocol

Directed diffusion is a data-centric routing protocol in the sense that routing decisions are made on the basis of the data requested rather than specific end points of the data. Directed diffusion is also a localized protocol; nodes know and interact only with their neighbors. Data generated by a node is named using attribute-value pairs. A *sink* node sends out an *interest* for the kind of data it is interested in. This initial interest is broadcasted and spreads through the network until some node has the data that matches the interest. Intermediate nodes establish *gradients* towards every neighboring node that forwards this interest. The *source* node has data that matches this interest and sends out this data on the preferred gradient to its neighbor, which then follows the same rules for forwarding this data. In this manner, interests and data *diffuse* throughout the network. Once the data reaches the sink node, it sends out a *positive reinforcement* with a higher data rate than the exploratory initial interest to the neighbor with the lowest delay. Other nodes may be sent a *negative reinforcement* to inhibit them from forwarding data packets. The process is shown in Figure 2.1

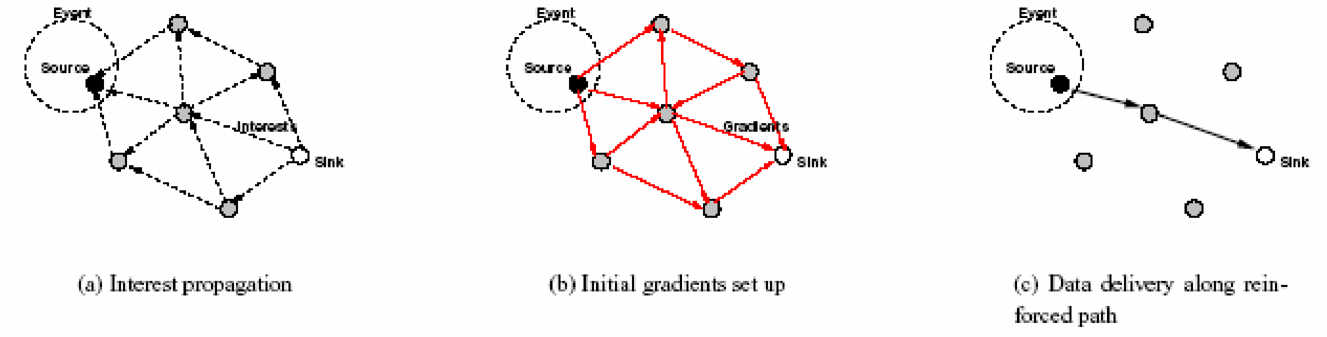

Figure 2.1 A simplified schematic for directed diffusion

For this research, we used directed diffusion implemented by the Information Science Institute, University of Southern California. Their implementation provides access to the diffusion core behavior through the *publish* and *subscribe* API functions. A sink sends out interest messages by subscribing to the kind of data it is looking for and listens to incoming data via publish. Likewise the source node uses subscribe to listen for interest messages and publishes matching data. Detailed explanation of the API and usage techniques is given in [7] and was used effectively in Yu's thesis [8].

2.2 Distributed Services

The three fundamental services described by Dr. Lim in [2] are the lookup service, the adaptation service and composition service. Through the implementation of these services, other network and system services can be defined spontaneously in the network. The distributed systems architecture is illustrated in Figure 2.2.

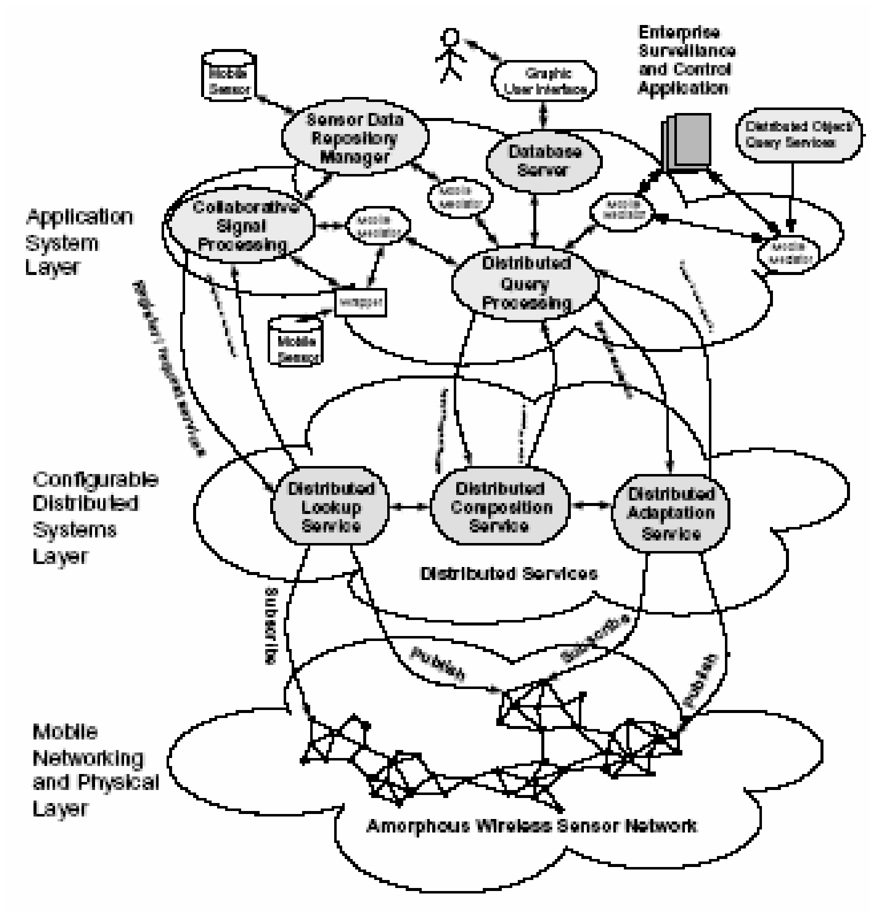
Enterprise Surveillance and Control Application
Graphic User Interface
Mobile Sensor
Sensor Data Repository Manager
Database Server
Distributed Object Query Services
Application System Layer
Collaborative Signal Processing
Distributed Query Processing
Wrapper
Mobile Sensor
Configurable Distributed Systems Layer
Distributed Lookup Service
Distributed Composition Service
Distributed Adaptation Service
Distributed Services
Subscribe
Publish
Subscribe
Publish
Mobile Networking and Physical Layer
Amorphous Wireless Sensor Network

Figure 2.2 Architecture of self-organizing sensor networks

Lookup Service: The primary function of the lookup service is to store information about the services offered in the network and distribute this information. The lookup service is available to an application via the lookup service API. If an application provides a service, it uses the *service_register()* call to register itself with the lookup server. A service-seeking application, given a service name, calls *lookup_service()* to obtain the service details – the type of and the order of the arguments needed to invoke the service. A client can also obtain the interfaces to all service providers that provide the same kind of service it is seeking and choose one among them to request the service from. This function enhances the availability and usability of the lookup service as the client can still avail of the service even if the specific service name is not known. Lookup servers are distributed over the entire network so that there is not excessive demand on one server and a demand-supply bottleneck is avoided. The information stored by one particular server can be exchanged with other servers when requested. Further information about lookup service implementation and improvements are found in [8].

Adaptation Service: An adaptive distributed system is one that modifies its behavior based on changes in the environment. The adaptation system is based on a general distributed system model consisting of three phases: change awareness, consensus and system behavior modification. The change detection phase monitors the possible change in the environment. All units affected by the change communicate with the adaptation service and decide on the action to be taken in the consensus phase. Once a suitable course of action is determined, the adaptation service then initiates a series of steps that results in system changing its behavior. Using the adaptation service, algorithms can be devised that complement the dynamic and changing sensor network environment. The composition service works in hand with the adaptation service to mask node failure due to environmental or network changes and incorporate fault tolerance into its operation. Adaptation service details are found in [9].

2.3 Integrated Sensor Network Execution Environment (ISEE)

Sensor networks are a relatively new area of research and it is not always possible to obtain actual sensors to carry out experiments. The solution is to simulate a sensor network by setting up nodes over fixed and easily available infrastructure and run distributed applications on top of that. These projects need a simple runtime framework for repeatable experimentation that allows for easy transfer to on-site set ups. Such an environment is necessary as simulation tends to present a misleading picture when environment sensitive and resource constrained networks are being simulated, simply because the infrastructure is not bound by these constraints.

A runtime framework that transparently supports testing simulated, emulated and real sensor networks is needed. Along with this requirement comes the necessity of event capturing and measurement to portray an accurate view of real world sensor events.

The Interactive Sensor Network Execution Environment ISEE [12] developed by Mark Ivester at Auburn University which serves to address such issues, is composed of four main modules, shown in Figure 2.3.

1. *Isview:* The Isview module handles the user interaction and sensor network visualization. Through Isview the user can direct the formation of the network, create processes on a single target node and view the sensor network status. State information is retrieved from the network through the Ethernet back channels every second and is used to update the information displays. Aspects of the network that can be monitored include application state, debug information, resource load and network load, among others. Isview also provides the interface for defining services that will run within the netwok.

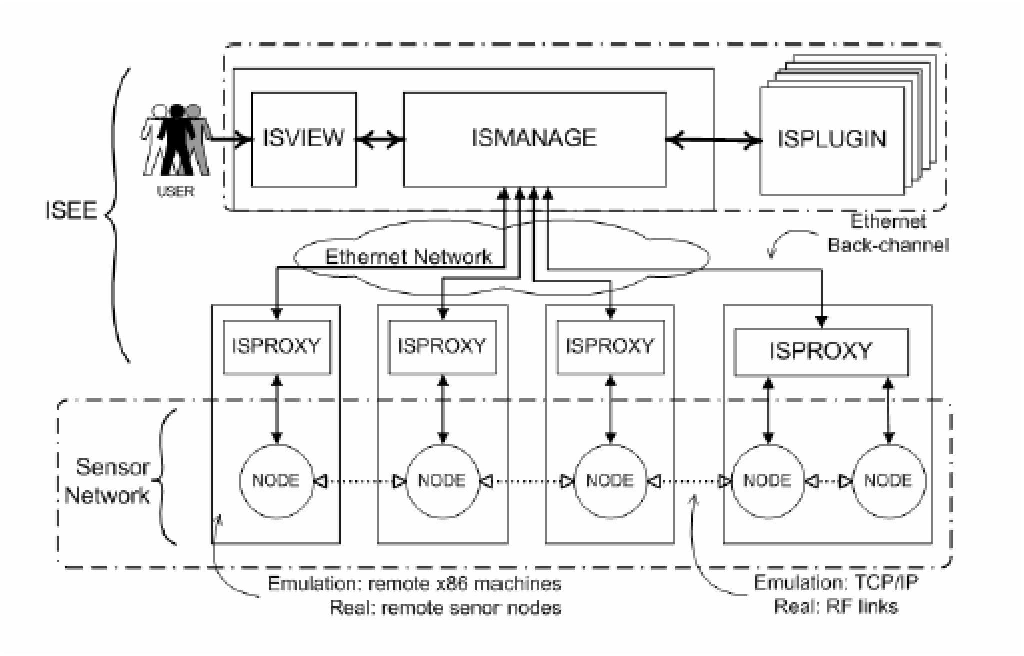

Figure 2.3 ISEE Architecture Overview

2. *Ismanage:* Ismanage has three responsibilities: running sensor network management, sensor network communication and runtime plugin management. Management of the network involves maintaining state information of the nodes and the processes running within the sensor network. State information includes individual link statuses, node status and traffic status. Communication with the sensor network is the means by which control information is sent into the network and feedback information is obtained from the network. Control information is either a set of commands to be run on multiple nodes or state information to be injected into the network. Feedback information may include sensor network state information or specific application information. Management of plugins is the key to the extensibility of ISEE and comprises of two steps. The loading of plugins consists of retrieving the particular plugin from predefined global and user directories. Once loaded, the plugin is

activated when the user issues a directive. A plugin may proactively or reactively affect the ismanage maintained representation of the sensor network state based on the feedback obtained from ismanage.

3. *Isplugin:* An isplugin is dynamic code that extends the ismanage module, enabling it to deal with different sensor network processes or protocols. An isplugin may also extend the capability of the isview module. Though isplugin is managed by the ismanage module, it may interact directly with the ismanage or isview module.
4. *Isproxy:* The isproxy module is remote code that can manage the individual nodes in the sensor network, although it can handle the responsibility of handling multiple nodes in practice. Management of individual nodes includes starting, stopping and monitoring processes, information about which is collected and sent to ismanage as feedback data. The domain of isproxy's responsibility is flexible and user-defined to be either physical or gateway. In the physical mode, the isproxy module is responsible for all the virtual nodes running within the context of execution, a physical machine or real sensor node. In the gateway mode, isproxy is concerned with a subset of the sensor network, using the network to send control messages to and receive feedback from applications running on other sensor nodes.

2.4 Component Based Engineering

The motivation for adopting component based engineering methodology for any project is increased productivity obtained by using pre-constructed components for development. However, the advantage of applying this design methodology to embedded devices goes beyond that; a developer should be able to concentrate on creating functionality that best serves the task at hand and delegate the responsibility of complex low-level interaction to *components* [10] constructed specifically for that purpose. A component is an independent and replaceable part of the system that fulfills a clear function in the context of the architecture. Components are described in terms of the *interface* that provides access to component functionality. Adopting the component based methodology in building a sensor network system will allow developers to rely on underlying existing APIs for device interaction and data transport.

Conventional interface specifications define the functional properties which express the service provided by that component and the signature of the service. But in order to achieve mobility and adaptability, a sensor environment needs to make optimal use of the available resources. For this to happen, components need to not only publish their interfaces and protocols for interaction, but also make known their resource requirements such as required bandwidth, available power, computational load, associated delay etc. Such *quality of service* attributes will enable sensor nodes to adapt in a better way to network dynamics and will also help the composition service in organizing nodes with compatible interfaces. Interfaces of components are compatible only if they have the same data format, complementary data flow direction and matching quality of service requirements.

An advantage of sensor networks is the possibility of implementing cooperative, possibly localized, algorithms for potential applications like target tracking and resolution, distributed query processing etc [13]. In all cases, when a cooperative function is required to exact information about a specific target, a local network comprising of a small set of nodes near the target location is built and communication is facilitated between them.

The composition service handles forming and managing this task group, controlling runtime resource binding. For this to happen, nodes need to communicate with the composition server and each other. However, unlike in a traditional Internet architecture, in a sensor network nodes are aware only about their own neighbors.

Additionally, nodes may not possess unique identification like an IP address for two reasons: there is no central authority handing out a unique ID and since the network is data driven, it makes sense to use a data-centric addressing scheme to target nodes for communication.

To provide higher level abstractions to application programmers, we model a node as a sensor object with its own interface. A *connector* specifies a data sharing relationship between compatible interfaces of such nature, and has a name and type. A description of this model will be given in the next chapter.

Traditional component-based techniques deal with finding and integrating components in a static configuration which usually run in a tightly-coupled set up such as a node or cluster. However, sensor networks have to deal with dynamic availability of data sources and make allowance for inherent distribution of sensors, constrained resources and environmental impediments. The sensor composition process model should be robust and transparent to the application developer, providing encapsulation of network complexity. A detailed explanation of this model in provided in the next chapter.

2.5 Related Work

Wireless sensor networks are an increasingly attractive means to monitor environmental condition and to map the bridge between the physical and virtual world. The network consists of a large number of cooperating small nodes capable of limited computation, wireless communication and sensing. By matching the output of several nodes, the network as a whole can provide functionality a single node cannot. However there are hurdles with deployment beyond the hardware limitations. There are chances for dynamic link failure, node failure and node mobility, all of which result in changing network conditions. The focus of much of the research in sensor networks has thus turned to defining middleware that sit above the operating system and below the application, abstracting lower-level functionality such as network connectivity and providing a coordination interface to the application. Much of the work has targeted the development of middleware platforms specifically designed to meet the challenge of the resource-constrained aspect of these sensor networks. We discuss some of the common middleware in the following section.

2.5.1 MiLAN (Middleware Linking Applications and Networks)

Wendi Heizelman et. al. describe the MiLAN middleware platform developed at the University of Rochester in [14]. The idea behind MiLAN is that additional savings can be achieved if the middleware varies the actual parameters of the network over time while simultaneously meeting the requirements of the application, increasing the lifetime of the network. Applications represent their requirements to MiLAN through specialized graphs that incorporate state-based changes in application needs. Based on this information, MiLAN makes decisions about how to control the network and the sensors to balance application QoS and energy efficiency, lengthening the lifetime of the application.

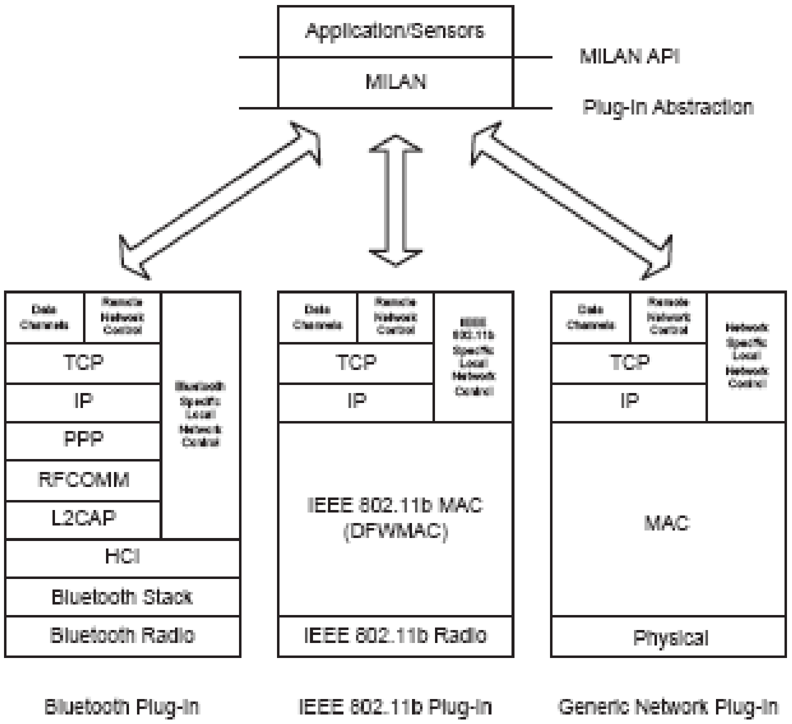

Figure 2.4 MiLAN protocol stack

Unlike traditional middleware, the MiLAN architecture, seen in Figure 2.4, extends into the network protocol stack. An abstraction layer is provided that allows network specific plug-ins to convert MiLAN commands into protocol specific commands that are passed through the usual network protocol stack. MiLAN can thus continuously adapt to the specific features of whichever network is being used to communicate to best meet the application’s needs over time.

MiLAN deals with the application’s interoperability with the network and does not inherently support any clustering techniques. In addition, the existence of a plug-in for MiLAN that supports Directed Diffusion or any other routing protocol commonly used in sensor networks is not confirmed. The storage requirements on the node for handling the MiLAN state-based graph for a complicated and dynamic network application are also not specified.

2.5.2 Impala

The Impala system [15] is a part of the ZebraNet project [16] at Princeton University. It is a mobile sensor network system aimed at improving tracking technology via energy-efficient tracking nodes and peer-to-peer communication techniques. The main focus of ZebraNet is wildlife tracking across large regions with little communication infrastructure.

While middleware like MiLAN, presented in the previous section, focus on the form of data presented to user applications, Impala considers the application itself, exploiting mobile code techniques to change the functionality of middleware executing at the remote sensor. Since many sensor networks will be deployed in harsh environments, they are intended to run without user intervention for long amounts of time. Impala advocates considering long-term management of the sensor application as an integral part of the design process.

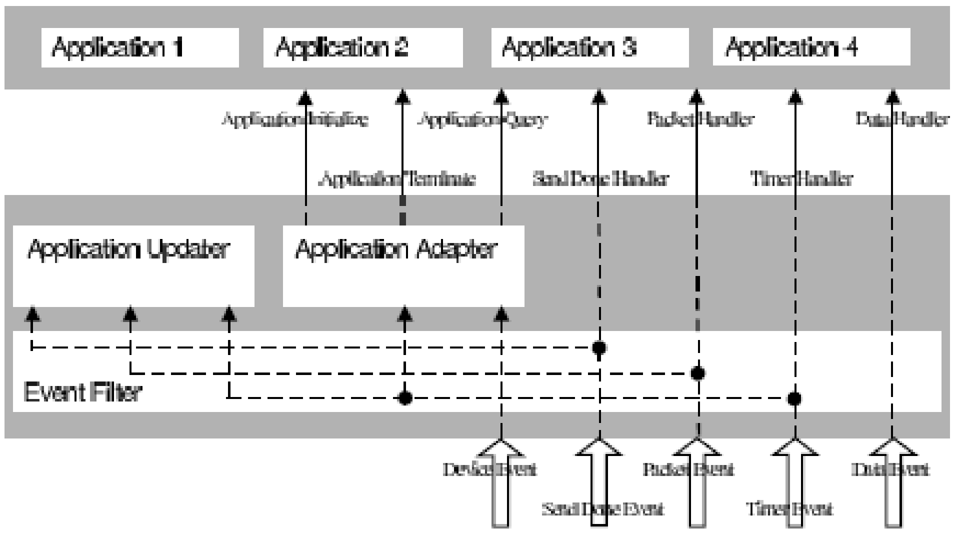

Figure 2.5 Impala system architecture

Figure 2.5 shows the Impala architecture. The upper layer contains all the application protocols and programs for ZebraNet. These applications use various strategies to achieve a common task of gathering the environment information and routing it to a centralized base station via peer-to-peer transmission. Only one application is running at a time.

The lower layer contains three middleware agents: the Application Adapter, the Application Updater, and the Event Filter. The Application Adapter adapts the application protocols to different runtime conditions to improve performance, energy-efficiency and robustness. The Application Updater receives and propagates software updates and installs them on the node. The Event Filter captures and dispatches events to the above system units and initiates chains of processing. Impala has five types of events.

A **Timer Event** signals that a timer has gone off. Impala has three timers owned, respectively, by the current active application, the Application Adapter, and the Application Updater. The owner of the timer handles these events. **Packet Event** signals that a network packet has arrived. Impala has two types of packets, application-to-application packets and updater-to-updater packets. The intended receiver of the packet handles these events. **Send Done Event** signals that a network packet has been sent or has failed to send. It allows asynchronous network transmission. The original sender of the packet handles these events. **Data Event** signals that a data sample from the sensing device is ready to read. The current active application handles these events. **Device Event** signals that a device failure is detected. The Application Adapter handles these events. When multiple events arrive at the same time, they are processed sequentially.

The layered approach of Impala has several advantages.

- **Modularity:** Applications can be independent and do not need to coordinate with each other for updates with the middleware layer also handling the update issues.
- **Correctness:** Impala makes application correctness easier to achieve because programming individual applications is simpler than programming a complex application with many interacting and updating components.
- **Ease of Updates:** Software changes such as adding, deleting and modifying an application can be simpler because they can involve only local code changes within a module.
- **Energy Efficiency:** Software updates are comparatively smaller program modules. Since the network transmitter is the most power hungry component, this offers significant energy savings.

As said above, Impala concentrates on the mechanisms to change middleware behavior by using mobile code updates. The key to energy efficiency in Impala is for the sensor node applications to be as modular as possible, thus enabling small updates that require little transmission energy. Applications must be then written specifically using the event-handling interfaces provided by the middleware, limiting the complexity of the application. No constructs for clustering or grouping of individual nodes are present in the Impala programming API.

2.5.3 DSWare

The Data Services Middleware [17] is based on the notion of events, whereby the application specifies interest in certain state changes of the physical world, called *basic events*. Upon detecting an event, the node sends an event notification towards interested applications. The application can also specify a certain pattern of events such that the application is only notified if occurred events match this pattern. The real data generated by the sensors can be stored or forwarded to other nodes for processing. Since this functionality will be desired for all types of sensor applications, a data-services middleware can avoid the re-implementation the common data service part of various applications. The DSWare layer exists between the application layer and network layer. The architecture of DSWare is illustrated in Figure 2.6.

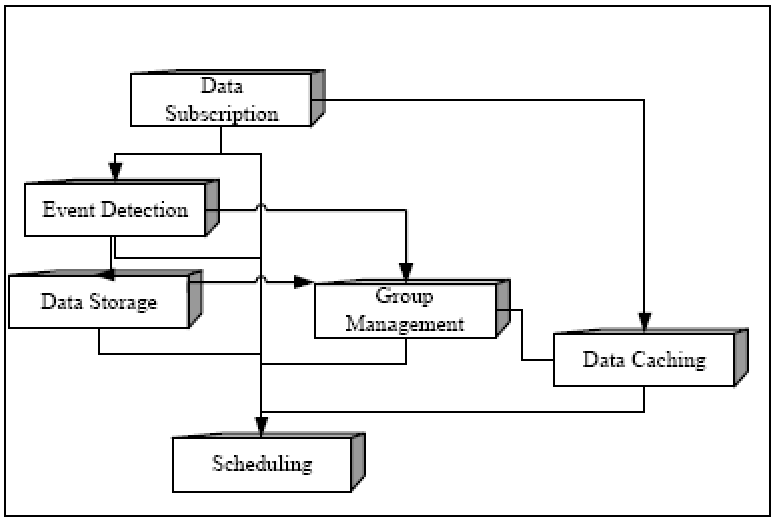

Figure 2.6 DSWare Framework

- **Data Storage**: The data storage module provides mechanisms to store information according to its semantics with efficient data lookup and supports robustness during node failures by supporting data-centric protocols. Data that describes different occurrences of some type of activity can be mapped to certain locations so that future queries for this type are not flooded to the whole network. The data lookup scheme uses hashing to map data to physical storage nodes using the unique identifier for the data. When a base station sends queries for this data, the information is fetched from one of these physical locations. Data is also replicated in several physical locations that map to one logical node. Queries are directed to any one of these locations to avoid high load on any one node, providing a degree of robustness to the system.

- **Data Caching**: The data caching service provides multiple copies of the data most requested. This data is spread out over the routing path to reduce communication, increase availability and fasten query execution.
- **Group Management**: The group management component provides localized cooperation among sensor nodes to achieve a more global objective. Mostly, groups are formed as the query is sent out and dissolved when the query is expired or the task is accomplished. Hence, the group formulation criterion is sent to the queried area first and nodes decide whether to join this group by checking whether they match the criterion. For a dynamic group, changed criterion is broadcast persistently in a small area whose center is the current group. Hence, nodes can join and leave the group when the target moves. Groups not sensitive to tasks can be formulated during system deployment or when explicitly specified by the applications.
- **Event Detection**: In the event detection service, events are pre-registered according to the specific application. Event detection is a common and important service in sensor networks.
- **Data Subscription**: As a type of data dissemination service, Data Subscription queries are very common in sensor networks. These queries have their own characteristics, including relatively fixed data feeding paths, stable traffic loads for nodes on the paths, and possible merges of multiple data feeding paths. When several base stations subscribe for the data from the same node at different rates, the Data Subscription Service places copies of the data at some intermediate

nodes to minimize the total amount of communication. It changes the data feeding paths when necessary.

- **Scheduling**: The Scheduling component is a special component because it provides the scheduling service for all components in DSWare. Two most important scheduling options are energy-aware and real-time scheduling. By default, a real-time scheduling mechanism is applied as the main scheduling scheme because most queries in sensor networks are inherently real-time tasks. Applications can specify the actual scheduling schema in the sensor networks based on the most important concerns.

DSWare meets most of the goals specified for the distributed composition service by providing support for group creation. Additionally, it also supports data-centric routing protocols like Directed Diffusion and though computationally intensive, provides facilities for information storage and retrieval. However, the creation of a group is triggered by a real-world event and requires the node to correctly identify and categorize an event. In the heterogeneous sensor network, not all nodes may possess the computational capability for this task. There is also no single entity to manage the group formation and handle group information storage. This ability is vital for the composition service to inter-operate with the adaptation service.

CHAPTER 3: COMPOSITION SERVICE ARCHITECTURE

This chapter presents the design methodology and architecture of the composition service. Efforts have been made to ensure the design is simple and the system is inter-operable with other network services.

3.1 Design Goals

Over the past decade, various technologies have been devised to alleviate the complexities associated with developing software for distributed applications. Some of the most successful of these technologies have centered on distributed object computing (DOC) middleware. Distribution middleware defines higher-level programming models whose reusable APIs and components automate and extend the native network programming capabilities such as connection establishment or interprocess communication and synchronization. Some examples of DOC middleware are OMG's CORBA and Sun's RMI. A complete list, brief descriptions and requirements to construct systems using the mechanisms provided by these platforms are found in [22]. However, the basic mode of operation of wireless sensor networks is significantly different from traditional computer networks, primarily due to their resource-constrained nature and tight integration with the physical world.

From a study of DOC middleware and an understanding of sensor network characteristics, a set of design goals can be formulated, as given below.

1. Shield software developers from low-level, tedious and error-prone platform details, like socket-level network programming
2. Provide a consistent set of higher-level network oriented abstractions that are much closer to application requirements in order to simplify the development of distributed systems.
3. Provide a wide array of developer-oriented services such as event capturing and logging that have proven necessary to operate effectively in a networked environment.

3.2 Composition Service Mechanism

Section 2.2 introduced the architecture of distributed system services for sensor networks. By using these services, smart nodes may simultaneously provide services to other smart nodes and be clients of services that other such nodes provide. Nodes may be dynamically composed into impromptu networked clusters under the management of a compositional server. The clustered nodes can then cooperate to provide abstract services to the dynamic sensor network, such as data filtering and aggregating summary information.

Clustered smart nodes encapsulate the networking and system capabilities provided cooperatively by the group of smart nodes. There will be a head node in the cluster that is responsible for the control of the cluster and inter-cluster communications and networking functions. Group communication to nodes in a cluster can be efficiently implemented by sending a message first to the cluster head which then multicasts it to the member nodes. Smart nodes in a cluster may cooperate to perform the networking and system functions for the cluster.

The composition service will also manage the various smart nodes that may be added or removed from the clusters in the agile sensor network. This composition request-response process is explained using Figure 3.1 in a series of steps.

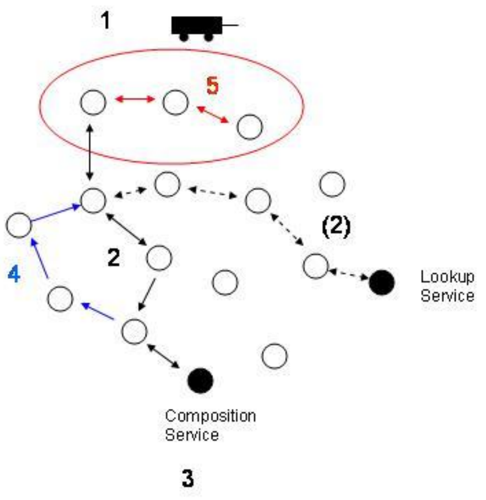

Figure 3.1 Composition Service Mechanism

1. A group of sensor nodes decide to form a group, either in response to some external stimuli such as target detection or nodes in close proximity decide to form a group in anticipation of future events. If the node holds the capability to accurately discern the event, it can send this information to the server. The event is described in terms of its event category and has a type, value and timestamp. For example, a tank detection event has category ‘vehicle, type ‘tank’, a value

which possibly might indicate the confidence of the reading and the associated timestamp.

2. The node(s) desirous of group formation contact the composition server requesting to join/create a group. Details about the event can be communicated to the composition server which may or may not use it. If the identity of the composition server is unknown, the lookup service is queried for that information.
3. The composition server receives this request and keeps a record of all the nodes it receives group creation requests from. The information in the server is time-sensitive; the event information may not be necessarily applicable after a certain amount of time has passed, especially in the case of mobile objects.
4. Within the prescribed time limit, if the composition service gets requests from sizeable number of nodes, it assigns a group ID and chooses a group leader among the nodes. Only nodes in close proximity to each other are chosen to be included in a group; this implies some location information is known to each node. Group information, group leader identity and status of all the other nodes in the group is communicated to the nodes in the group in the reply from the server.
5. The nodes receive this group ID and can then communicate among themselves to perform coordinated activities, using this group ID to uniquely identify the group they are currently part of.

3.3 Sensor Object Model

In the sensor network environment, use of data-centric protocols necessitates the addressing of nodes by the data they generate rather than by unique network identification. The sensor object model facilitates inter-sensor communication by categorizing nodes based on the data they generate and the services they offer, treating nodes as data handling entities. An application can use the sensor object model to encapsulate the properties of the node it is running on. The following describes the model in detail. The representation of node characteristics is shown in Table 3.1.

1. Every sensor object has an ID which is passed to the composition server when a join group request is sent. This ID is used for unique identification purposes along with the sensor node properties.
2. A real world event is encapsulated by the Event object, illustrated in Table 3.2. The composition server associates a SensorObject with an Event object and uses this information to make group creation decisions, grouping together objects that sense similar events. Event objects have a *category*, *description*, *value* and a *confidence* reading. The category and type of an Event depend on the data generated or processed by that node. The naming of these categories and types is thus an important task since it facilitates efficient data-centric routing and resource location. Naming techniques are discussed in [19].
3. Some method of energy conserving geographical routing like GEAR [20] can be applied in conjunction with directed diffusion for achieving higher routing efficiency. Thus, every object has location information specified by the *latitude* and *longitude* properties of that object. Location information can be obtained from a Global Positioning System and updated as and when the sensor node moves. This information is also used by the composition server when assigning groups as nodes within a predefined, flexible region are included in a group.

```
class SensorObject{                         //representation of a sensor node
public:
        static const int INACTIVE = 0;
        static const int ACTIVE = 1;
        static const int PRODUCER = 0;
        static const int CONSUMER = 1;
        static const int PEER = 2;

         SensorObject(int ID);
        SensorObject(int status, float latitude, float longitude, QosParams*, int role);
        SensorObject(int status, float latitude, float longitude, int role);
        SensorObject(float latitude, float longitude,int role);
        SensorObject (NRAttrVec*);

         float getLatitude();
        float getLongitude();
        int getStatus();
        NRAttrVec getSensorAttributes();

        void setLatitude(float latitude);
        void setLongitude(float longitude);
        void setStatus(int status);
        void setRole(int role);
        void setQosParams(QosParams*);
        void setGroupID(int);

        int getID();
        int getGroupID();
        QosParams* getQosParams();
        int getRole();

        void print();
        int size();
        int compareTo(SensorObject*);

private:
         int ID;
         int groupID;
         float latitude;
         float longitude;
         int status;                        //0 for inactive, 1 for active
         int role;                          //0, 1 or 2 for producer, consumer and peer
         QosParams *currentQos;      //the QOS parameters for this node
}
```

Table 3.1 Representation of a sensor node as a SensorObject

```
class Event{
public:
        Event(char*, char*, float);
        Event(char*, char*, float, int);
        Event(NRAttrVec*);

         char* getCategory();
        char* getDescription();
        float getValue();
        int getConfidence();
        int getValidity();
        NRAttrVec getEventAttributes();

        void setCategory(char*);
        void setDescription(char*);
        void setValue(float);
        void setConfidence(int);
        void setValidity(int);
        void print();
        int compareTo (Event*);

private:
         int valid;
        char *category;
        char* type;
        float value;
        int confidence;
}
```

Table 3.2 Representation of an event as an Event object

4. Some method of energy conserving geographical routing like GEAR [20] can be applied in conjunction with directed diffusion for achieving higher routing efficiency. Thus, every object has location information specified by the *latitude* and *longitude* properties of that object. Location information can be obtained from a Global Positioning System and updated as and when the sensor node moves.

This information is also used by the composition server when assigning groups as nodes within a predefined, flexible region are included in a group.

5. Nodes can also publish their current quality of service attributes by way of a QosParams object as shown in Table 3.3. One example of a quality of service attribute is the *power level*, indicating the current power of that node. Additional attributes are the *maximum data rate* the sensor node can transmit/receive at and the *delay* experienced by the node as a result of its queue length. These extra-functional properties can be used by the composition server to match compatible sensors in a group to enable more efficient application operation.

```
class QosParams{
public:
        QosParams();                    //no-arg constructor
        QosParams(int,int,int);         //initialize to respective  values
private:
        unsigned int powerLevel;        // current power level of node
        unsigned int maxDataRate;       //data rate supported by node
        unsigned int delay;             //delay in milliseconds
}
```

Table 3.3 Representation of Quality of Service parameters as an QosParams object

6. In a sensor network, nodes fulfill a clear function in the context of the environment; some sink nodes query the network for certain types of data while source nodes generate data and send this data to the sink nodes. A sensor object can specify the *role* it plays in the set up via the role attribute which can have three values

   a. PRODUCER – This sensor object is a data producing source
   b. CONSUMER – This sensor object is a data consuming sink
   c. PEER – An intermediate node

7. In addition, a sensor node can mark itself ACTIVE or INACTIVE using the *status* attribute. Some examples of why a node may decide to be INACTIVE are if it experiences problems with its physical sensors or if its power level goes below a certain threshold. This is for inter-operation with the adaptation server and will be explained in a later section.

3.4 The Connector Model

A sensor network experiences extreme dynamics by virtue of the sensor nodes and the entire system being tied to the physical world. In particular, environmental factors influence the characteristics of the RF communication channel, creating inaccessibility or mobility even in stable configurations. Device failure due to power outage or the node being physically destroyed is an additional factor which can cause interruption of the data flow in the network. The *connector* abstraction serves to mask these problems and provides the application with a consistent view of the underlying network.

A connector represents a data exchanging relationship between two or more sensor objects. Using connectors, sensor objects can establish connections with other sensor objects for the duration of the task. Connectors are reconfigurable in the sense that the adaptation server can replace inactive sensor objects with other sensor objects to achieve reliable data association between existing sensor objects. The model is as described below.

1. A connector maintains a reference to the invoking sensor object and the group ID to which the sensor object belongs. Using the group ID, the connector can establish a data path with other sensor objects in the same group. The sensor object can then use the connector reference to communicate with other sensor objects in the group. In a way, connectors are similar to sockets in an IP-based network; they are the endpoints of the communication. It is also possible that some computationally superior sensor objects form a group to offer services such as in-network data aggregation and location triangulation. Service seeking nodes can avail of these services by using the group ID once the group information is registered with the lookup service. The attributes of the connector object are shown in Table 3.4
2. A connector is made up of *links* from the current sensor to other sensors. A Link is the physical representation of the connection between two or more sensor objects. Links contain information about the endpoints of the connection and work with the routing protocol to identify these endpoints and efficiently discover and/or maintain paths to them. Figure 3.2 illustrates the relationship between a connector and link. A connector can contain links to more than one sensor objects depending on the type of the connector.

```
class Connector
{
public:
        Connector(){};
         int addEndPoint(SensorObject *addNew);
        int removeEndPoint(SensorObject *oldEnd);
        CommunicationInterface* getCommunicationInterface();
        void setCommunicationInterface(CommunicationInterface*);
        int send(NRAttrVec);
        NRAttrVec receive();
protected:
        int ID;
        int groupID;
        CommunicationInterface *dataSend;
        SensorObject *thisSensor;
        int status;
        int linkCounter;
}
```

Table 3.4 Representation of connector as a Connector object

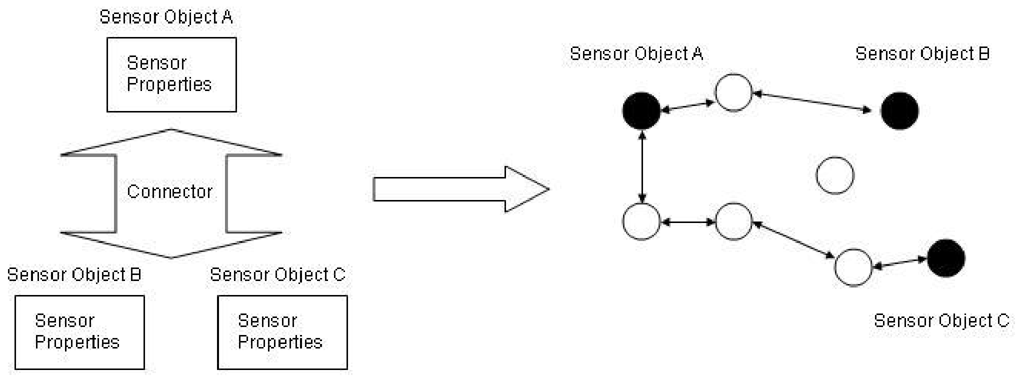

Figure 3.2 Relationship between Connector and Link

Links can be destroyed or additional links can be added to a connector if the situation so demands. For example, a sensor node may fail, prompting the removal of the link from the connector. The adaptation service may then be queried to find a replacement node with similar properties to the unavailable node, and a new link to that node is added to the connector. Alternately, sensor nodes may be added to an existing group. Figure 3.3 illustrates addition of a new node to a pre-existing group.

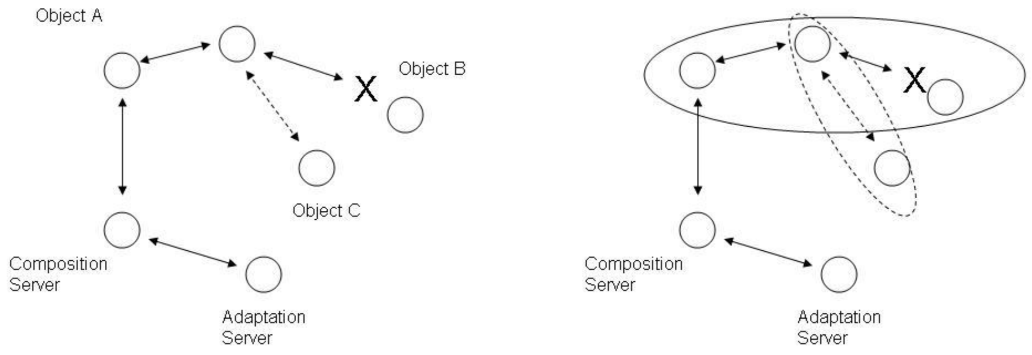

Figure 3.3 Operations on Links

Hence, connectors have the power of being reconfigurable in a transparent manner to the application, ensuring application requirements are not compromised and any service offered by the application is available to the network.

3. Associated with each connector is the *connector type* described in terms of the number of endpoints addressed by that connector. Presently we identify two types of connectors, both shown in Figure 3.4.

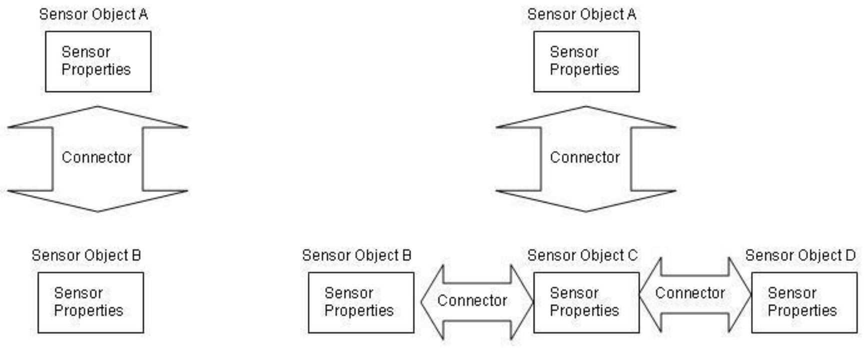

Figure 3.4 Two Types of Connectors

a. One to One – This type of connector has only two endpoints; the sink and the source.

b. One to N – A connector of this type has multicast behavior; data is communicated to all the endpoints in the connector.

In comparison with traditional IP-based networks, we can say that a One to One connector behaves similar to a TCP socket while the One to N connector is like a multicast socket.

4. At the heart of the connector is the *Communication Interface*, which interfaces the connector with the routing protocol. The communication interface also provides mechanisms for receiving and sending data which are used by the connector; the application does not interact with the communication interface. In this manner, the routing protocol can be changed without propagating changes to the application layer and affecting the behavior of the connector. Communication interface is of three types:

a. RPC – Provides the synchronous behavior associated with a remote procedure call. A call to send does not return till some form of acknowledgement is received and a call to receive does not return until data is received from the routing layer.

b. Multicast - Data is sent to all the parties in the group

c. P2P – A Peer to Peer type communication interface. Data is simply received from one endpoint and relayed to the other endpoint.

Though at first glance it would seem that a RPC type communication interface would be associated with a One to One connector and a Multicast communication interface with a One to N connector, it is also possible for a One to N type connector to have a RPC communication interface to multiple nodes if that is more efficient and appropriate under the circumstances.

3.5 System Architecture

The system architecture is shown in Figure 3.5. The composition service comprises the composition server which makes use of the Composition Service API to listen to incoming requests for joining or leaving a group. Applications use the Composition Service API to join or leave a group.

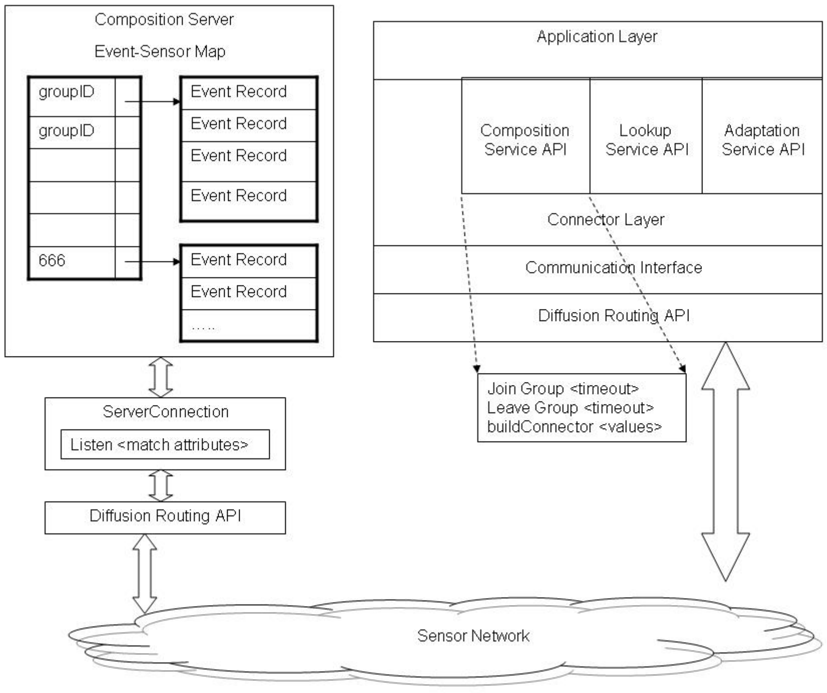

Figure 3.5 System Architecture

If two sensor objects wish to interact on a short term basis without the additional overhead associated with forming a group, they can use the Connector API directly for this purpose. As long as the attributes for the participating entities are common knowledge, both the sensor nodes can set up communication channels meant for interaction with each other.

3.5.1 The Application Layer

As the name suggests, applications supporting a variety of distributed tasks, such as collaborative clustering, target tracking and in-network aggregation can operate in the application layer. Once the concerned nodes have established the channel of communication between themselves by means of the Connectors, they can maintain data flow between themselves in accordance with the particular application’s needs. A group of such nodes can be formed by nodes in physical proximity to each other and possibly detecting the same type of event. Other nodes can join this group depending on their location, detected event type or to replace failed nodes in the original group. However, in a sensor network, nodes are deployed in an ad-hoc manner and are not aware of other nodes in the network. The task of group formation and maintaining this group formation is thus undertaken by an autonomous node which provides this service to the other nodes in the network. Service seeking nodes can avail of this service by making service calls to this particular node. The composition service is thus divided into two distinct applications; composition server and the service client.

3.5.2 The Composition Server

As discussed above, a task group consists of sensor nodes near each other and possibly detecting similar events, sharing data between themselves. The type of events detected by a node can be numerous, with different categories and of different types. For example, a node can detect an event of category “vehicle” and type “car”. Another node can detect an event of type “vehicle” but having type “tank”, and yet another can sense an “animal” of type “deer”. It is necessary to separate these events based on their categories and types.

Another factor that must be considered during group formation is the location of the nodes; communication should only be facilitated between nodes close to each other. Once the task group is formed, all participating nodes can then start communicating with each other using the group identification parameter. The composition server must thus also communicate the group information to the nodes contained in that group. There are two feasible approaches for broadcasting group information. The first approach suggests giving information about the other nodes in the group to only the group leader. It is then the responsibility of the group leader to communicate with fellow group members and inform them about their inclusion into the group. In the second approach, the server broadcasts group membership to all the nodes.

Though the first approach is more cost-effective in a resource constrained sensor network environment, a problem arises when the group leader is not able to carry out its duties for whatever reason. For this reason, it is better for all the group members to receive group membership status of all nodes in the group. This approach will also benefit reconfiguration of the group in case of node failure. New links can be established with another node having similar characteristics as the failed node.

With this in mind, the responsibilities of the composition server can be given as follows.

- Assist with group formation based on location and event characteristics. Incoming requests for group creation may result in a new group being created or the sensor node sending that request will be integrated into an existing group if all the characteristics of the `SensorObject` match the ones in the group.
- Inform the nodes in the group about their status and that of other nodes in their group.
- Maintain time-sensitive group information and inter-operate with the adaptation server for task group reconfiguration in case of node failure using this group information.

For constructing the composition server we use a hash map with the event category as the key. The value corresponding to this key is a table containing the relationship of an `Event` and `SensorObject`, collectively called `EventRecord`. The `EventRecord` object representation is done in Table 3.4.

```
Class EventRecord
{
public:
        EventRecord(Event*,SensorObject*);
        Event* getEvent();
        SensorObject* getSensorObject();
private:
        Event* event;
        SensorObject* recSensor;
}
```

Table 3.5 Representation of Event-Sensor Object relationship as EventRecord

The composition server uses a centralized approach to group composition and management in a distributed network environment. The server is more than a simple collection of `Events` and `SensorObjects`; it also manages changes in the group structure in response to changes in the real sensor network.

3.5.3 The Service Client

The service client detects an event and sends a request to the composition server to join a group using the composition service API. The reply that arrives from the server contains the group number, the ID of the sensor node that is the group leader and identification of all the other nodes in that group. The group leader then sets up a multicast channel through which group members send details of detected events to each other. This is done by constructing a `Multicast` connector to the group members. The group leader again uses the composition service API for this purpose. Once this channel is set up, an application can perform its duties and depend on the connector to provide a reliable path of interaction and mask network inconsistencies.

3.5.4 The Composition Service API

The application uses the composition service API for all the instances it has to interact with the composition server. The API masks the complexity of sending and receiving messages from the network using the appropriate routing protocol from the application.

3.5.4.1 Join/Leave a Group

The service client can use the `joinGroup()` and `leaveGroup()` methods to join and/or leave a group respectively. The prototypes of the `joinGroup()` function are as follows.

```
int joinGroup(char* serviceType, int minGroupSize, int xSpan, int
              ySpan, SensorObject* thisSensor, NRAttrVec*
              receive);
int joinGroup(char* serviceType, int minGroupSize, int xSpan, int
              ySpan, SensorObject* thisSensor, Event*
              senseEvent,NRAttrVec* receive);
int joinGroup(char *serviceType, SensorObject* thisSensor, int
              groupID, NRAttrVec* receiveAttrs);
```

Input parameters:

- `serviceType` - indicates the service the application is looking for, generally "composition". The exact value is decided by the server and can be obtained from the lookup service as part of the composition service registration details.
- `minGroupSize` – The minimum number of group members that will constitute the entire group. This number will be decided by the application as per its needs.
- `thisSensor` – A pointer to the properties of the sensor the application is running on. The composition server will keep a record of all the sensor nodes it receives the request from.
- `receiveAttrs` – A vector where the return values extracted from the reply are to be put. Return values include group ID, identity of group leader and identities of other group members.
- `xSpan` – The location bounds for the group in the horizontal direction.
- `ySpan` – The location bounds for the group in the vertical direction.

Return value:

The return value of `joinGroup()` is an integer that returns 1 or 0 depending on the success or failure of the operation.

Applications can call `leaveGroup()` to leave the current group once group interaction is not needed or if the node detects a different kind of event and wants to join another group. The prototype is as follows.

```
int leaveGroup(char* serviceType, int groupID, SensorObject
              *thisSensor, NRAttrVec* receive);
```

Input Parameters:

The input parameters have the same purposes as those for `joinGroup()`. In addition, the application needs to pass both group ID and sensor node properties to the server since the server stores a record of the sensor nodes and events seen by them in an `EventRecord` object.

Return value:

The return value of `leaveGroup()` is an integer that returns 1 or 0 depending on the success or failure of the operation.

Both `joinGroup()` and `leaveGroup()` are synchronous in the sense that they will wait until a reply arrives from the server. There are versions of `joinGroup()` and `leaveGroup()` in the API which allow the application to pass a timeout value to the function, which will force the function to return after that particular amount of time has passed.

3.5.4.2 Building a Connector

Connectors are built by calling the `buildConnector()` method and the application can then use the `Connector` interfaces to maintain a stable data flow to any other application on another sensor node. The type of `Connector` object constructed will depend on the requirement and role the application plays in the group, as defined by the role attribute explained in Section 3.3. The remote endpoints of the connector have to be known to the application before the `Connector` object can be constructed. These can be obtained from the reply sent by the server.

Alternately, if the node simply wants to contact another node with matching properties, the application can create the appropriate `SensorObject` and invoke `buildConnector()`. The prototype of the functions is as follows.

```
Connector* buildConnector(int groupID, SensorObject *thisSensor,
                          SensorObject* toSensor, NRAttrVec
                          matchAttrs);
Connector* buildConnector(int groupID, SensorObject *thisSensor,
                          SensorObject* to[], NRAttrVec
                          matchAttrs);
```

Input Parameters:

- `groupID` – the identification of the group this sensor node is a part of.
- `thisSensor` – encapsulation of the properties of this sensor.
- `toSensor/toSensor[]` – the properties of the sensor node(s) to which is connector is being built
- `matchAttrs` - only requests that match the attributes defined in `matchAttrs` are passed by the diffusion routing protocol. The application can choose to limit the broadcasting of data beyond a certain location. Nodes that were part of the group but have since possibly moved out of that location will not receive that data.

Return value:

The `buildConnector()` returns a handle to a Connector object, the type of which is decided by the `toSensor` input parameter.

3.5.4.3 Deleting a Connector

A connector can be deleted by using the `deleteConnector()` API method. The prototype is as given.

```
int deleteConnector(Connector* deleteConn);
```

Input Parameters:

- `deleteConn` – the handle to the connector to be deleted

Return value:

It returns 1 or 0 depending on success or failure respectively.

It is necessary to use `deleteConnector()` instead of using language defined ways to delete an object because certain operations were carried out in conjunction with diffusion, for example, subscribing to particular data. These matching rules will not be modified if the object is simply deleted and the application might keep on receiving old data from the routing layer.

3.5.4.4 Listening To Requests

The composition service API also provides the server with the means of listening to incoming requests for group operations. This is done via the `ServerConnection` class. This class contains only one method - the `listen()` method, the prototype of which is given below.

```
Connector* listen (char *service, NRAttrVec *matchAttrs);
```

Input Parameters:

- `service` – the service offered by this server, generally "composition". The server decides under what name it registers, however, it must provide the lookup server with details of its service.
- `matchAttrs` – only requests that match the attributes defined in `matchAttrs` are passed by the diffusion routing protocol. The server may choose to reject requests from sensor nodes that are distant from its position, as there may be another nearby composition server to handle them.

3.5.5 The Connector Layer API

The connector layer lies just below the Application layer in the system architecture. A connector has dual responsibilities. On one side, the connectors provides applications with a consistent feel of the network and maintains relationship with members of the group even if group composition changes. On the other hand, connectors also work with lower layers dealing with transport and routing.

A connector has functions for sending and receiving data from other sensor nodes. Before that happens, the connector must construct links to these other sensors. The concept of a link and the relationship between connectors and links has been explained in Section 3.4. A `Link` object encapsulates the properties of a physical link to another sensor node.

```
class Link
{
public:
        Link(SensorObject* sink, SensorObject* source, int linkID);
private:
        int linkID;
        SensorObject *sink, *source;
}
```

Table 3.6 Representation of a link as a Link object

A link has an ID which is used to identify the link and two endpoints. The first endpoint is the current sensor node and the other is the sensor node to which this link is made. `Link` objects work with the routing protocol to discover efficient paths to these other sensor nodes.

3.5.5.1 Creating/Removing a Link

An application can add an endpoint to the connector to increment the number of end points by using the `addEndPoint()` function which is a member of the connector class. The operation of adding an end point involves creating a link to the sensor node to be added. The prototype of the function is as follows.

```
int addEndPoint(SensorObject *addNew);
```

Input Parameters:

- `addNew` – a reference to the `SensorObject` to be added

Return value:

The function returns 1 or 0 depending on if the end point was added successfully.

When dealing with one to one connectors, there are simply two endpoints attached to the connector. In that case, when the connector is completely populated with links, adding a new end point is not possible. A one to N type connector is not under any such restrictions, however, the number of links created to other nodes will depend on the application. This number is usually obtained as a parameter in the reply received from the composition server.

The counterpart to `addEndPoint()` is the `removeEndPoint()` function applications can use to deconstruct a particular link.

```
int removeEndPoint(SensorObject *oldEnd);
```

Input Parameters:

`oldEnd` – a reference to the `SensorObject` to which the link is to be removed

Return value:

The function returns 1 or 0 depending on if the end point was added successfully.

For a one to one type connector, deleting an end point will simply mean reconfiguration of that connector. For a one to N type connector, this function can be called only till all the established links are deleted. Once this happens, reconfiguration of the connector will take place.

3.5.5.2 Sending/Receiving Data

Applications can use the `send()` and `receive()` functions to send and receive data from the Communication Interface layer respectively.

```
int send(NRAttrVec data);
NRAttrVec receive();
```

Input Parameters:

- `data` – the attribute vector that contains the attributes to be sent into the network.

Return values:

- The `send()` function returns a integer that specifies success or failure of the send operation.
- The `receive()` function returns an attribute vector containing the data sent by the remote sensor node represented as attributes supported by the diffusion routing protocol.

3.5.6 The Communication Interface Layer

The communication interface bears the responsibility of interacting with the routing layer to discover efficient paths to remote nodes. Sending and receiving of data or interest packets that match the diffusion protocol standards is also done at this layer.

3.5.6.1 Setting up Interests and Data Matching

Before an application can start communicating with applications on other sensor nodes, it has to send out a diffusion `Interest` specifying the type of data it is interested in. This can be done using the `setupSubscription()` function, the prototype of which is given below. This function is called by the `joinGroup()` and `leaveGroup()` functions of the composition service API.

```
void setupSubscription(NRAttrVec interest);
```

Input parameters:

- `interest` – a vector carrying the diffusion attributes that constitute the `Interest`. Other nodes will match the data they have to this interest using matching rules specified by directed diffusion.

If a sensor node produces certain type of data, it must look for interests that seek that data before the node can send out its data. For this purpose, applications can use the `setupPublication()` function.

```
void setupPublication(NRAttrVec matchAttrs);
```

Input Parameters:

- `matchAttrs` – the attributes that are matched to the incoming interest.

3.5.6.2 Sending/Receiving Data

Once interests are sent out on the network and the type of data a node offers has been advertised, data can be sent and received by using the `send()` and `receive()` functions respectively.

```
int send(NRAttrVec data);
NRAttrVec receive();
```

Input Parameters:

- `data` – the vector containing the data to be sent represented as diffusion attributes.

Return values:

- The `send()` function returns a integer that specifies success or failure of the send operation.
- The `receive()` function returns an attribute vector containing the data sent by the remote sensor node represented as attributes supported by the diffusion routing protocol.

3.5.7 Directed Diffusion Routing Protocol Layer

Fabio Silva et al. have explained the diffusion API [7] in detail. A quick overview of the diffusion routing protocol has also been provided in Section 2.1. However, the aspect of how diffusion handles data and transfers it over the network needs to be seen in order to complete the discussion about the composition service.

Data requests and responses are composed of data attributes that describe the data. Each piece of the subscription i.e. the attribute, is described by means of a key-value-operation trio, and implemented by the built-in class `Attribute`. A key indicates what the attribute stands for (temperature, speed etc.) and are simply constants (integers) that are defined in the application header. Type indicates the primitive type the key will be and will determine the type of matching algorithms run. The available types are `INT32_TYPE` (32 bit signed integer), `FLOAT32_TYPE` (32 bit), `FLOAT64_TYPE` (64 bit), `STRING_TYPE` (UTF-8 format) and `BLOB_TYPE` (un-interpreted binary data). The operator describes how the attribute will match when two attributes with the same type and key are compared. Available attributes are `IS`, `EQ` (equal), `NE` (not equal), `GT` (greater than), `GE` (greater than or equal to), `LT` (less than), `LE` (less than or equal to), `EQ_ANY`. In addition, attributes have values, which themselves have types and contents.

In diffusion, data is exchanged when there are matching subscriptions and publications and the publisher data. For example, a sensor node may publish the following set of attribute to signify detection of an 'animal' and its own position.

```
LATITUDE_KEY IS 32.67
LONGITUDE_KEY IS 35.6
TARGET_KEY IS animal
```

Any user interested in receiving data about the animals in the region might send out an interest of the following format.

```
TARGET_KEY EQ animal
```

While the above will result in data about animals being reported from all over the network, the user might want to narrow the search to a particular region. This might be accomplished by sending out an interest having the format given below.

```
TARGET_KEY EQ animal
LATITUDE_KEY GE 32
LATITUDE_KEY LE 33
LONGITUDE_KEY GE 33
LONGITUDE_KEY LE 38
```

In order to simplify the creation and manipulation of attributes, the diffusion API provides factories to create attributes. An example of the definition and creation of an attribute is as shown below.

```
#define TEMPERATURE_KEY 5050    //defines key value

NRSimpleAttributeFactory<float> TemperatureAttr (TEMPERATURE_KEY,

NRAttribute::FLOAT32_TYPE);

NRAttribute *temp = TemperatureAttr.make(NRAttribute::IS, 55.75);
```

Table 3.7 Definition and Creation of Diffusion Attributes

Since several types of data need a set of attributes for its definition, the API defines the `NRAttrVec` structure, which is a C++ STL vector of pointers to attributes. For example, the state of a region can be described by using the following parameters: temperature, humidity, latitude, longitude and time. By creating their diffusion equivalent attributes and giving them the respective values, all of them can be grouped together and sent as the data for a matching interest.

Since our composition service uses directed diffusion as the routing protocol, it makes judicious use of the built in types and factories provided by diffusion as seen in the previous sections. Though this makes the application somewhat dependent on directed diffusion, it eliminates the extra processing needed to convert application defined attributes to diffusion attributes.

```
//define key values here

TEMPERATURE_KEY 5050

HUMIDITY_KEY 5051

……

//define attributes here

NRSimpleAttributeFactory<float> TemperatureAttr (TEMPERATURE_KEY,

NRAttribute::FLOAT32_TYPE);

NRSimpleAttributeFactory<int> HumidityAttr (HUMIDITY_KEY,

NRAttribute::INT32_TYPE);

…..

NRAttrVec state;

state.push_back(TemperatureAttr.make(NRAttribute::IS,55.76));

state.push_back(HumidityAttr.make(NRAttribute::IS,65));
```

Table 3.8 Creating an NRAttrVec structure

3.6 Relationship Between Composition Service and Diffusion Attributes

As seen above, any packet traveling on a diffusion network must contain attribute-value pairs. It thus becomes necessary to convert the information needed by the application into such attribute-value pairs before sending and receiving operations are carried out. The application needs to define the required attributes using the attribute factories provided by diffusion according to its needs.

For the composition server, it is important to identify sensor nodes along with the events they sense and form groups based on this information. Group formation information then has to be conveyed back to these nodes so they can start communicating with other nodes in their group. It is thus necessary to define attributes that represent the current state of the above mentioned entities in order to support the composition service mechanism.

3.6.1 The SensorObject Attributes

The data members that constitute a `SensorObject` are shown in Section 3.3. Since the data members give the current state of the node, attributes that correspond to every member are necessary. The attributes and their types are shown in Table 3.8.

Attribute Name	**Attribute type**
SensorIDAttr	INT32_TYPE
SensorRoleAttr	INT32_TYPE
SensorStatusAttr	INT32_TYPE
LatitudeAttr	FLOAT32_TYPE
LongitudeAttr	FLOAT32_TYPE
GroupAttr	INT32_TYPE
QosPowerAttr	INT32_TYPE
QosDataRateAttr	INT32_TYPE
QosDelayAttr	INT32_TYPE

Table 3.9 Attributes for a SensorObject

3.6.2 The Event Object Attributes

The data members that constitute an `Event` object are shown in Section 3.3. The attributes and their types are shown below in Table 3.9.

Attribute Name	**Attribute Type**
EventCategoryAttr	STRING_TYPE
EventDescriptionAttr	STRING_TYPE
EventValueAttr	FLOAT32_TYPE
EventConfidenceAttr	INT32_TYPE

Table 3.10 Attributes for a Event object

3.6.3 Other Important Attributes

Other than the attributes needed for the node and event data, there are some attributes needed for the working of the composition service, listed in Table 3.10.

Attribute Name	**Attribute Type**
TargetAttr	STRING_TYPE
TaskAttr	STRING_TYPE
GroupLeaderAttr	INT32_TYPE
ServerIDAttr	INT32_TYPE
ServerGroupDetails	STRING_TYPE

Table 3.11 Other essential attributes

The use of all the attributes mentioned above will be made clear when the system implementation is discussed. The application is free to declare any other attributes as it may seem fit.

CHAPTER 4: SYSTEM IMPLEMENTATION

The previous chapter provided a wide view of the overall architecture and individual components of the composition service. Some ideas about the underlying structures needed to support an application were also presented. This chapter shall take a deeper look into the composition service API and the mechanisms that together constitute the composition service.

As can be seen from the architecture, the composition service implementation is spread across several layers. Each layer provides a set of interfaces to the layer above it. These interfaces shield the upper layer from the functional complexity encapsulated by the current layer and also allow it to invoke the desired behavior from the current layer. The responsibilities of each layer have been discussed in the previous chapter; here we will explain in detail the construction of these layers.

4.1 Implementation Issues

Chapters 2 and 3 gave the overview of the system architecture and the technologies used in connection with the composition service. Section 3.5.7 explains the Directed Diffusion data handling mechanism and the concept of diffusion attributes.

Directed diffusion is significantly different from IP-style communication where nodes are identified by the data they offer, and inter-node communication is supported by an end-to-end delivery service provided in the network. Diffusion adopts a `publish/subscribe` based API for communication. To receive data, users subscribe to a particular set of attributes, becoming data sinks. This takes the form of an `Interest` message which is broadcasted to all the neighbors. A callback function is then invoked whenever relevant data arrives at the node. Other sensors publish what data they can offer that matches the interest attributes. In both cases, what data is provided or received is distinguished by the attribute-based naming scheme described in Section 3.5.7. Diffusion also uses the attribute-based naming scheme to associate source and sinks of data. This naming scheme is data-centric, allowing applications to focus on desired data instead individual nodes. The exact process of determining which publications and subscriptions are related is called matching, which is driven by rules defined in the Directed Diffusion core program.

This has important implications for any client-server application running on top of the diffusion protocol. In an IP-based network a client can directly contact a server using only the IP address of the server and all intermediate routers know exactly where to forward the data. Since nodes are addressed by the data (or services) they offer in a diffusion network, the client has to obtain the properties of the server node and send out an interest message with those properties. The composition server can register with the lookup service [2] in order to enable the service-seeking nodes to discover it.

Secondly, the `Interest` message generated by the client will be initially flooded over the entire network. In case the composition server is at a distant location from the client(s), every intermediate node will receive and have to forward these packets, resulting in power consumption for that node. The region filter [25] can be used with directed diffusion to limit this initial flooding since the location of the server is known from the lookup service.

It also becomes necessary for any application to convert any message that needs to be passed over the network into diffusion-level attributes. All the attributes required thus need to be pre-defined, including those for group creation and management. Matching rules for diffusion, explained in Section 3.5.7, also need to be set up as per the application's desire to send or receive data. The composition service API provides methods for these purposes so that the application can concentrate on its task.

For the composition server, in addition to the above, there are other things that must be considered. Some responsibilities, like that of the composition server, are well defined and easy to implement. The composition server needs to simply accept any interest packet from other nodes which contain information about these nodes and are for the specific purposes of group activities. Hence the server is just required to set up a matching rule with local scope for the above purpose. The responsibility of a composition client however, is multifold. Initially, the client broadcasts `Interest` messages with the intent on joining or creating a group. The server replies to the client's request with group information. The setting up of connectors in the group depends on the application. Accordingly, a group member can either produce data, offer in-group services, or be the data consuming end point of the group. This translates to setting up different attributes to be matched for each node and possibly for different types of network traffic. Though the composition service programming constructs provide the means for the application to define its needs, the application will still need to take into account its requirements.

4.2 Implementing the Composition Server

The composition server listens for interest messages from event sensing nodes using the `ServerConnection listen()` function. The server also publishes its own attributes to be included when a reply to a join group or leave group request is to be sent. The `listen()` function returns a `OneToOne` type connector through which the server sends a reply to each of the nodes from which it received a matching interest. For all its essential operations, the server defines its own internal (private) member functions and uses them extensively. The sequence of events for joining and leaving groups and associated mechanisms needed to send a reply are explained below.

1. On receiving a join group request the server first checks for the presence of a pre-existing group whose members have matching properties such as location to the node from which the request was sent. This step is necessary to prevent a multitude of new groups being formed for the same purpose.
2. A group is created between nodes that are physically close to each other. The server matches the latitude and longitude of the events for all the nodes involved. If such a group is found, the reply is sent to the new node with the group ID and group members. If a group does not exist, a new group is created at the server.
3. The server employs a structure for storing the properties of the nodes that must be indexed by group identity for easy retrieval of group member information. It does this by means of `Hash Map`, the structure of which is as shown in Figure 4.1.

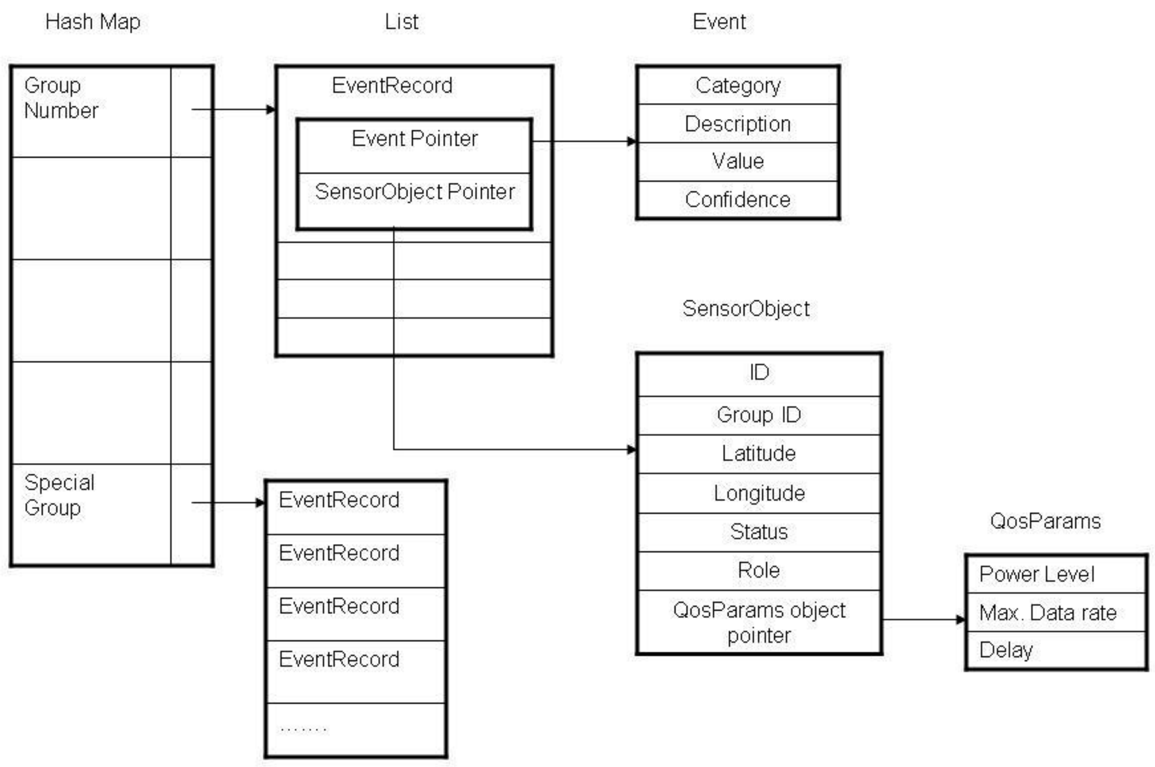

Figure 4.1 Server storage structure

4. As can be seen from Figure 4.1, all incoming requests are grouped together into a special group. This group maps to a list of `EventRecord` objects, explained in Section 3.5.2. This list will be scanned for all nodes that fall within the limits of a prescribed area and do not already belong to a group.
5. When the server receives a request to join a particular group, it checks for the presence of a group ID in the message. If a group ID is found and such a group exists, the sensor node is added to that group. If there is no group ID, the server attempts to add it to the list for the special group number. This can be done by using the `insertEntry()` server member method.

```
int insertEntry(EventRecord* currentRec ,int xSpan ,int ySpan,
                    list<EventRecord*> * matchEntries)
```

Input Parameters:

- `currentRec` – the `EventRecord` of the incoming sensor and `Event` sensed (if present)
- `xSpan` – the area to be scanned for matching nodes in the horizontal direction
- `ySpan` – the area to be scanned for matching nodes in the vertical direction
- `matchEntries` – the list of all nodes that match the current node the request was received from.

Return Value:

`insertEntry()` returns an integer specifying the number of nodes of similar type that exist in the list.

The return value is compared by the server to the maximum number of nodes value supplied by the application to the server through the `joinGroup()`function. If such a value is not specified by the application, a default value of 4 is used by the server.

Before inserting an `EventRecord` object in the storage, the `insertEntry()` function checks if the same record already exists in the storage since the current request may be a duplicate one. Duplicate requests can occur due to the inherent broadcast nature of directed diffusion or because the originating node sent out more than one interest messages. A check for the existence of duplicate information can be done using the server member `contains()` method. This method will use the sensor ID to distinguish one sensor node from another.

```
int Server::contains(EventRecord* currentRec)
```

Input Parameters:

Same as those for `insertEntry()`.

Return Value:

1 or 0 signifying the presence or absence of record in storage respectively.

6. Once a sufficient number of similar nodes have been obtained from the special group list, the server will then form a group and assign a group number to the concerned nodes. The exact span of the area to be considered depends on the server; currently we include all nodes that lie within 50 units (for example, 50 meters) of the present node. The server uses the member `formGroup()` function for this purpose. The group number and list of nodes in the group will be entered into the server storage.

```
void formGroup (int groupID, list <EventRecord*> groupEntries)
```

Input Parameters:

- `groupID` – the new group number
- `groupEntries` – the `EventRecord` list of all nodes in the group

7. The server then composes a reply using the attributes defined by the application and sends the reply to all the nodes in the newly formed group. The exact attributes used and their purpose are given in Table 4.1.

Attribute Name	**Type**	**Purpose**
ServerIDAttr	INT32_TYPE	The identity of the server.
GroupAttr	INT32_TYPE	The group identification number.
GroupLeaderAttr	INT32_TYPE	The identity of the node that is the group leader for this group.
TaskAttr	STRING_TYPE	Specifies the reason for sending the reply; “join group reply” in this case.
ServerGroupDetails	STRING_TYPE	The identities of all the other nodes in the group.

Table 4.1 Join group reply attributes and their purpose

8. The reply attributes are packed into a `NRAttrVec` vector and passed to the connector `send()` function which sends the packet using the functions provided by the underlying `CommunicationInterface` object.
9. If a leave group request is received, the node in question is obtained from the map and deleted by using the member `removeEntry()` function.

```
int removeEntry(int groupID,EventRecord *existEntry)
```

Input Parameters:

- `groupID` – the group number the node was a part of
- `existEntry` – the `EventRecord` corresponding to the sensor node to be deleted

Return Value:

It returns 1 or 0 depending on the success or failure of the remove operation.

10. The server then composes a leave group reply addressed to the node from which it received the request. The attributes used are shown below in Table 4.2.

Attribute Name	**Type**	**Purpose**
ServerIDAttr	INT32_TYPE	The identity of the server.
TaskAttr	STRING_TYPE	Specifies the reason for sending the reply; “leave group reply” in this case.

Table 4.2 Leave group reply attributes and their purpose

The server API is also used to maintain or reconfigure group information when nodes are unable to communicate due to failure or when their power runs out. The composition server will then work with the adaptation server to discover new nodes and add them to pre-existing groups.

4.3 The Composition Service API

An overview of the composition service API functions was given in Section 3.5.4. The main goal of the API is to make the task of communicating with the server as seamless for the application as possible. The API also aims to use the underlying connector interface and set up a stable communication channel to the server. This channel is to be maintained for as long as the application needs to interact with the server. In the context of directed diffusion, efforts must be made to minimize flooding of interest messages when discovering a reliable path to the composition server and other nodes in the group. The packets sent over the network also have to be as small as possible, as radio transmission is the single most expensive operation on a sensor node [4,19].

4.3.1 Joining a Group

The `joinGroup()` API function undertakes the task of translating the object references sent by the application into diffusion attributes and sending the appropriate interest message over the sensor network. The prototype for the function is given below.

```
int joinGroup(char* serviceType, int maxGroupSize, int xSpan, int
              ySpan, SensorObject* targetSensor, SensorObject*
              thisSensor, NRAttrVec* receive);
int joinGroup(char* serviceType, int maxGroupSize, int xSpan, int
              ySpan, SensorObject* targetSensor, SensorObject*
              thisSensor, Event* senseEvent,NRAttrVec* receive);
int joinGroup(char *serviceType, SensorObject* thisSensor, int
              groupID, NRAttrVec* receiveAttrs);
```

The semantics for the `joinGroup()` API function were explained in Section 3.5.4.1. The function uses member methods from the `SensorObject` and `Event` classes; these methods are highlighted in boldface in the working description given.

1. The application calls `joinGroup()` passing in the service to be invoked, the references to the calling and target sensor node objects along with a reference to a

`NRAttrVec` where the return values are to be stored. A version of the function also allows for passing the `Event` information to the server if the application and sensor node are able to accurately categorize the sensed event.

2. A `NRAttrVec` structure is created which holds the attributes to be sent out with the interest message. The contents of this vector are given in Table 4.3.

Attribute Name	**Type**	**Purpose**
NRClassAttr	INT32_TYPE	Specifies the type of the message to be sent is an interest Message. The value is thus NRAttribute::INTEREST_TYPE.
TargetAttr	STRING_TYPE	The service to be found. The value is usually “composition” unless composition server registers with the lookup server using some other value.
TaskAttr	STRING_TYPE	The task for which this interest message is being created, in this case, “join group”
GroupAttr	INT32_TYPE	The group to which node belongs
XSpanAttr	INT32_TYPE	The span in the horizontal direction
YSpanAttr	INT32_TYPE	The span in the vertical direction

Table 4.3 Join Group Attributes

The message also contains the details of the node on which the application is running and the event which triggered the sequence of events. The `joinGroup()` function can get these properties from the `getSensorAttributes()` member function of the `SensorObject` class and the `getEventAttributes()` function of the `Event` class. Both these functions return a `NRAttrVec` containing the which is then copied into the send vector using the `AddAttrs()` diffusion API function.

3. The application then calls the `setupSubscription()` method of the communication interface which belongs to the member `Connector` object of the API, passing in the entire `NRAttrVec` vector containing all the attributes to be sent. The `setupSubscription()` method will send out an interest message on the network.
4. The function then calls the `receive()` function of the `OneToOne` connector class to receive matching data for the interest sent. The `receive()` function queries the member communication interface which in turn queries incoming packet queue set up between the communication interface layer and the routing layer and returns the packet on top of the queue. The operation is a blocking one, i.e. the function does not return until a packet is received from the diffusion routing layer.
5. When `joinGroup()` receives data, it checks the data to ensure the packet is a join group reply from the server and is meant for the current node. This is necessary as a node might receive join group replies meant for some other node in another unrelated group. Until both of the above match, the received packet is discarded, `receive()` is called and the checking process continues.

6. Once a valid packet is obtained, the `joinGroup()` function then copies all the attributes from the packet into the `receiveAttrs` vector passed as an argument to it. The control will now pass to the application which has the join group reply from the server and all the necessary attributes it needs.

4.3.2 Leaving a Group

The `leaveGroup()` API function allows the application to convey its intention of disassociating itself from a group to the server. There may be many reasons why a node might want to leave a group; it wants to join another group, the event data that prompted group creation is no longer being sensed etc. In any case, an interest message must be created and sent to the server requesting a node delete operation at its end.

```
int leaveGroup(char* serviceType, int groupID, SensorObject
               *thisSensor, NRAttrVec* receive);
```

The input parameters and return values for `leaveGroup()` were presented in Section 3.5.4.1. The mechanism is described below.

1. Since the server uses a group-node association to store records in the map, we need to pass the group ID and the sensor node ID to the server in order to enable it to make a match.
2. An `NRAttrVec` vector is used to hold the attributes that make up the interest message to be sent over the network to the composition server. The attributes utilized in creating this message are given in Table 4.4.

Attribute	**Type**	**Value**
NRClassAttr	INT32_TYPE	Specifies the type of the message to be sent is an interest Message. The value is thus NRAttribute::INTEREST_TYPE.
TargetAttr	STRING_TYPE	The service to be found. The value is usually “composition” unless composition server registers with the lookup server using some other value.
TaskAttr	STRING_TYPE	The task for which this interest message is being created. The value in this case will be “leave group”
GroupAttr	INT32_TYPE	The group to which this sensor belongs to.
SensorIDAttr	INT32_TYPE	The ID of this sensor which wants to leave the group.

Table 4.4 Leave Group attributes

3. Once all the attributes are created and entered using the `push_back()` vector member function, `leaveGroup()` calls the `setupSubscription()` member function of the `OneToOne` type connector which will send the interest package in a similar way to the `joinGroup()` function.
4. When `leaveGroup()` receives data, it checks the data to ensure the packet is a join group reply from the server and is meant for the current node. This is necessary as a

node might receive join group replies meant for some other node in another unrelated group. Until both of the above match, the received packet is discarded, `receive()` is called and the checking process continues.

5. Once a valid packet is obtained, the `leaveGroup()` function then copies all the attributes from the packet into the `receiveAttrs` vector passed as an argument to it. The control will now pass to the application which has the leave group reply from the server. The application then must either delete the `Connector` object (see Section 3.5.4.3) or reconfiguring it using the `Connector` API.

4.4 Listening to Incoming Requests

There can be a multitude of services offered by nodes in the sensor network environment like the adaptation service, lookup service and so on. The server needs a way of filtering out requests meant for its offered service type and respond to these requests. The `ServerConnection` class implements the `listen()` function for this purpose.

```
Connector* listen (char *service, NRAttrVec *matchAttrs);
```

The input parameters and return value were explained in Section 3.5.4.4. The mechanism is described below.

1. Like the `joinGroup()` and `leaveGroup()` API functions, the server expresses a desire for receiving packets that match a particular criteria. However, unlike both those operations, this match is not required to travel out into the network as an

interest packet. Rather, it is kept local to the node and every packet received is matched to the attributes specified. On a match, a `Connector` object reference is passed to the application.

2. An `NRAttrVec` vector is used to hold the attributes that are to be matched to the attributes in the incoming packet. In addition, the application can also pass additional attributes to the function for matching purposes. attribute name, types and values, if any, are described in Table 4.5.

Attribute	**Type**	**Value**
NRClassAttr	INT32_TYPE	Specifies the type of the message to be received is an interest Message. The value is thus anything that is not NRAttribute::DATA_CLASS
NRScopeAttr	INT32_TYPE	The interest expressed is to be kept local to the node. The value is NRAttribute::NODE_LOCAL_SCOPE
TargetAttr	STRING_TYPE	The service to be offered. The value is usually “composition” unless composition server registers with the lookup server using some other value.

Table 4.5 Listen attributes

In this case, the scope for the interest is to be given explicitly, since the default scope used by directed diffusion is the global scope which will propagate the interest over the network. Since this was the intention for `joinGroup()` and `leaveGroup()`, the `NRScopeAttr` is not required to be included in the packets they send out.

3. All the above attributes and the extra attributes specified by the application are copied into a `NRAttrVec` structure after which the subscription is set up in the same way as the `joinGroup()` and `leaveGroup()` functions. The connector reference is returned to the application which can be used to always receive messages from the service-seeking nodes.

4.5 The Connector API

The function and purpose of a connector has been explained in Section 3.4. In this section we present the construction of the connector interface utilized by the composition service API and applications. The creation of a `Connector` object has been explained in Section 3.5.4.2.

4.5.1 Sending and Receiving Data

The `send()` and `receive()` functions were presented in Section 3.5.5.2. Both functions call the `send()`and `receive()` of the member communication interface respectively.

4.5.2 Adding and Deleting Endpoints

A connector provides the application with the means of communicating with other sensor nodes via a reliable channel. The node the application is running on and other nodes constitute the endpoints. The connector API provides the `addEndPoint()` and `removeEndPoint()` functions for manipulating the end points of the connector.

```
int addEndPoint(SensorObject *addNew);
int removeEndPoint(SensorObject *oldEnd);
```

Input Parameters:

- `addNew` – the sensor object to which a connection is to be made
- `oldEnd` - the sensor object to which the established connection is to be broken

Return Parameters:

Both functions return 1 or 0 based on the success or failure of the operation.

The exact implementation of these functions may differ for different types of connectors. As explained in Section 3.4, the two types of connectors are `OneToOne` and `OneToN`. An `OneToOne` connector represents a connection between just two endpoints, hence it is not possible to add an endpoint without removing the existing remote endpoint first. The `OneToOne` type connector thus keeps track of the number of active endpoints and imposes a maximum limit of two endpoints. On the other hand, an `OneToN` type connector can have a number of remote endpoints, the exact number of which depends on the application. A `OneToN` connector needs a data structure for tracking all the active nodes it presently has communications channels set up to.

When all other conditions are satisfied, the `addEndPoint()` operation results in the creation of a new `Link` object. An internal link counter is incremented, which is used by the connector to report the number of active links it presently contains. Conversely, the `removeEndPoint()` operation results in the appropriate `Link` object being deleted. Both operations have been illustrated in Section 3.4.

4.5.3 Redirecting a Connector

The task of redirection or reconfiguration of a connector is performed in the case of failure of one or more nodes in the group. The composition server works with the adaptation server to discover operational nodes within the vicinity the original group members and replaces the failed node. This entire operation is transparent to the application which only sees and uses the connector resource for its communication purposes.

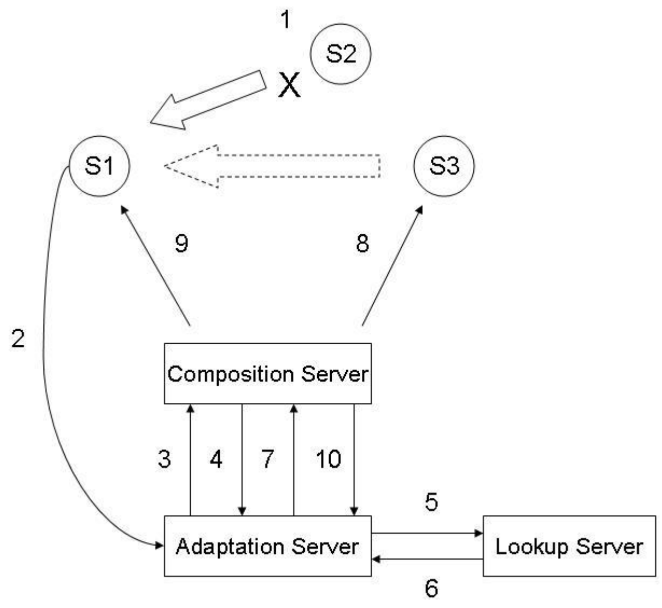

Figure 4.2 Redirecting a Connector

All the nodes in the group (S1 and S2) already have connectors set up to each other. In Figure 4.2 we have shown just two nodes to be part of the group for the sake of clarity. In reality there could be up to six nodes in the group. The following is a sequence of actions for managing node failure and reconfiguration.

1. One of the nodes in the group goes down, which is detected by the other node(s) receiving data from this node.
2. These ‘downstream’ nodes send a node failure message to the adaptation server.
3. The adaptation server communicates with the composition server to get information about the task structure of the group to determine the possible failed upstream node.
4. The composition server returns the structure of the group.
5. The adaptation server will query the lookup server to obtain information about all nodes that match location criteria and offer a similar service.
6. The lookup server returns a list of possible replacement nodes to adaptation server.
7. The adaptation server will choose the most appropriate node as replacement for failed node. It will also initiate the operation for the new node to replace the failed node. This decision will be conveyed to the composition server.
8. The composition server shall make the requisite changes in its local database.
9. Composition server reconfigures the endpoints of the connectors of all concerned nodes.
10. Upon successful reconfiguration, the composition server returns the status to the adaptation server.

At first glance, it looks like the replacement node can be chosen by the composition server instead of the adaptation server, minimizing communication overhead between these two services. However, the main purpose of the composition server is to form groups based on sensor location. The algorithm of which node is to be designated as the replacement should be best implemented by the adaptation server.

CHAPTER 5: PERFORMANCE EVALUATION

The previous chapters covered the thought process, architecture and implementation of the composition server. This chapter attempts to demonstrate the benefits of the composition service in real-world scenarios by considering two different areas; mobile target tracking and automatic failure recovery. Applications concerned with both areas need to be independent from the effects of sensor node mobility and the unreliability of sensor network environment.

Most of the applications designed for sensor networks require some form of sensor clustering to achieve a degree of accuracy in reporting the sensed data. This chapter presents the performance of a target tracking application which uses our composition service API extensively.

A sensor node in an ad-hoc sensor network is also subject to failure from a variety of reasons like hostile environmental conditions or drop in battery power level of the node. Since every node contributes in routing decisions, at the very least node failure may result in loss of data for data-seeking nodes. The situation is exacerbated if the faulty node was a service offering node; some way for dynamically obtaining similar nodes and replacing the faulty node transparently must be devised. The composition server can work with the adaptation server to discover replacements and initiate them into the set up.

To optimize the recovery process, the adaptation server implements some constraint satisfaction algorithms with the help of the composition server [25].

5.1 Dynamic Sensor Clustering

Richard Brooks gives a high level view of a multi-sensor tracking system [21], also shown below in Figure 5.1.

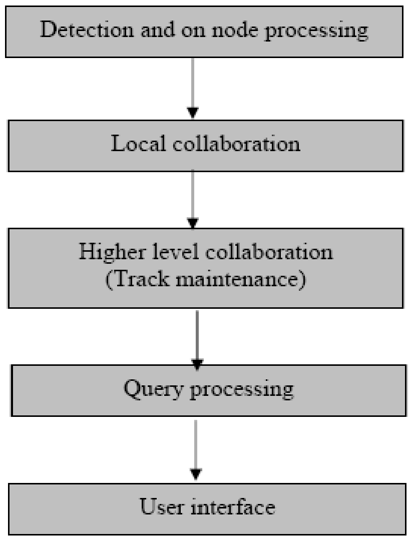
Detection and on node processing
Local collaboration
Higher level collaboration (Track maintenance)
Query processing
User interface

Figure 5.1 High level view of a target tracking system

Each sensor node continuously monitors its environment and tries to detect events as they occur within the sensor's field of operation. Target information is then used by the node to create a detection event by matching it with the node's known target type database. Local collaboration is performed to create a accurate categorization of the target, and includes only nodes within a dynamically determined geographic neighborhood and time frame. Results of this local collaboration are then used to initiate and maintain tracking of the target. Tracking information may also be stored in a distributed database which is possibly part of a complex query processing system and is tied to a user interface.

As can be deduced, detection and in-node processing is the first step for target tracking. Robustness and reliability are also important factors one must consider in this process of sensor fusion. In this context, the number of sensor failures the network can tolerate becomes crucial, as is the manner in which data from fit sensors is separated from the unfit ones. Richard Brooks, et al, give a solution [22] that satisfies the requirements of inexact agreement problem by merging the sensor fusion algorithm with Mahaney and Schneider's Fast Convergence algorithm [23]. The solution assumes the self-organization of sensor nodes into clusters. The sensor fusion algorithm runs on the cluster head, which collects the processed data from the group members and inputs this into the algorithm.

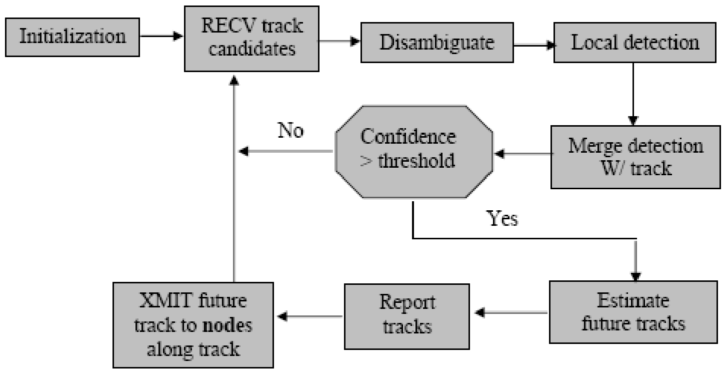

Figure 5.2 Target tracking process flow chart

A flowchart for the process of target tracking is shown in Figure 5.2. The initialization phase consists of *publish-subscribe* calls. The methodology in [22] proposes two waves of publish-subscribe calls propagated in four directions.

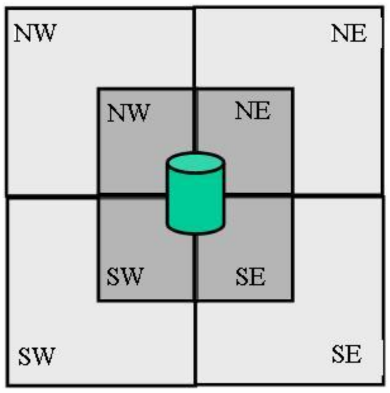

Figure 5.3 Regions in which candidate tracks are published

Figure 5.3 illustrates the *near* (dark grey) and *far* regions for a node, where both terms differ in the distance between the node broadcasting track information and the node receiving this information. Each node sends subscription messages to all nodes in the eight regions. If a node detects a target in the north-west direction, it will transmit tracking information to near/far nodes with subscriptions in the north-west area. Nodes in this region can receive multiple candidate tracks from a multitude of nodes. Local collaboration then determines the exact position of the target in the vicinity of a nearby cluster. Candidate tracks that are inconsistent are filtered out and promising tracks are retained. A Euclidean metric is then used to determine the best-fit track. Merging the new detection with track is done using an Extended Kalman filter [24]. A rectangular region is constructed enclosing the places the target is likely to visit in a short time period.

The above concepts were successfully applied to a multi-target, multi-sensor tracking application set up at the 29 Palms Base in Southern California [21]. However, if the network starts to become large-scale (more than 1000 nodes) with the directed diffusion routing protocol, some problems are apparent.

The most obvious problem is packet flooding. An interest packet is broadcasted over the entire network for the sink to find some source node with the requested data. For the scheme illustrated in Figure 5.3, each node needs to send out eight interests, each corresponding to a different region. Since each node has the ability to subscribe to needed data, the total number of packets can be least approximated by multiplying these eight interests by the number of nodes in the network.

Scalability of a diffusion network is also restricted by the size of the gradient table of each node. Gradients directly correspond to the number of distinct interest messages received by the node. Greater the number of such messages, greater is the gradient table, which might eventually exceed the capacity of the node. It must also be noted that transmission of interest messages is also done as reinforcements to maintain established paths of communication. As long as tracking collaboration exists in the network, interest packets will be re-broadcasted after a certain interval.

For a sensor node, the expense incurred for communication is several orders more than for computation [4]. Reducing the number of messages transmitted and received thus goes a long way in increasing the longevity of the entire network. Equally important is to limit the number of nodes traversed by interest messages; more the number of hops, more the nodes that spend energy in re-transmission. All packets that are exchanged between nodes for collaboration should be limited within one or several clusters instead of propagating across the entire network. Subscription rules should also be as specific as possible to enable finding the appropriate nodes within the specified region. The region filter developed at Auburn University [12] as an enhancement to directed diffusion works to contain message re-transmission within a specific region. A detailed explanation of the mechanism is given in [25].

5.1.1 Evaluation for thirty nodes with multiple groups

This section will give the performance of the composition service mechanism using diffusion broadcasting techniques over the sensor network. The metrics used are the average number of interest and data packets in the network relative to the number of groups and network size. The data is collected over a series of time intervals. The testing environment provided by ISEE [12] was utilized and results are obtained using the metric filter. The region filter [25] has been used to limit the flooding of packets in the network. The number of messages includes the interest messages sent from the clients to the composition server and the data messages sent as the reply from the server to the clients.

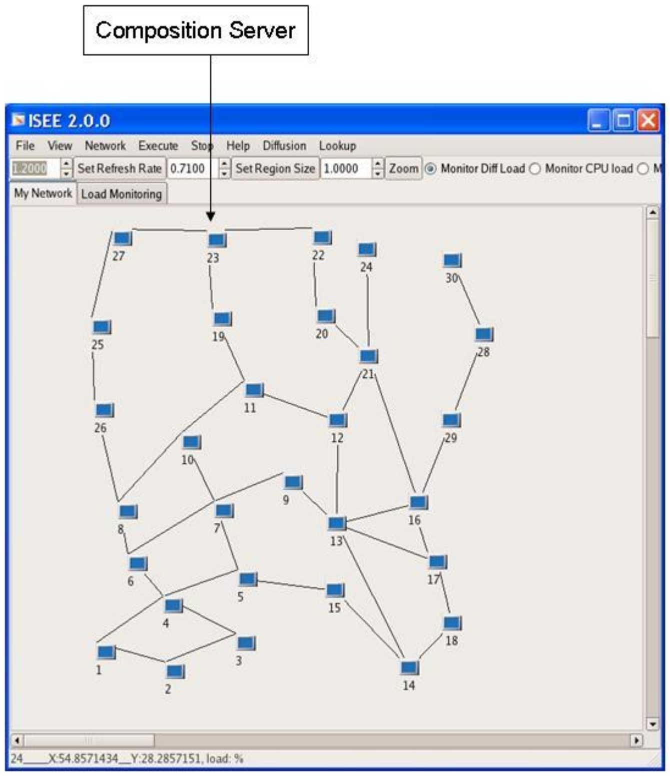
Composition Server
ISEE 2.0.0
File View Network Execute Stop Help Diffusion Lookup
Set Refresh Rate 0.7100 Set Region Size 1.0000 Zoom Monitor Diff Load Monitor CPU load
My Network Load Monitoring
27 23 22 24 30 25 19 20 21 28 11 12 26 29 10 9 16 8 7 13 17 6 5 15 4 18 3 1 2 14
24___X:54.8571434__Y:28.2857151, load: %

Figure 5.4 Network topology in ISEE

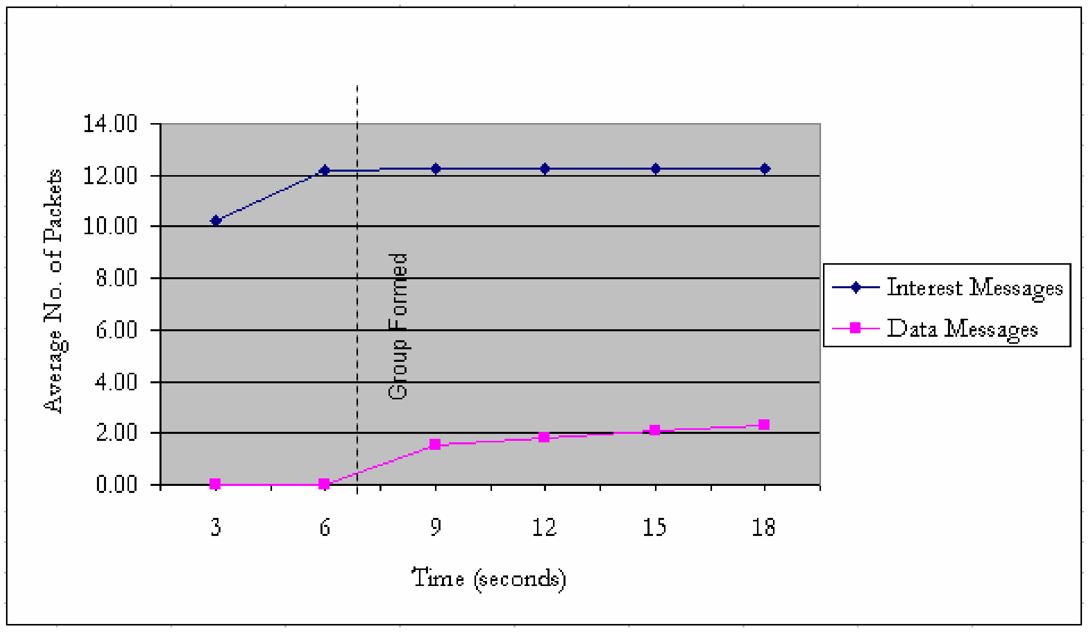

Figure 5.5 Interest and Data messages for 30 nodes with 1 group

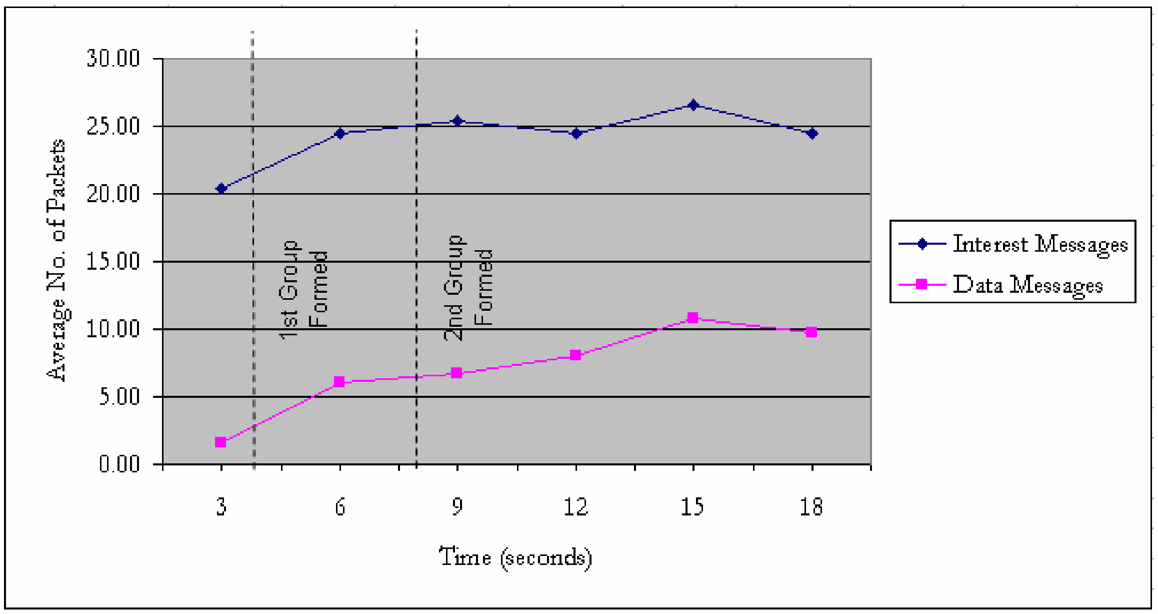

Figure 5.6 Interest and Data messages for 30 nodes with 2 groups

As seen from Figure 5.5 and Figure 5.6, interest packets are flooded into the network initially as the nodes contact the composition server for joining a group, while there is virtually no data packet traffic. As the nodes are grouped by the server and start communicating between themselves, the data packet traffic increases as the average interest packets gradually remain the same or start decreasing as negative reinforcements are sent by group members using the composition service API. The number of data packets will actually depend on the application; our test application sends 10 data messages between each other. Other applications may use the set up to collaborate for a far longer time.

5.2 Automatic Failure Recovery

The previous section presented an application that required node collaboration in a sensor network and the performance of the application with respect to directed diffusion. However, nodes in a sensor network are susceptible to failure and/or mobility, introducing the element of unpredictability into the set up. Irrespective of whether the failed node is a service-providing or service-seeking node, the task at hand will be interrupted in case of a node failure. The network is expected to adapt and survive these conditions, recovering by replacing failed nodes with others suitable to take its place. In a large network there might be many such nodes providing a similar kinds of service and registered with the lookup server or composition server.

Consider a target tracking scenario illustrated in Figure 5.7. Nodes 1, 2, 3, 4 and 5 form a group and exchange data between themselves. The solid lines indicate the data flow between the nodes. Nodes 1 and 2 sense the raw data (such as images) and forward it to Node 3 which provides a data filtering service before forwarding to Node 4, a data caching service. Node 5 is entrusted with the responsibility of implementing data transformation for improving transmission efficiency. It also assembles the data fragments into a complete image before forwarding it to a base station. Node 4 is the faulty node which fails when the connectors have been set up and stable data flows are established.

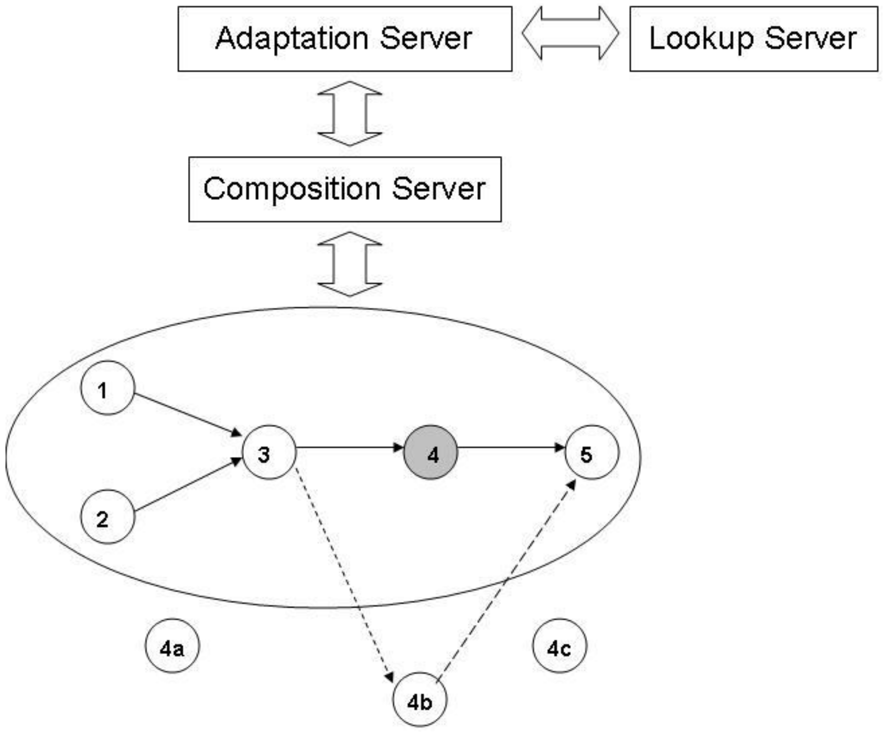

Figure 5.7 Service Node Replacement

Some degree of adaptability to node failure is in-built in Directed Diffusion. Every sink node has a positive gradient towards its stable data-sending neighbor in the gradient table. When data stops arriving at the sink Node 5, as will happen when Node 4 goes down, directed diffusion will respond by flooding similar interest packets in the whole network to try and find other sources of data. Other source nodes will respond by sending matching data which will eventually reach the sink node. The sink node will then send out a positive reinforcement to one source after which the source will send data at the requested rate. This basic form of adaptability is adequate for simple query-response systems but not sufficient for complex applications. The time required to find a replacement is one deterrent factor; the physical location of the replacement is another.

The new node may be in a distant location to the group, increasing the number of hops the data has to go through, resulting in energy expenditure of a greater number of intermediate nodes The replacement node must also have matching characteristics to the one it is going to replace. Additionally, the task of reconfiguring the upstream nodes such as 1 and 2 above is not addressed by this scheme. A faster and better replacement method can be implemented by using an adaptation server. The goal of the adaptation process is to achieve highly efficient reconfiguration which may extend the lifetime of the task and save overall group energy.

5.2.1 Automatic Failure Recovery with Adaptation and Composition Services

The adaptation server will need some basic meta-data about the sensor nodes in order to make a decision about replacing a failed node with a suitable replacement. There are two approaches to gathering this information. The adaptation server can extract sensor node data from all the distinct interest packets it sees and use a local repository to store and access this information. This approach’s attractiveness lies in its ability to minimize communication overhead for adaptation purposes. However, a problem arises if the adaptation server is at a far off location from the failed nodes. In that case, the server might not see all the packets sent by the concerned nodes and has to resort to the random method for obtaining the replacements. The other way is for the adaptation server to access some repository that maintains sensor node information like service provided, or group details and so on. Such information is already maintained by the lookup and composition servers. The adaptation server will thus need some mechanism to access the stores of these other services and obtain the needed information, indicating a degree of communication and data exchange between two services. Section 4.4.3 explained the procedure of this interaction. In this chapter we are concerned with the cost-effectiveness of that method.

5.2.2 Performance Evaluation

We implemented a basic application described in the scenario presented in the previous section on a sensor network with 30 nodes. The data is collected by implementing groups with 6 nodes in the group and measured over time intervals. The metrics used for the comparison are the time it takes for the node replacement, called *Recovery Time*, and the increase in the average number of packets in the network.

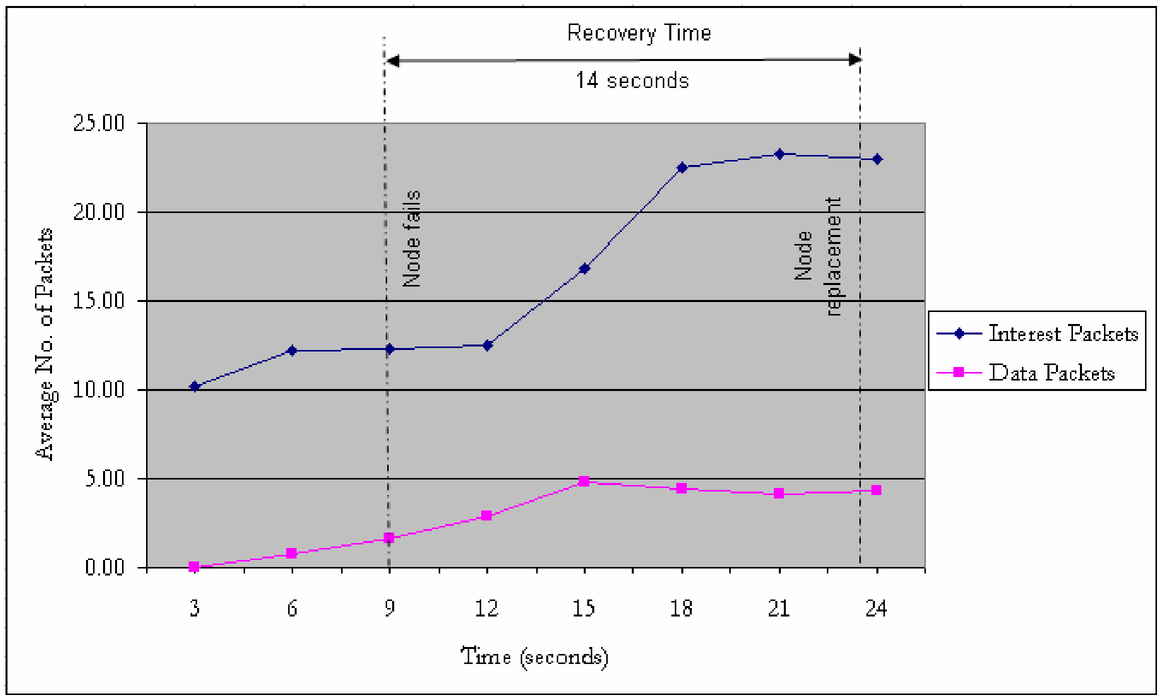

Figure 5.8 Adaptation using Directed Diffusion for one group

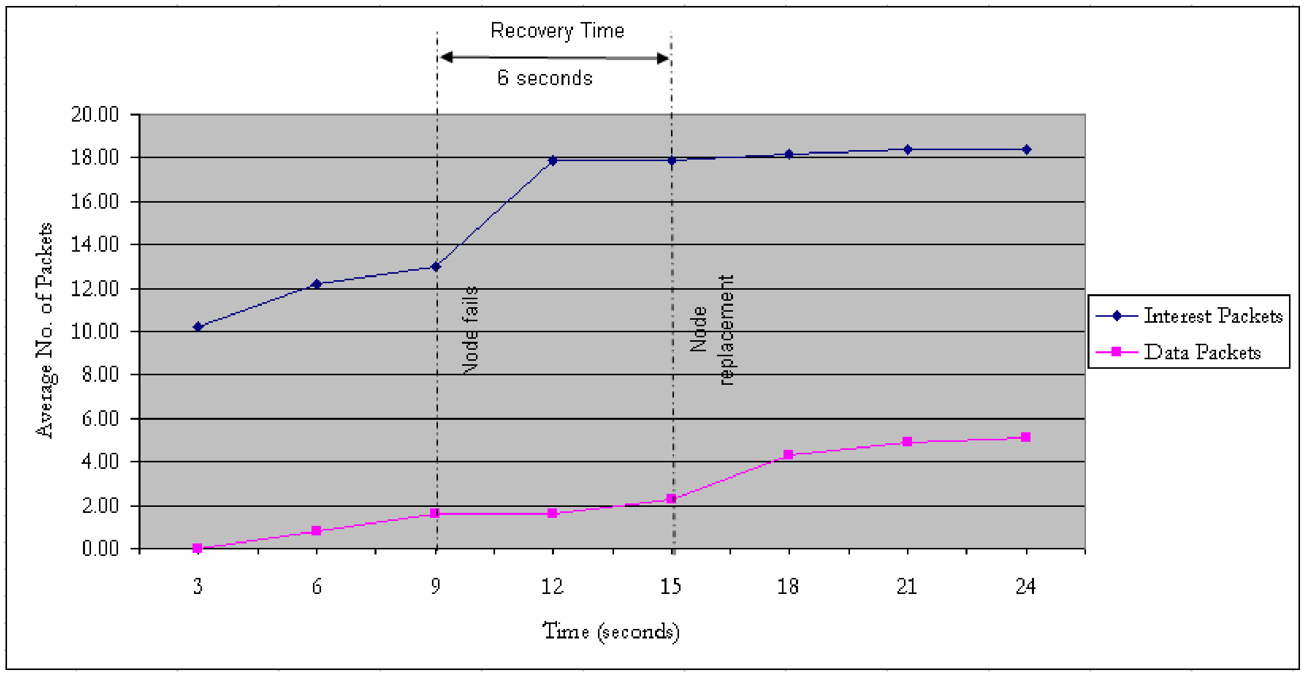

Figure 5.9 Adaptation using Composition server and Adaptation server for one group

As seen in Figure 5.8, after the node fails, there is a dramatic increase in the average number of interest packets in the network as diffusion tries to cope with the loss of data. Data packets are still arriving at the sink node but at a slower rate. The average number of data packets then drops as the source is simply not sending any new data. A new source is available, albeit at an unknown location which may be far-off from the other sink nodes. The recovery process is thus a slow one, as can be gauged from the lengthy recovery time. The application suffers from data loss for a relatively long time.

Comparatively, Figure 5.9 shows the adaptation process using the Adaptation and Composition servers. After the upstream node fails, the downstream node sends a failure indication to the adaptation server with the failed node characteristics, causing an increase in the number of interest packets, while the data rate suffers. The adaptation server then contacts the lookup server which responds with all candidate nodes in the vicinity of the other group members. The adaptation server chooses a suitable node and informs the composition server with its decision. All these tasks cause an increase in the number of interest packets. The composition server then sends an indication to the replacement node, which takes over the task of the failed node, explaining the resumption of the data packet flow. The system recovery time for directed diffusion is 14 seconds as compared to 6 seconds with the addition of the adaptation server; a significant difference. The distributed services combined with the region filter also result in controlled flooding of exploratory packets, as can be seen from the lower interest packet level in Figure 5.9.

The test results for the adaptation process with 3 groups with a sensor network of 30 nodes are illustrated in Figure 5.10 and Figure 5.11.

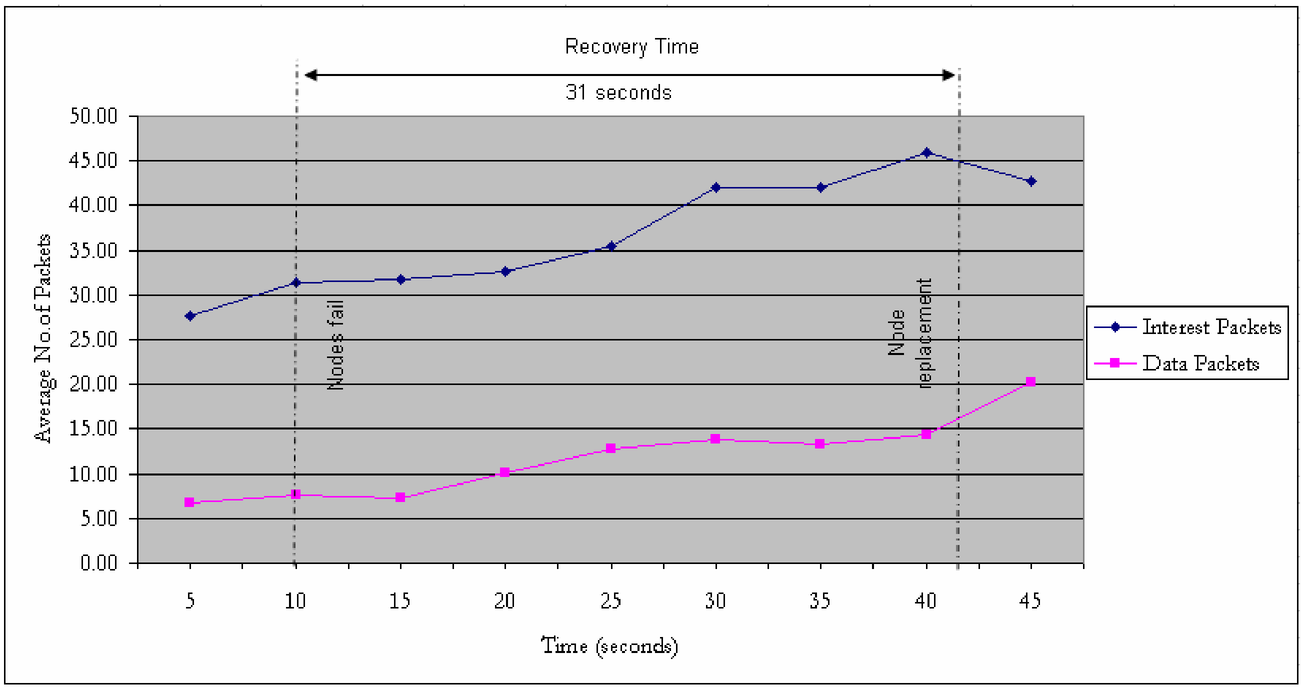

Figure 5.10 Adaptation using Directed Diffusion for three groups

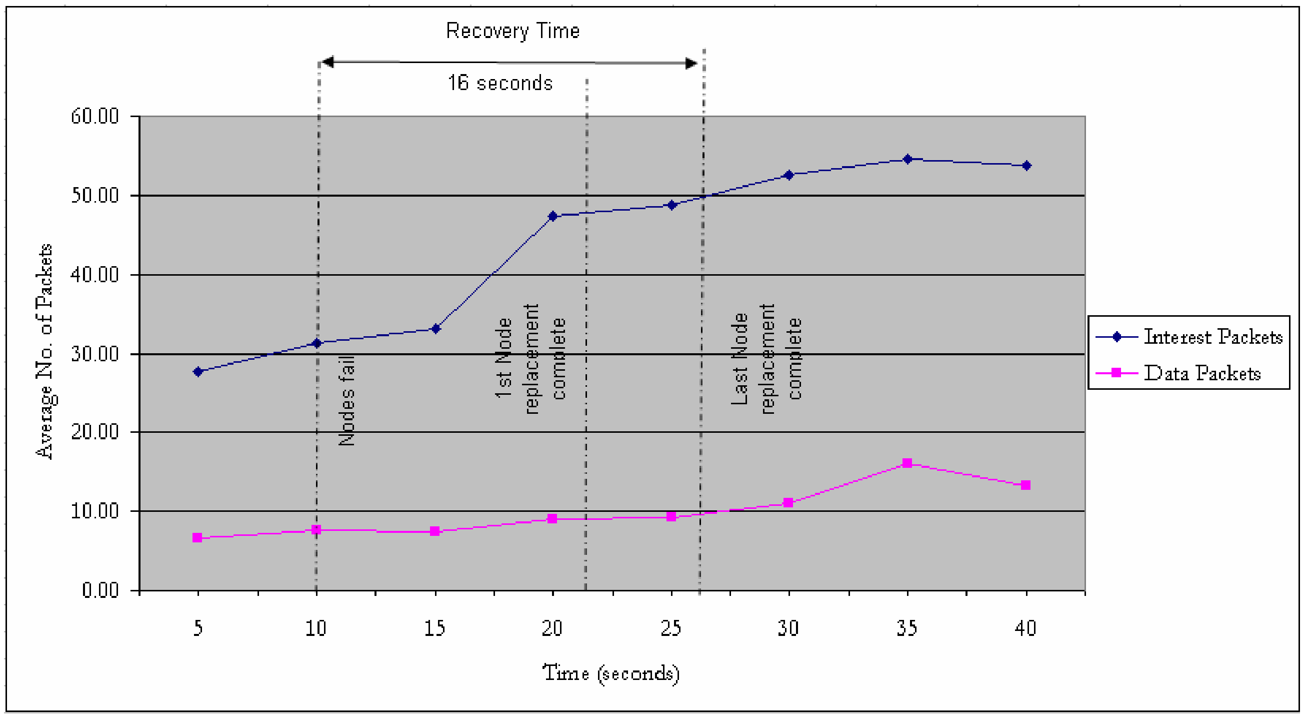

Figure 5.11 Adaptation using Composition server and Adaptation server for three groups

As can be deduced from Figures 5.10 and 5.11, the adaptation with the distributed services support results in a considerably smaller recovery time (16 seconds) than with directed diffusion (31 seconds). Applications constructed to take advantage of these system services can recover from failure fairly quickly.

Directed diffusion tries to cope with loss of data by flooding the initial interest over the entire network until an alternate data source is found. This detection and replacement discovery phase is done only after data loss is confirmed after some period and generates extra traffic in the sensor network. In comparison, an application has to simply send one indication to the adaptation service about node failure and the whole replacement discovery process is carried out by the services using controlled flooding. This results in a much shorter recovery time. There may be slight increase in the density of interest packets which occurs due to the communication between all services and includes the messages sent from the downstream nodes to the adaptation server.

CHAPTER 6: CONCLUSIONS AND FUTURE WORK

An ad-hoc sensor network is composed of resource-constrained nodes working in an unpredictable and possibly hostile environment. Although each sensor node can generate a fair amount of data when tasked, a node by itself cannot guarantee the fidelity of the generated data. Several nodes in collaboration with each other can support more realistic and flexible applications in the dynamic sensor network. Distributed system services provide the bridge between complex communication-intensive user applications and individual sensor characteristics. The composition service allows groups of nodes to be formed and manages them to enable proper coordination among the nodes.

Some of the goals the composition service accomplishes are given below.

- Applications can use the composition service API to send group formation or creation requests to the composition server. Likewise, the server uses the API to listen to requests and respond to them. The API thus frees the application from any diffusion routing protocol specifics.
- Nodes that have matching characteristics are grouped together into one group. Some of these characteristics are close physical proximity, node status, power level and so on. The individual characteristics of each node are thus taken into account in order to provide maximum benefit to the application.

- The stored configuration data can be used by the adaptation service to restore the task group in a transparent manner to the application. Task group reconfiguration can also be done for higher performance at runtime.

This document described the architecture and implementation of the composition service which was tested on a sensor network of considerable size. The composition service framework provides an API for all the requirements of the application.

- Joining/Leaving a group
- Building/deleting/modifying connectors
- Sending/receiving messages through connectors

The higher-level interfaces provided by the composition service framework such as `SensorObject`, `Event` and `QosParams` allow for easy representation of the sensor node properties. Moreover, these interfaces take into account the memory limitations of a sensor node and are lightweight – the `SensorObject` is of 32 bytes, the `QosParams` is 16 bytes and the `Event` is of variable size because it depends on category and type.

The composition service also inter-operates with other distributed services in the network for dynamic adaptation to node failure and reconfiguration of the task cluster. The service was tested on 30 nodes with multiple clusters and significant performance improvement was observed as compared to the adaptation provided by directed diffusion.

- For a cluster of 5 nodes, the replacement node is discovered and the replacement is done in 6 seconds from the time the node fails when the composition server works with the lookup service and adaptation service. In contrast, it takes 14 seconds for directed diffusion to discover an alternate source of data. There is a drop in the interest packet density when distributed services are used for node replacement because of controlled flooding, whereas diffusion floods interest packets over the whole network.

- For three clusters each of 5 nodes, the first replacement node is found in 11 seconds and the last replacement is found within 16 seconds from the time node failure occurs. By comparison, it takes 31 seconds for diffusion to find the first alternate data source.

Future improvements to the composition service are proposed in the following directions.

- Current implementation supports the operation of one composition server in the network. The behavior of the system when several such services operate in the network needs to be studied. A protocol for server to server communication and synchronization needs to be defined.
- The current grouping process needs to be upgraded to exhibit dynamic re-clustering. In the current scenario, it is possible to have clusters within clusters. Dynamic re-clustering will result in more efficient task group configuration.
- The connector layer API uses some constructs specific to the directed diffusion routing protocol. A way of eliminating this dependence and employing those constructs completely in the Communication Interface layer needs to be devised.

REFERENCES

1. David Tennenhouse, "Proactive Computing", Communications of the ACM, Vol. 43 Issue 5, May 2000.
2. Alvin Lim, "Distributed Services for Information Dissemination in Self-Organizing Sensor Networks", Special Issue on Distributed Sensor Networks for Real-Time Systems with Adaptive Reconfiguration, Journal of Franklin Institute, Elsevier Science Publisher, Vol. 338, 2001, pp. 707-727.
3. J.M. Kahn et al., "Next Century Challenges: Mobile Networking for Smart Dust", Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile computing and Networking, August 1999.
4. Jason Hill et al., "System Architecture Directions for Sensor Networks", Proceedings of the 9th International conference on architectural support for programming languages and operating systems, Vol. 34, Issue 5, November 2000.
5. Chalermek Intanagonwiwat et al., "Directed Diffusion for Wireless Sensor Networking", IEEE/ACM Transactions on Networking (TON), Vol. 11 Issue 1, February 2003.
6. David B. Johnson, David A. Maltz, "Dynamic Source Routing in Ad Hoc Wireless Networks", Proceedings of the workshop on Mobile Computing Systems and Applications, pp. 158-163, IEEE Computer Society, Santa Cruz CA, December 1994.

7. Fabio Silva et al., “Network Routing Application Programmer’s Interface (API) and Walk through 9.1”, USC/Information Sciences Institute, December 2002.
8. Xuan Yu, “Implementation of Lookup Service Protocols for Self-Organizing Sensor Networks”, Masters Thesis, Department of Computer Science and Software Engineering, Auburn University, 2001.
9. Ye Wang, “A Dynamic Adaptation Service Framework in Self-Organizing Sensor Networks”, Masters Thesis, Department of Computer Science and Software Engineering, Auburn University, 2002.
10. Frank Luders, “Adopting a Software Component Model in Real-time Systems Development”, Proceedings of the 28th annual NASA/IEEE Software Engineering workshop, August 2004.
11. Jan Blumenthal et al., “Wireless Sensor Networks – New Challenges in Software Engineering”, Proceedings of the 9th IEEE Int. Conference on Emerging Technologies and Factory Automation, September 2003.
12. Mark Ivester, “Interactive and Extensible Runtime Framework for Execution and Monitoring of Sensor Network Services”, Masters Thesis, Auburn University, 2005.
13. G.J. Pottie et al., “Wireless Integrated Sensor Networks”, Communications of the ACM, vol. 43 no. 5, May 2000, pp. 51-58.
14. Wendi Heizelman et. al., “Middleware to support sensor network applications”, Network, IEEE, Vol. 18 Issue 1, pp.6-14, February 2004.
15. Ting Liu, Margaret Martonosi, “Impala: A Middleware System for Managing Autonomic, Parallel Sensor Systems”, ACM SIGPLAN Notices, Proceedings of the

ninth ACM SIGPLAN symposium on Principles and practice of parallel programming, June 2003.

16. Christopher M. Sadler et. al., “Wide Area Monitoring of mobile objects: Implementing software on resource-constrained mobile sensors: experiences with Impala and ZebraNet”, Proceedings of the 2nd International conference on mobile systems, applications and services, MobiSys 2004.
17. S. Li, S. Son, and J. Stankovic, “Event detection services using data service middleware in distributed sensor networks”, Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks, April 2003.
18. Richard E. Schantz, Douglas C. Schmidt, “Middleware for Distributed Systems: Evolving the Common Structure for Network-centric Applications”, Encyclopedia of Software Engineering, John Marciniak and George Telecki, Eds. Wiley & Sons, New York, 2002.
19. John Heidemann et al., ”Building Efficient Wireless Sensor Networks with Low-Level Naming”, ACM SIGOPS Operating Systems review, Proceedings of the 18th ACM symposium on OS Principles, pp. 146-159, October 2001.
20. Yan Yu et al,”Geographical and Energy Aware Routing: A Recursive Data Dissemination Protocol for Wireless Sensor Networks”, UCLA Computer Science Dept. tech. rep., UCLA-CSD TR-010023, May 2001.
21. Richard Brooks, “Reactive Sensor Networks”, Applied Research Laboratory, Pennsylvania State University, 2002.
22. Richard Brooks, S. S. Iyengar, “Multi-Sensor Fusion, Fundamentals and Applications with Software”, Prentice Hall PTR, 1998.

23. S. Mahaney and F. Schneider, “Inexact Agreement: Accuracy, Precision and Graceful degradation”, 4th ACM Symposium on the Principles of Distributed Computing pp. 237-249, ACM, New York 1985.

24. Joseph LaViola, “A Comparison of the Unscented and Extended Kalman Filtering for estimating Quaternion Motion”, In the Proceedings of the 2003 American Control Conference, IEEE Press, 2435-2440, June 2003.

25. Qiao Shen, “A Distributed Composition Service for Self-Organizing Sensor Networks”, Masters Thesis, Department of Computer Science and Software Engineering, Auburn University, 2002.