Adaptation Service Framework for Wireless Sensor networks with Balanced Energy Aggregation

Eun kyung Kim

Certificate of Approval:

Adaptation Service Framework for Wireless Sensor networks with Balanced Energy Aggregation

Eun kyung Kim

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
May 11, 2006

Adaptation Service Framework for Wireless Sensor networks with Balanced Energy Aggregation

Eun kyung Kim

Signature of Author

Date of Graduation

Thesis Abstract

Adaptation Service Framework for Wireless Sensor networks with Balanced Energy Aggregation

Eun kyung Kim

Master of Science, May 11, 2006

86 Typed Pages

Directed by Alvin S. Lim

Wireless sensor networks consist of tiny, energy-constrained sensor nodes that may be deployed in large numbers. Considering these unique characteristics, our adaptation service framework is designed to deliver distributed events and react to changes through distributed actions. This adaptation service framework makes use of energy efficient data aggregation which helps to maximize network lifetime under limited energy constraints. The architecture for efficient adaptation service with distributed events consists of three type of components: event sensor, adaptation server, and action node. An event sensor is responsible for detecting, collecting, and sending events to an adaptation server. An adaptation server receives events from various event sensors and sends action requests to action nodes. An action node executes the requested action and replies to the adaptation server when it has completed the action.

Our research primarily focuses on energy-efficiently delivering the large amount of events by building on prior work done on data aggregation over Directed Diffusion. Although using aggregation paths is energy efficient, nodes at aggregation points consume more energy than any other nodes since they are expected to have greater load and processing overhead for aggregating events. As a result, these nodes have a shorter system lifetime than others. Our new aggregation scheme, called *balanced energy aggregation*, focuses on maximizing the overall lifetime of the sensor network. When a node at an aggregation point is overloaded, the next closest node to the sources on a shared path is selected as a new aggregation point. This allows for energy to be saved by distributing loads across different paths from the sources to the sinks. Our programming model is based on a concept of channels which encapsulate properties of the underlying communication system, i.e. a data forwarding path with aggregation points for conserving energy and maximizing lifetime. Channels are accessible to applications for communication used between adaptation servers and action nodes as well as event sensors and adaptation servers. The algorithms have been successfully implemented and tested over Directed Diffusion protocol. Our experimental results demonstrate that the proposed aggregation algorithm outperform previous methods such as Greedy Aggregation and Path Sharing Aggregation in terms of system lifetime, average dissipated energy and number of events.

Style manual or journal used Journal of Approximation Theory (together with the style known as "aums"). Bibliograpy follows van Leunen's *A Handbook for Scholars.*

Computer software used The document preparation package TEX (specifically LATEX) together with the departmental style-file `aums.sty`.

Table of Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

The improvements in digital circuity technology allow for integration of sensing, processing, and wireless communication on a single chip. Small and inexpensive sensor nodes can be deployed everywhere, resulting in distributed wireless sensor network to sense and collect various types of information. A typical task of a wireless sensor network is the monitoring of a larger area for some given physical quantity, e.g., temperature. Unexpected events such as network link failure and hardware failure could arise due to loosely connected wireless networks, sensor node mobility or limited energy. Wireless sensor network need an adaptation service that delivers continuous events and responds appropriately to the changes in information conditions or unexpected events.

Event-based communication is well suited for addressing the requirements of the adaptation service that delivers events and actions. In general, event-based communication provides communication mechanism for supporting large-scale and dynamic distributed applications in heterogeneous environment. In wireless sensor networks, event-based communication mechanisms are used widely in systems involving mobility of mobile computers, adaptive systems which react to changes in resources and demands for quality of service, and all type of monitoring applications such as earthquake monitoring, habitat monitoring or target tracking and detection. Distributed event-delivery systems, such as ECho [10], and Event Web [7] supports event-based communication and handle all types of events.

Distributed event-based systems in sensor networks have several limitations such as severe energy constraints, redundant data, and data flow over multiple paths. Events need to be disseminated and routed in ways that increases energy efficiency and network lifetime in the sensor networks. Data aggregation is an essential paradigm for wireless routing in sensor networks. If large amount of events are sent to a destination through individual paths, limited resources in the sensor network may be wasted. Instead of sending every event individually, aggregated events combined in the intermediate node are forwarded through shared paths to reduce network traffic. Previous work on Greedy Aggregation [12] and Path Sharing Aggregation [17] has explored energy conservation. Greedy aggregation creates greedy incremental tree for data aggregation, introducing the incremental cost message. In the other hand, path sharing aggregation eliminates the extra incremental cost messages and creates more energy efficient shared paths compared to the greedy aggregation.

Although using aggregation paths is energy efficient, nodes at aggregation points consume more energy than any other nodes since they are expected to have greater load and processing overhead for aggregating events. As a result, these nodes have a shorter system lifetime than others. Our new aggregation focus on maximizing the overall lifetime of the sensor network. Data forwarding path with such an aggregation path for energy efficiency and maximum lifetime is called a balanced energy efficient channel.

We propose an adaptation service framework that deal with hundreds to thousands of events as well as various types of events and actions for monitoring systems or adaptive systems. Efficient data aggregation that maximize network lifetime is used to deliver events and actions. The architecture for efficient adaptation service with distributed events consists of three type of components: event sensor, adaptation server, and action node. An event sensor is responsible for detecting, collecting, and sending events to an adaptation server. An adaptation server receives events from various event sensors and sends action requests to action nodes. An action node executes the requested action and replies to the adaptation server when it has completed the action.

The remainder of the thesis is organized as follows. We first discuss the background information for this research and related research that motivates this research in Chapter 2. We summarize directed diffusion protocol and programming API, which is the basis for this research. Then, we describe Greedy Aggregation and Path Sharing Aggregation, that are the prior works on data aggregation mechanisms over Directed Diffusion. We also discuss an event-based communication model that delivers a large number of events and supports the implementation of dynamic distributed applications. In Chapter 3, we present the architecture of the adaptation service and our improved algorithm, *Balanced Energy Aggregation*, for optimizing the energy efficiency and event propagation. Following that, Chapter 4 describes the implementation of our energy efficient adaptation service. Chapter 5 examines the performance of our *Balance Energy Aggregation* and compares it with *Greedy Aggregation* and *Path Sharing Aggregation*. Finally, Chapter 6 summarizes our conclusions and discusses some key areas for future work.

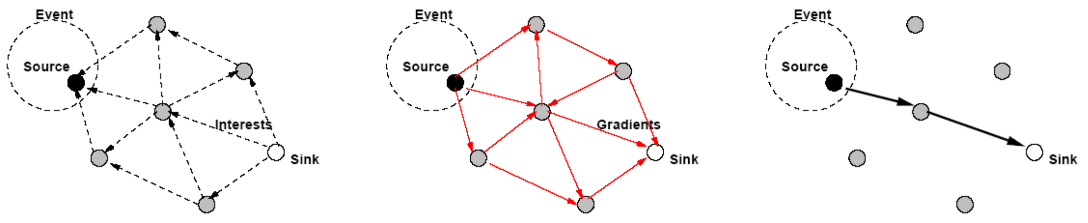
Chapter 2

BACKGROUND and RELATED WORK

2.1 Directed Diffusion

2.1.1 Directed Diffusion Protocol

Directed diffusion is a data-centric network protocol, that is different from a traditional IP-based network protocol. The traditional IP-based approaches to networking use IP addresses for finding the shortest routes between pairs of addressable end nodes whereas a data-centric approach uses the content of data to find the best route from multiple sources to sinks. A *sink* node that requests a service sends out a request for data by flooding an interest to its neighboring nodes. An *interest* refers to a named description of a service that a sink node requires. The interest is subsequently broadcasted by neighbors and neighbors' broadcasting is continuing until the interest arrives at a *source* node. The source node with the appropriate data for responding to the interest, forwards to the neighbors *exploratory data*. When the interest diffuse throughout the network, a node that receives an interest from neighboring nodes forms a *gradient* pointing to the sending node. A gradient specifies both a data rate and a direction for forwarding events from the sources to the sinks. A sink node eventually receives the exploratory data sent by a source. A sink that receives exploratory data message from more than one neighbor, selects one neighbor from whom it first received the latest data. It becomes empirically the low-delay path [1]. Then, a sink node sends only the selected neighbor reinforced interest which is an interest to receive events at a higher data rate. This chosen neighbor also performs the same procedure on its neighboring nodes from which it received an exploratory data message. The reinforced interest propagates only along the low-delay path, formed by reinforcement gradient. When the reinforced interest arrives at the source, data from a source propagates repeatedly along the reinforced gradient until new interest arrives. In directed diffusion, a sink send the interest periodically, in order to cope with varying network dynamics. Reinforced paths can always change and move because wireless sensor networks are loosely connected and a sensor node itself has a limited lifetime.



(a) Interest propagation

(b) Initial gradients set up

(c) Data delivery along reinforced path

Figure 2.1: A simplified schematic for directed diffusion

As explained above, the directed diffusion can be divided into four steps, namely the interest propagation step, exploratory data propagation step, reinforcement interest step, and data delivery step. In the first two step, the message is flooded throughout the network, creating multiple data paths, but, after the next two steps, future data messages from the source use only one reinforced path.

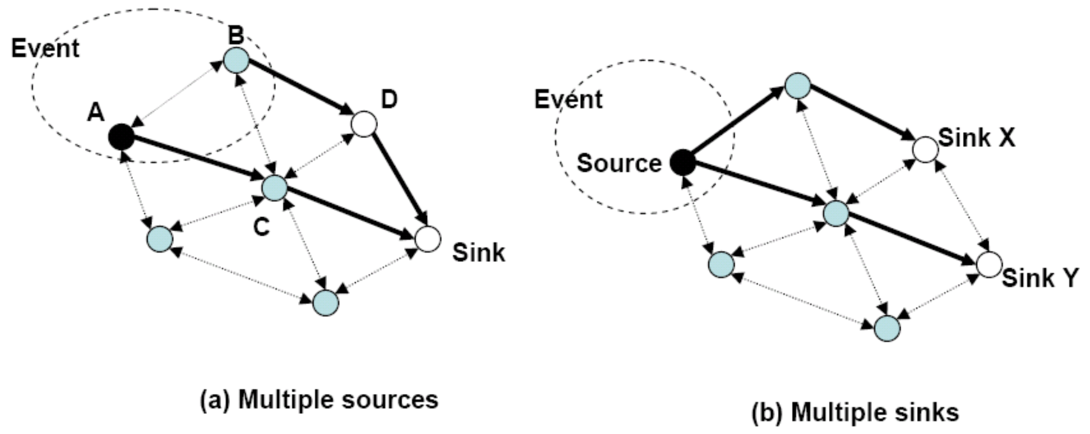


Figure 2.2: Path establishment for multiple sources and sinks

In describing directed diffusion so far, we imply a single source and single sink case. However, as Figure 2.2(a) and Figure 2.2(b) show, directed diffusion works also for multiple sources and sinks. In Figure 2.2(a), data from both sources reaches the sink via both of its neighbors C and D. If the sink hears A's exploratory data earlier via C, the sink reinforces to node C. However, if B's exploratory data comes to the sink earlier via D, not C, the sink send a reinforcement to node D and data streams from source B travel down via D. In this case, the sink gets both sources' data from both neighbors. Similarly, when two sinks send identical interests, directed diffusion follows the same procedure for both sinks: interest propagation, exploratory data propagation, reinforcement, and data delivery step. In Figure 2.2(b), we assume that sink Y has already reinforced to the source. However, when new sink X uses the same interest as sink Y, it determines the empirically best path. Although this multi-path for multiple sources or multiple sinks has an advantage in dispersing data traffic, it has a disadvantage in not aggregating the same type of data to reduce the total network traffic. Data from different sources can be opportunistically aggregated at intermediate nodes along the reinforcement paths. Data aggregation performs in-network data reduction, thereby resulting in significant energy savings in network. Therefore, a proposed data aggregation rule in this thesis is that energy efficient paths would be selected rather than low-delay paths [1]. We will give the full detail of the data aggregation algorithms in Section 2.2.

2.1.2 Directed Diffusion API

The directed diffusion routing and filter API allows for access to the network protocol and extensions to the network mechanisms. The routing API allows user level programs to generate subscriptions for data and publications of data. Directed diffusion filters can dictate how the generated packets will be processed within the sensor network based on the declared attributes of publications or subscriptions. A filter is a process that sits between the directed diffusion core and the user level processes that access the directed diffusion core [2].

```
static NR* NR::createNR()
```

To initialize the NR class, `NR::createNR()` is called. This static function returns an `NR` pointer, which encapsulates all network routing mechanisms and management. The following is a description of methods of the network routing API class for implementing user level applications.

- `handle NR::subscribe(NRAttrVec* subAttrs,`
  `const NR::Callback* callback)`

  - `subAttrs`: A pointer to a vector containing elements describing subscription information.
  - `callback`: A pointer to a callback object to be invoked when the subscription attributes are fulfilled.

- `handle NR::unsubscribe(handle subscription_handle)`
  - `subscription_handle`: A handle associated with a subscription, return by the subscribe method.

- `handle NR::publish(NRAttrVec* pubAttrs)`
  - `pubAttrs`: A pointer to a vector containing elements describing the data to be sent.

- `int NR::unpublish(handle publish_handle)`
  - `publish_handle`: A handle associated with a publication, returned by the publish method.

- `int NR::send(handle publish_handle, NRAttrVec* sendAttrs)`
  - `publish_handle`: The handle of the publication that the data being sent is associated with.
  - `sendAttrs` : A pointer to a vector of data attributes and the original elements describing the publication.

Filters are application-specific software modules that allow applications to influence routing and data processing. Our uses of filters' implementations is for routing and in-network aggregation. A priority is used to define the order filters will be called when multiple filters match the same incoming message. Higher priority filters are called first. The following API describes interfaces used in our filter implementation.

- `handle addFilter(NRAttrVec* filterAttrs, int16_r priority, FilterCallback* callback)`
  - `filterAttrs`: A pointer to a vector containing the filter attributes.
  - `priority`: A unique filter priority between 2 and 253. The higher the number the higher the priority.
  - `callback`: A pointer to the object to be called when an incoming packet matches the filter.
- `void sendMessage(Message* msg, handle h, u_int16_t priority)`
  - `msg`: The pointer to the message to send along to the next filter.
  - `h`: The handle to the filter, returned by addFilter.
  - `priority`: An optional argument to augment the filter matching scheme.

With the Filter API, filters can specify a set of attributes describing what messages they are interested along with a callback. This causes the filter core to send incoming messages matching those attributes to the filter callback. The handle `h` is the same handle returned by `NR::addFilter`. The message `msg` points to a message class containing the message header and the message.

```
class FilterCallback {
public:
      virtual void recv(Message* msg, handle h) = 0;
};

class Message {
public:
      int32_t next_hop_;
      int32_t last_hop_;
      int8_t msg_type_; //INTEREST, DATA, EXPLORATORY DATA,
            POSITIVE REINFORCEMENT, NEGATIVE REINFORCEMENT
      int32_t pkt_num_;
      int32_t rdm_id_;
      .  .  .
      NRAttrVec *msg_attr_vec_;
}
```

2.1.3 Matching Rule

In directed diffusion, the publisher sends data when there are matching subscriptions and publications. One-way matching compares non-IS operators in the first set of attributes against IS operators in the second set, calling it a **match** if the operation of operators is successful. Two-way matching procedure executes one-way matching algorithm twice in both directions. For instance, a sensor would publish this set of attributes [2]:

```
LATITUDE_KEY IS 30.4
LONGITUDE_KEY IS 104.5
TARGET_KEY IS vehicle
```

while a user might look for vehicles by subscribing with the attributes:

```
TARGET_KEY EQ vehicle
```

Whenever a sensor receives a user's subscriptions, it executes the two-way matching procedure described in Table 2.1 to decide whether user subscriptions match with the set of attributes to publish. In the first one-way matching of the sensor, assume that the first set of attributes(`attr1`) is the attributes for a subscription and the second set of attributes(`attr2`) is the attributes for a publication. The attribute(TARGET_KEY EQ vehicle) which has `EQ` operator looks for a matching attribute(TARGET_KEY IS vehicle) in the set of attributes to publish, where operator is `IS`. Then, it compares the value(`vehicle`) of a attribute for subscription with the value(`vehicle`) of a attribute for publication using operator `EQ`.

```
proc OneWayMatch(attr1, attr2)
  For each attribute in attr1 that has an operator different than "IS",
  find a "matchable" one in attr2

proc MatchAttrs(attr1, attr2)
  if (OneWayMatch(attr1, attr2))
    if (OneWayMatch(attr2, attr1))
      return true;
  return false;
```

Table 2.1: Matching rule

When data or exploratory data messages find matching subscriptions, two-way matching procedure is performed. Filters are a special case for the matching rules. When matching filters against incoming messages, only one-way matching is performed.

2.2 Data Aggregation

2.2.1 Overview of Data Aggregation

Sensor networks are typically data driven. The network nodes cooperate in forwarding data from sensors to sinks. However, one of the main challenges is the fact that they are usually power constrained. Sensor networks' power limitation is aggravated by the fact that, once deployed, they are left unattended for most of their lifetime. Thus, a fundamental challenge in the design of wireless sensor networks is to maximize their lifetimes.

Data aggregation is a well known energy-efficient technique for propagating data from data sources to sinks. Instead of sending individual messages to data sinks, intermediate nodes delay messages, compute an aggregated value of all data, and then forward only a single message with the aggregated value. Thus, data aggregation in wireless sensor networks reduces the number of transmissions of sensor nodes, and hence minimizes the overall power consumption in the network.

Similar to data compression, data aggregation can be classified into two different types; namely, lossless aggregation and lossy aggregation [12]. Lossless aggregation refers to concatenating individual data into larger packets, or eliminating redundant information e.g., packet headers. Thus, in this case, no data is lost and detailed information is preserved. On the other hand, lossy aggregation may discard some detailed information and degrade data quality for more energy savings. The best example of lossy aggregation is the averaging of sensor values. For example, a user may need to know only of the average temperature in a region, as opposed the individual readings of all sensors. Similarly, it may be enough to report only the average estimated location of a target, as opposed to the exact locations of all motion sensors. Therefore, Averaging can be a natural choice in many applications. Our application use a pressure value for detecting tsunami. Pressure events sensed in each sensors are averaged at aggregation points. We also refer to perfect aggregation [12], whereby the data size of an aggregate is equal to the data size of an individual event. An aggregated data packet with average pressure has a same size.

Data aggregation efficiency is affected by several factors, such as the placement of aggregation points, the aggregation function, and the density of sensors in the network. The determination of an optimal selection of aggregation points is extremely important. To opportunistically determining aggregation points in directed diffusion may not be energy efficient. Because directed diffusion protocol forward data along low-latency path, data may not be aggregated near the sources. As shown in Figure 2.3, data is early shared and merged in early aggregation close to sources, resulting to less transmitting data than in late aggregation. Thus, we needs to find out proper data aggregation rules to favor the

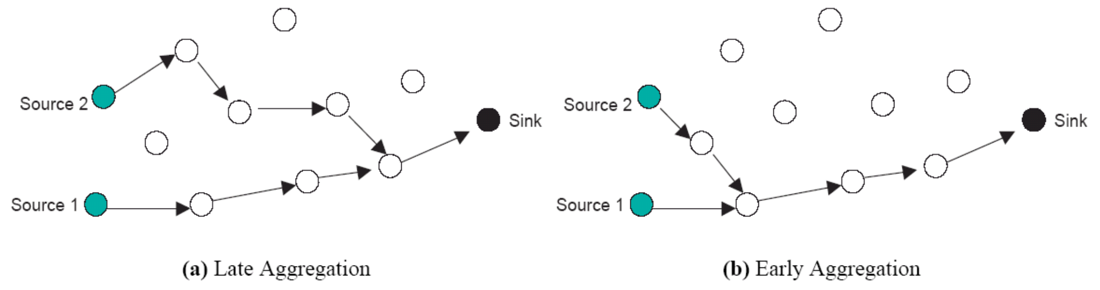


Figure 2.3: An example of late aggregation and early aggregation

path selection to increase early sharing of paths and reduce energy consumption. Greedy

Aggregation [12], that aggregates data in a greedy incremental tree is one of early path sharing aggregations. However, in greedy aggregation, incremental messages are essential to create greedy tree introduce extra traffic. Path sharing aggregation [17], proposed by Shi, eliminates the need for incremental messages and achieves more energy savings. In such an early path sharing aggregation, many sources send data to an aggregation point, the node at an aggregation point could be overloaded with processing overhead for data aggregation. As a result, the lifetime of this sensor network is adversely affected. Thus, we need to propose an aggregation mechanism to balance the network load. The goal of our *balanced energy aggregation* algorithm is to increase both aggregation efficiency and network lifetime. We define a certain threshold in residual energy for each node. When the residual energy reach the threshold, the aggregation point is moved to the next closest neighbor node. We prevent prolonged energy drain of one aggregation point. We will describe the details in Chapter 3.

2.2.2 Greedy Aggregation

As explained above, the greedy aggregation selects energy efficient paths rather than low delay paths using a combination of data aggregation rules and reinforcement rules. To achieve energy saving paths, the energy cost and incremental energy cost are defined in greedy aggregation. Each exploratory event contains an additional attribute E, the energy cost for delivering this event from the source to the current node. In addition, each source generates an incremental cost message when it receives a previously unseen exploratory message generated by other sources. The incremental cost message is the same as the original exploratory message except the E field is replaced by C field, indicating the additional energy cost C required for delivering that exploratory message to the existing tree. Once a sink receives a previous unseen exploratory event, it reinforces the neighbor that forwarded the exploratory event or the incremental cost message at the lowest energy cost. In Figure 2.4, assume there is a pre-existing path $2 \rightarrow 5 \rightarrow 8 \rightarrow 10 \rightarrow 12$ between source 2 and a sink. When source 1 sends exploratory events, the sink selects node 10 to reinforce among node 10 sent exploratory message with incremental cost 2 and node 11 sent an exploratory message with energy cost 5. So, the low energy data path for source 1 is $1 \rightarrow 3 \rightarrow 5 \rightarrow 8 \rightarrow 10 \rightarrow 12$.

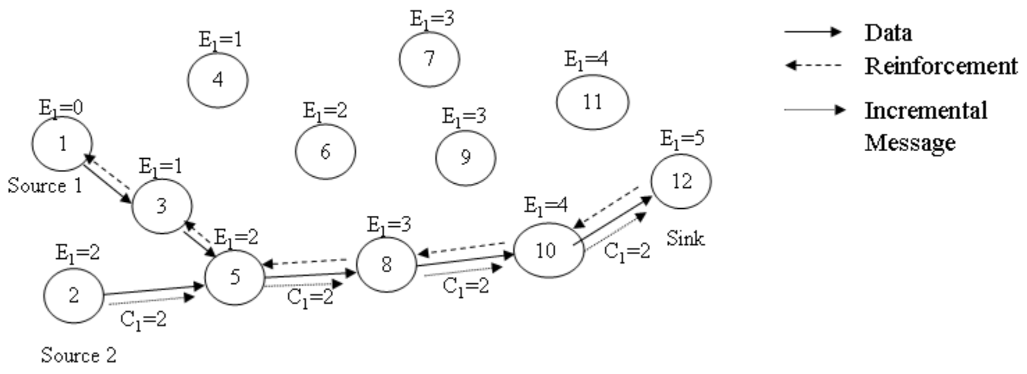


Figure 2.4: Greedy Aggregation

2.2.3 Path Sharing Aggregation

Path Sharing Aggregation (PSA) eliminates incremental cost message that introduce extra complexity and network traffic and redefine the energy cost E. Although the energy cost E is defined as the extra energy cost to deliver data from the source to the current node, the increment of the energy cost E is used to decide whether the high gradient

```
for ( $i = 1$; $i \leq$ number of neighbors; $i$++ )
{
  Search the interest cache to determine if neighbor $i$ has high gradient or not;
  if (neighbor $i$ has high gradient)
      forward this exploratory event without delay and without increment of $E$;
  else
      forward this exploratory event with delay $Te$ and with increment of $E$ by one;
}
```

Table 2.2: Rule for forwarding an exploratory event in PSA

paths exist, as shown in Table 2.2. This is because there is no extra energy cost to send an aggregated data for perfect aggregation. A node may receive exploratory event from different nodes with different E (energy cost) values. The minimum of these values is picked as the energy cost E for this node. When a sink receives all the exploratory event messages from its neighboring nodes, it reinforces the node with a lowest energy cost.

In an example of PSA(Figure 2.5), there are two paths: one is $4 \to 3 \to 5$, and the other is $2 \to 5 \to 8 \to 10 \to 12$. All the nodes on an existing high event-rate path, 3, 5, 8, and 10, have the same energy cost value and node 11 sends an interest to receive data from source, node 1. It indicates that forwarding the data on these nodes will have no extra energy cost with PSA. After reinforcement, data from node 1 will be sent by the following path: $1 \to 3 \to 5 \to 8 \to 10 \to 11$. The total energy cost for this choice is only 2.

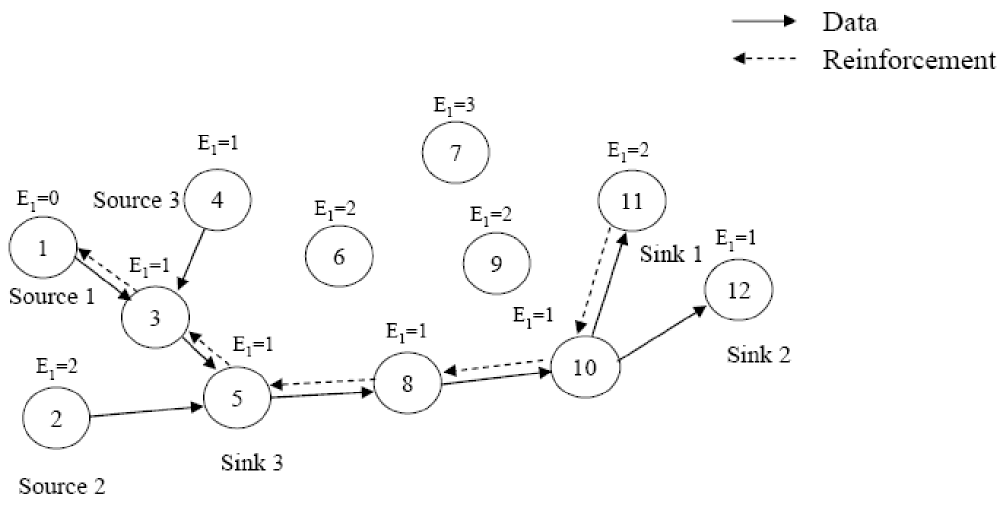


Figure 2.5: Path Sharing Aggregation

2.3 Event channel

2.3.1 Overview of Event-Based Middleware

Event-based middleware allows the applications to interact through event notifications. Event notifications, or simply events, contain data that represent a change of the state of the sending application, called event producers [9]. Events are propagated from producers to the receiving application, called event consumers. Events typically have a name and may have a set of typed parameters whose specific values describe the specific change to the producer's state. In order to receive events, event consumers have to subscribe to the events in which they are interested and event producers publish the events. So, event producers are also called publisher and event consumers are called the subscriber. The sink in directed diffusion is an event subscriber and the source is an event publisher. Directed diffusion provides well-established software architecture for managing events. An event system using event-based middleware, may consists of potentially large number of producers and consumers. Event channels allow multiple producers to communicate with multiple consumers.

The general paradigm of event-based programming has been received widely for more complex distributed systems. We introduce two event-based middleware, *Event web* and *Echo* to support event-based communication programming. Also, one of the event models for real-time systems is introduced.

2.3.2 Event Web

Event web [7] is a software architecture to manage dynamic situation, especially crises. It is an approach for developing distributed applications that help manage crises in rapidly changing situations. The event web architecture consists of four parts: event processors that process event streams; the dissemination network that distributes events among these processors; the event directory that allows for quick discovery of the types of information available in the system; and a service layer that provides various services to the event processors. We will describe details of the event processors and the dissemination network that influenced our research.

An event processor is an object in the event web that may receive and process event streams and may generate new event streams. Event processors are classified into an event generator and an event consumer according to their behaviors with respect to event streams. Each event generator monitors the environment and generates an event stream that describes some aspect of that environment. An event consumer is an event processor that receives event streams from event generators. An event consumer performs actions based on the events it receives. Event generators and event consumers are similar in functionality to event sensors and adaptation servers in our event channel framework. An event sensor is responsible for collecting event and sending an event stream to an adaptation server. An adaptation server receives the events and sends actions to an action node.

The event dissemination network sends a copy of each event published by the event generators to every event consumer interested in that event. For each event consumer, the dissemination network has an associated set of named input event queues. Every subscription by the event consumer is associated with exactly one of these queues. Thus, there is one event channel connecting each event generator to each input queue in the event web. Similarly, in our event channel framework, an adaptation server is responsible for subscribing for events, and event sensors are responsible for publishing events. One channel is created for subscribing and publishing and an adaptation server has one input queue associated with the channel. Although one channel is created for several events in our event channel framework, applications could create more channel to put different type of events in the different input queue. In addition, an adaptation server has a channel for communicating with a set of action nodes. The input queue of the channel for action store action reply messages from action nodes in the order that they arrive.

Overall, event web provides an architecture to manage crises effectively and the ability to quickly process potentially large amounts of information about a rapidly changing world. On the other hand, our event channel framework is for adapting systems which react to changes in sensor networks.

2.3.3 ECho : Event Delivery system

ECho is a distributed event delivery middleware system, developed by Georgia Institute of Technology [10]. Like most event systems, what ECho implements can be viewed as an anonymous group communication mechanism. Message are delivered to receivers according to the rules of the communication mechanism. In this case, event channels provide the mechanism for matching senders and receivers. Messages (or events) are sent by sources via channels which may have zero or more subscribers (or sinks). A program or system may create or use multiple event channels, and each subscriber receives only the messages sent to the channel to which it has subscribed. Sinks subscribers specify a handler to be run whenever a message arrives.

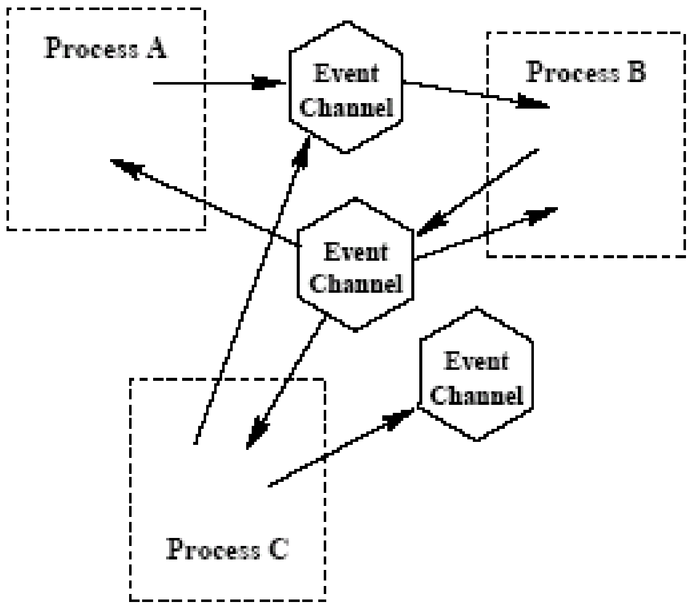


Figure 2.6: Event channels in ECho

The event channels exists in the network between processes. Channels are created once by some process and can be opened anywhere else they are used. The process which creates the event channel is distinguished in that it is the contact point for other processes wishing to use the channel. The channel ID, which must be used to open the channel, contains contact information for the creating process as well as information identifying the specific channel.

Although the functionality of our event channel is similar to that of the event channel described by ECho, ECho does not specify action channels. Unlike ECho, our event channel are created and accessed without channel ID since characteristics of channels and the role of the processes accessing the channels are already defined in the channels.

2.3.4 The Event model for Real-Time Systems

In a real-time event system, the communications architecture substantially affects the temporal and reliability properties of event dissemination. Specifically, manufacturers have access to critical business events and can handle them in a timely fashion. Watson Research Center presented a proactive, real-time sensing and alert management system [6]. This system enables continuous monitoring of diverse data sources and generate alerts based on domain specific rules.

At the center of the system solution is an Event Stream Processor which handles all events, ultimately deciding on the actions that need to be taken. When running, the Event Stream Processor proceeds through the following steps: event message receipt, event transformation, metric calculation, metric evaluation, and Action invocation. During message transformation, the messages are converted into a common format. The formatted messages are used for metric calculation. Then, the metric evaluation service determines what actions if any need to be taken based on the newly calculated metric. Our adaptation server may simplify this Event Stream Processor and specify an abstraction accessible to applications.

Chapter 3

ARCHITECTURE AND ALGORITHM FOR ADAPTATION SERVICE

3.1 Architecture of Adaptation Service

Adaptation service in self-organizing distributed sensor networks in prior work by Wang [16] was defined as a service for modifying the behavior of the sensor network based on changes in the environment. The adaptation service may initiate self-reconfiguration in response of changes due to processor or link failures, changes in the communication patterns, or changes by user requirements in sensor networks. Adaptation servers in such an adaptation service monitor sensor nodes and execute runtime reconfiguration operations to recover from failures. Also, the previous work described a general adaptation service framework for supporting distributed adaptive system. It presents a common adaptation model that is divided into three phases: change detection, agreement, and action. The change detection phase monitors possible changes in the environment. The agreement phase assesses the change to decide if the action is necessary. In action phase, the action is executed as a result of the agreed changes. The model was used for the adaptation service in self-organizing sensor networks [16].

Based on the general adaptation service and event driven architecture in Chapter 2, we develop an energy efficient adaptation service. We divide an adaptation service into event service and action service. Event service receives events or changes from sensor nodes and action service performs actions in response to events or changes. Three types of components are involved in an adaptation service: event sensors, adaptation servers, and action nodes. An event sensor is responsible for collecting and sending events related to changes in an environment to an adaptation server. An adaptation server receives the events, performs agreement for actions and sends action requests to an action node. An action node executes the requested action and replies to an adaptation server when it has completed the action.

These components in an adaptation service are distributed in the sensor network and have to be able to deal with hundreds to thousands of events as well as various types of events and actions, such as earthquake monitoring, target detection, or changes in a network configuration. So, we need event channels capable of delivering all types of events and actions among nodes in a energy efficient way. Shared paths for data aggregation as described in Section 2.2, are use for delivering events and actions.

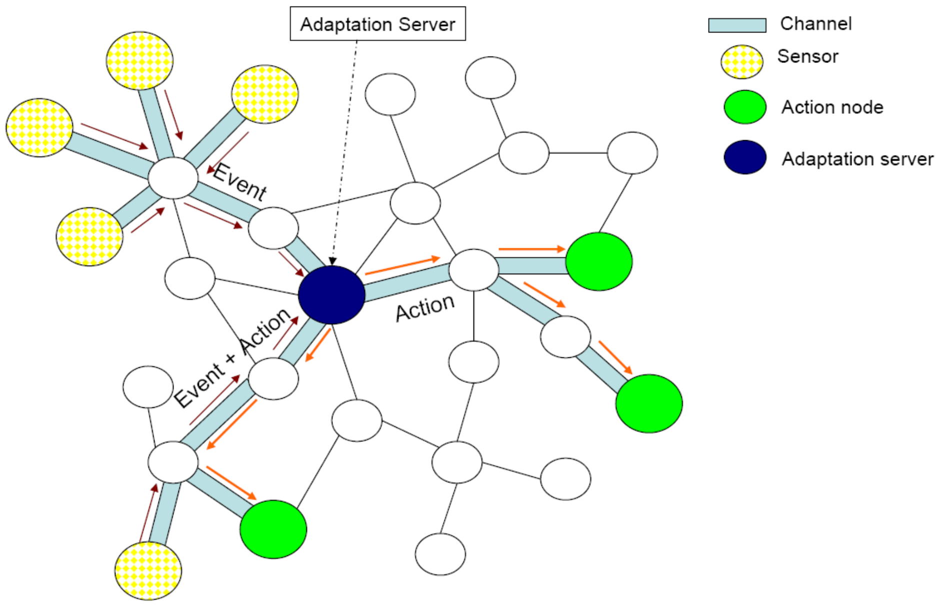


Figure 3.1: Adaptation service by a channel

As shown in Figure 3.1, a sensor network forms channels between sensors, adaptation server, and action nodes. The event data or action data are propagated to and from the adaptation server through these channels. Channel could be logically classified into two types of channel: event channel and action channel. In general, an event channel is shared between sensors and adaptation servers and an action channel is shared between action nodes and adaptation servers. Each channel may create its own data aggregation path or shared path. An event channel implements many-to-one communication relation whereas an action channel implements one-to-many communication relation. Each adaptation server could communicate with multiple event sensors, i.e. the subscriptions of a adaptation server may be received by many event sensors and multiple event sensors may respond to them by sending data to a adaptation server. An adaptation server can receive all types of events and decide if actions are necessary. On the other hand, each adaptation server could communicate with multiple action nodes. Action request messages sent by an adaptation server may be received by multiple action nodes which has same type of actions. Action nodes that receive action request message and send action reply messages to an adaptation server.

3.2 Balanced Energy Aggregation

Balanced Energy Aggregation (BEA) is an algorithm of sharing paths for efficiently delivering events and actions. BEA not only achieves energy saving due to in-network data aggregation, but also provides methods for maximizing system lifetime of sensor networks. In this section, we will give an algorithm of BEA and see how it works through an example in a simple sensor network.

3.2.1 Energy Model

Before we descirbe details of the *balanced energy aggregation*, we compute the energy cost and the extra cost for *balanced energy aggregation* in this section. Many energy models for predicting energy has been presented. Among them, Heizellman *et al.* presented a simple energy model where transmission energy is twice that of receiving energy [19]. Energy consumption on sensor node is given by,

$$\boldsymbol{EC = R + 2T} \tag{3.1}$$

where R is the number of receiving packets and T is the number of sending packets. Intanagonwiwat *et al.* [12] used an energy model such that the power dissipation in the idle time was about 39 mW, receive power dissipation was about 396 mW, and the transmit power dissipation was about 660 mW. This model is close to Heizellman's model. The energy cost calculation of this research follows these two energy models. Thus, dissipated energy in a node is defined by,

$$\begin{aligned}
E &= DissipatedEnergy && (3.2)\\
&= Er(EnergyForReceiving) + Et(EnergyForSending) && (3.3)\\
&= \frac{ReceivingBytes}{TransmissionRate} * receivePowerDissipation && (3.4)\\
&\quad + \frac{SendingBytes}{transmissionRate} * transmitPowerDissipation && (3.5)\\
&= \#bytes/(1000bps) * 0.395W + \#bytes/(1000bps) * 0.660W && (3.6)
\end{aligned}$$

Usually, energy is defined by $time * power$, and $TransmissionRate$ is calculated as the measured number of units of data, such as bits, transmitted during a time interval divided by the $time$. The dissipated energy that is estimated by counting the number of packets sent and received could be redefined as $(numberofbits * TransmissionRate) * power$.

Residual energy rate, r is the ratio of residual energy to initial energy in the node. The residual energy can be redefined as the difference between the initial energy of each node and the dissipated energy computed by Equation 3.2 to 3.6. So, residual energy rate is calculated by using the dissipated energy as shown in Equation 3.10.

$$\begin{aligned} r &= ResidualEnergyRate && (3.7) \\ &= \frac{R(ResidualEnergy)}{I(InitialEnergy)} && (3.8) \\ &= \frac{I - E}{I} && (3.9) \\ &= 1 - E/I && (3.10) \end{aligned}$$

Finally, extra cost measures the ratio of a pre-defined threshold to residual energy rate, r. The threshold is the level of residual energy rate below which the sensor node may not have sufficient energy to sustain continuous transmissions.

$$ExtraCost = \frac{Threshold}{r} \qquad (3.11)$$

3.2.2 Algorithm

Nodes at aggregation points consume more energy than any other nodes since they are expected to have heavy load and high processing overhead for aggregating data. As a result, the nodes has lower system lifetime than others. *Balanced Energy Aggregation* (BEA) improves the overall lifetime of the sensor network. Though it may not perform the most efficient aggregation, i.e. aggregating data at the closest point to the sources as shown in Figure 2.3, BEA will prolong the network life time by balancing the load. When a node at an aggregation point is overloaded, new aggregation point is the next closest node near sources along the shared path. The algorithm chooses balanced energy aggregation to achieve energy efficiency as well as extensions for network lifetime.

Balanced energy aggregation implements energy efficient load-balancing aggregation by following the path sharing aggregation scheme, proposed by *Shi* [17] with a special modification for balancing a load. As in the greedy aggregation, and path sharing aggregation, $EnergyCost$ is defined as the cost to deliver data from the source to the current node in the normal case. The energy cost is increased by one when an unreinforced hop is taken. If a path already exists, the energy cost is not incremented. When nodes are overloaded, an extra cost is added to the energy cost. Table 3.1 shows a pseudo code of the the *Balanced Energy Aggregation* algorithm. Whenever exploratory messages are received in each node, Balanced Energy Aggregation examines the residual energy of this node. If the residual energy is less than a defined threshold, it sets an extra cost. The extra cost would be greater than 1 since it is computed as the ratio of a pre-defined threshold to residual energy rate, r, as shown in Equation 3.11. It is only set when the residual energy rate is less than the threshold. If this node has enough residual energy, the extra cost is 0. When the extra cost is 0, this node is not overloaded and the energy cost of this node is same as the minimum energy cost of all the energy costs from neighbors. When an exploratory timer (introduced in order to receive the exploratory messages from all the neighbors) expires, a node computes its own energy cost. The

```
if ( message type is EXPLORATORY DATA )
    compute residual energy rate, $r = \frac{R(ResidualEnergy)}{I(InitialEnergy)}$;

if ( Residual energy rate($r$) less than Threshold )
    set $ExtraCost = \frac{Threshold}{r}$;
else
    set $ExtraCost$ to 0

if ( exploratory timer is expired )
    Set energy cost as follows:
          find $MinEnergyCost$, which is the minimum value among energy costs of
              all incoming exploratory messages;
          $EnergyCost = MinEnergyCost + ExtraCost$;


for ( $i = 1$; $i \leq$ number of neighbors; $i{+}{+}$ )
{
    determine if neighbor $i$ is placed on a reinforced path;
    if (neighbor $i$ is on a reinforced path)
        forward this exploratory event without delay
        and without incrementing $EnergyCost$;
    else
        forward this exploratory event with delay
        and after incrementing $EnergyCost$ by one;
}
```

Table 3.1: Algorithm for Balanced Energy Aggregation

energy cost is the minimum value of the energy costs in all exploratory messages received from neighbors plus the extra cost. The exploratory message with the computed energy cost have to be forwarded to the neighbors. The procedure for forwarding the exploratory message is same as that of Path Sharing Aggregation shown in Table 2.2. If the neighbor has high gradient, i.e. the neighbor to forward the exploratory message is on an existing reinforced path, the exploratory message is forwarded without increment of the energy cost. Otherwise, the exploratory message is forwarded with some delay after incrementing the energy cost by one. The delay is introduced to make sure that exploratory messages with low energy cost sent along an existing path arrives earlier than the exploratory messages with the high energy cost [17].

When the source sends an exploratory message to its neighbors, it decides if the energy cost should be increased. On the existing reinforced path, source sends exploratory message with energy cost 0 to neighbors. Otherwise, it sends exploratory data with energy cost of 1 to neighbors not already on an existing reinforced path. An intermediate node caches exploratory messages in order to find the neighbor with the minimum energy cost. After a exploratory timer for cache expires, nodes compute its own energy cost using the extra cost. Like the source, the intermediate nodes decide whether to send incremented energy cost to their neighbors before sending exploratory data messages to them. When the sink node receives all exploratory messages, it checks the energy cost for every exploratory messages from its neighbors and favors the neighbor with the lowest energy cost to reinforce. Similarly, all the intermediate nodes select the neighbor in the same way and a lowest energy cost path can be set up between the source and the sink.

3.2.3 Example of BEA

Figure 3.2 shows how Balanced Energy Aggregation works, compared to path sharing aggregation in the same network configuration. Path Sharing Aggregation, as explained in Section 2.2, is first used in the scenario of Figure 3.2(a). Initially, there is an existing reinforced path, from node 2 to node 12 ($2 \rightarrow 5 \rightarrow 8 \rightarrow 10 \rightarrow 12$). Source 1 then sends an exploratory message to sink 1, which sends a reinforcement interest back through 12 → 10 → 8 → 5 → 3 → 1. Before the residual energy of node 5 reaches the threshold, data flows through the same path as PSA(2 → 5 → 8 → 10 → 12). As node 5 becomes overloaded and the residual energy drops below the threshold, using BEA, the reinforced interest from sink 1 goes through 12 → 10 → 8 → 6 → 3 → 1.

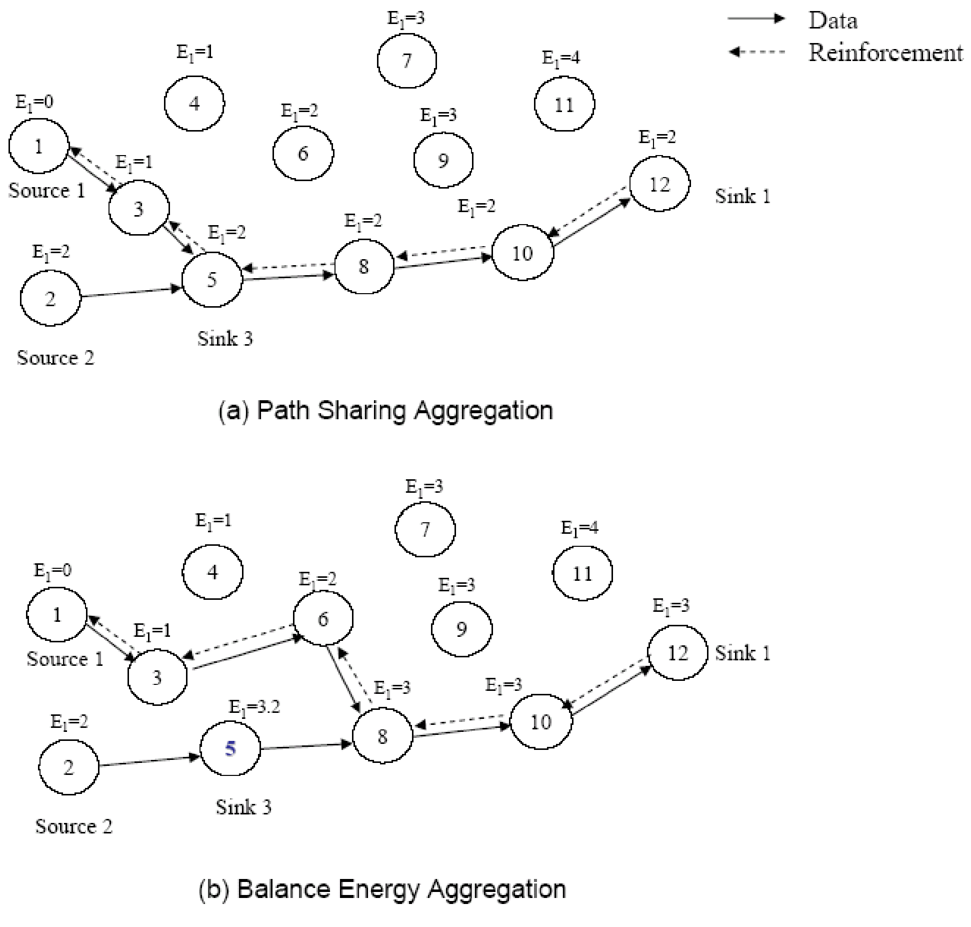


Figure 3.2: Balanced Energy Aggregation

Exploratory messages from source 1 include the energy cost. When a node receive exploratory messages with the energy cost from all its neighbors, it computes a new energy cost and forwards the exploratory messages with the new energy cost to its neighbors. Node 5 may receive two exploratory messages before a timer for this message has expired. One is from node 3 and the other from node 2. Both of the energy costs are 2, and the message from node 3 arrives earlier than the one from node 2. Therefore, normally, node 5 would forward the exploratory data with lowest energy cost, 2 to node 8 without incrementing the energy cost, because the path between node 5 and node 8 is an already reinforced path. Node 8 performs the same procedure. The energy cost of the exploratory message from node 5 is 2, and that of the exploratory message from node 6 is 3. Therefore, the energy cost of node 8 is 2 and later, it can forward reinforcement interest to node 5. Once the residual energy of node 5 is less than the pre-defined threshold, the energy cost of node 5 is increased to a higher value than 2 due to the extra energy cost. The extra energy cost in Figure 3.2 is 1.2 calculated by Equation 3.11 if the residual energy rate is 0.5 and the threshold is 0.6. Node 5 forwards the exploratory message to node 8 with the energy cost 3.2, which is the sum of the lowest energy cost, 2 and the extra cost, 1.2. Node 8 receives the exploratory message with the energy cost 3.2 from node 5 and the exploratory message with the energy cost, 3 from node 6. Node 8 will choose the lowest energy cost, 3, and will now forward reinforcement interest from the sink to node 6 instead. The extra cost expressed in Equation 3.11 is always greater than 1. When data travels down on non-reinforced paths, energy cost is proportional to the number of hops. If the extra cost in the node 5 is less than 1, node 8 will still forward the reinforce interest to node 5. This is because the energy cost from node 8 is 3 and that from node 5 is less than 3. This means, the extra cost should be greater than 1 to make the next closest neighbor an aggregation point.

3.3 Sensor Data Aggregation Function

The key idea for data aggregation is to combine data from different sensors to eliminate redundant transmissions. Although data aggregation results in fewer transmissions, it comes at the cost of latency. There is a tradeoff between data delay and aggregation efficiency (energy efficiency). To aggregate more data and save more energy, it is necessary to introduce more delay in intermediate nodes. Also, tradeoffs must be made between energy efficiency, data accuracy and freshness. There is a study that shows considerable energy savings while maintaining data accuracy and freshness [20]. Here, the data aggregation method we will propose, is concerned with reducing a data size and network traffic. Our application presents simple perfect aggregation, without considering data accuracy or freshness. The goal of our data aggregation method is to achieve minimum data delay and maximum energy saving.

To aggregate data from different sources at aggregation points, we have two design goals. First, data from one source should be distinguished from data from other sources. Second, if intermediate nodes keep receiving data from only one source, it should not introduce extra delay to forward the data. To achieve these design goals, a message window in every node buffers all incoming data messages. Data messages remain in the message window until the node receives the next message from the same source.

Given the protocol, we will see how each node performs data aggregation in the message window in Figure 3.3.

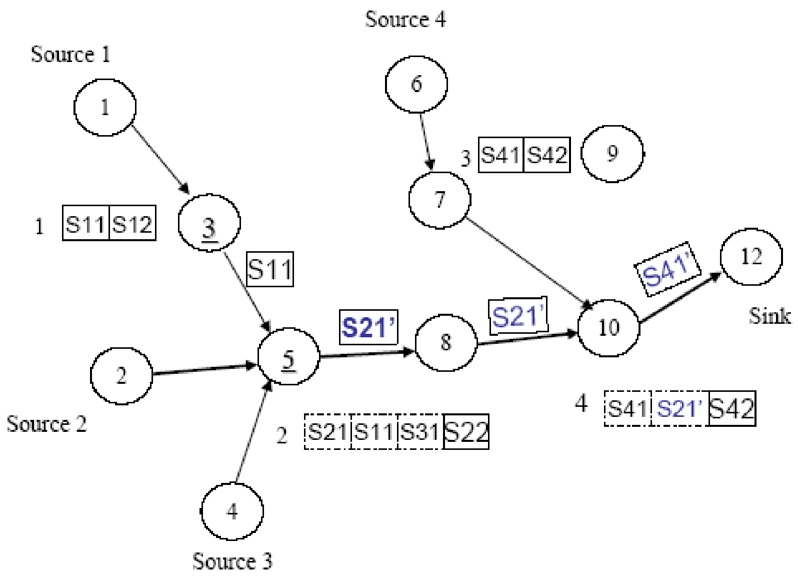


Figure 3.3: Data aggregation function along a shared path

- Node 3 : when node 3 receives the first message, it keeps the data message in the message window. When node 3 receives the second message from Source 1 it forwards the first message to node 5 and keeps the second message in the message window, since the two messages come from the same source.

- Node 5 : Node 5 receives S22, the second message from Source 2. Because it already has a message from Source 2 in the message window, it aggregates all previous messages from different sources. Then, it sends an aggregated message including the location information of Source 2. If node 5 receives S12, the second message from Source 1, not S22, the aggregated message includes the location information of Source 1. Now, only S22 is in the message window, waiting for the next message.

- Node 8 : Node 8 receives only aggregated data messages with one location information. Node 3 functions in a similar manner.

- Node 10 : Node 10 receives aggregated data from node 5, and data from Source 4. When data S42 comes to the message window, there is the message from the same source. So, node 10 aggregates and forwards data with S41 and S21' to the sink(node 12), leaving S42 in the message window.

- Node 7 : Node 7 performs same function as node 3 about data from source 4.

All data messages from a node's neighbors are first placed in the message window. They are not aggregated or forwarded to the next node until a message from the same source arrives. The data event rate is the speed that sources send data. Once a message from the same source as some message in the message window arrives, all messages in the message window are aggregated. All messages are forwarded at the same rate as the event rate defined by the sources. For instance, the first message from Source 1 received at node 3 remains in the message window until the second message arrives from Source 1, at the event rate of Source 1. So, an aggregation delay corresponds to the fastest event rate of all sources that forward packets to the aggregation point. All nodes keep track of the message window to aggregate data. Unlike a sliding window in TCP protocol, the message window does not have a fixed window size. The window size is variable, depending on the number of sources and the event rates of the sources.

Chapter 4

IMPLEMENTATION

4.1 Software Architecture

The implementation of adaptation service consists of a channel library and application, a data aggregation filter, a balanced energy aggregation filter, and a modification of gradient filter. As shown in Figure 4.1, directed diffusion core receives data from the network or applications. As mentioned previously, filters in directed diffusion have priorities for message pre-processing. The data aggregation filter, balanced energy aggregation filter, and modified gradient filter have the priority to pass data. The filters are prioritized in the following descending order: balanced energy aggregation filter, modified gradient filter and data aggregation filter. Therefore, the messages that are forwarded into the network will pass through these three filters.

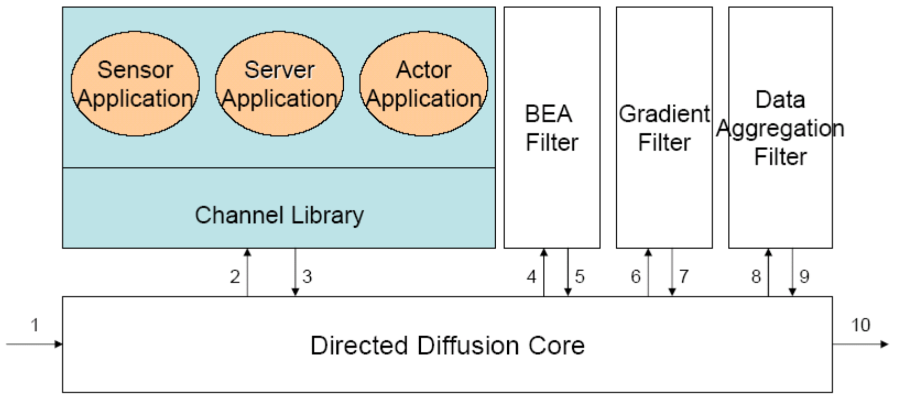


Figure 4.1: Architecture of Adaptation Service

While passing through each filter, the message is manipulated according to the purpose of the filters. First, the BEA filter receives all exploratory messages from all the neighbors, even though they are duplicated. Then, the BEA filter finds the exploratory message with the lowest energy cost and forwards it to the modified gradient filter. The gradient filter forward messages to the neighbors. The energy cost of the forwarded exploratory message is either the same energy cost as it receives from the BEA filter if the neighbor is on a reinforced path (the gradient reinforcement flag is true). Otherwise, the energy cost increased by one. The reinforcement flag is true if the neighbor is on an existing high gradient path, otherwise it is false. The exploratory messages are forwarded to the sink as it is. The sinks send the reinforcement interest through nodes that have the lowest energy cost. The source, after receiving the reinforcement, sends data to a reinforced neighbor. Finally, the data aggregation filter handles the data messages. As described in Section 3.3, it integrates the data messages from different sources and send an aggregated data. Although each message passes through three filters, the message types that each filter subscribes to are different. While the BEA filter see only exploratory data messages, the data aggregation filter handles only data messages. The gradient filter handles all the message types, including interest, exploratory, and reinforcement messages.

The channel library is a collection of directed diffusion APIs, that allows applications to communicate through the channel mechanism and diffusion core. It uses those publish/subscribe APIs of directed diffusion, and shares the callback function, `recv()`. Event-based applications are implemented using four methods provided by the channel library: `Channel()`, `setupChannel()`, `sendData()`, and `receiveData()`. A typical event-based adaptive application consists of three types of components: adaptation server, event sensor, and action node. Although details of these components were explained in the Chapter 3, we will discuss here how they are implemented using the channel library in the next section.

4.2 Channel Library

4.2.1 Logical Channel

In the previous chapter, channels are logically classified as event channels or action channels. An event channel is used for communication between sensors and adaptation servers, whereas, an action channel is used for communication between action nodes and adaptation servers. An adaptation server on an event channel has a event queue to receive events from sensors and a reply queue to receive reply messages for actions. A request queue in action nodes store request messages sent from an adaptation server. All queues are of the first-in-first-out type.

All data sent from sensors regardless of data type, are stored at first in the event queue. An adaptation server is responsible for classifying the events and deciding to request actions. An adaptation server creates a channel by subscribing interest for the events and sensors publish data through the directed diffusion APIs and the BEA path. Similarly, action nodes subscribe interest to receive action requests and an adaptation server publish action requests. The message of published action request includes a request ID which is incremented by one for each new request from an adaptation server. Optionally, the request message may include an action type that specifies what kind of action an adaptation server requests and a node ID to perform the action. Using an

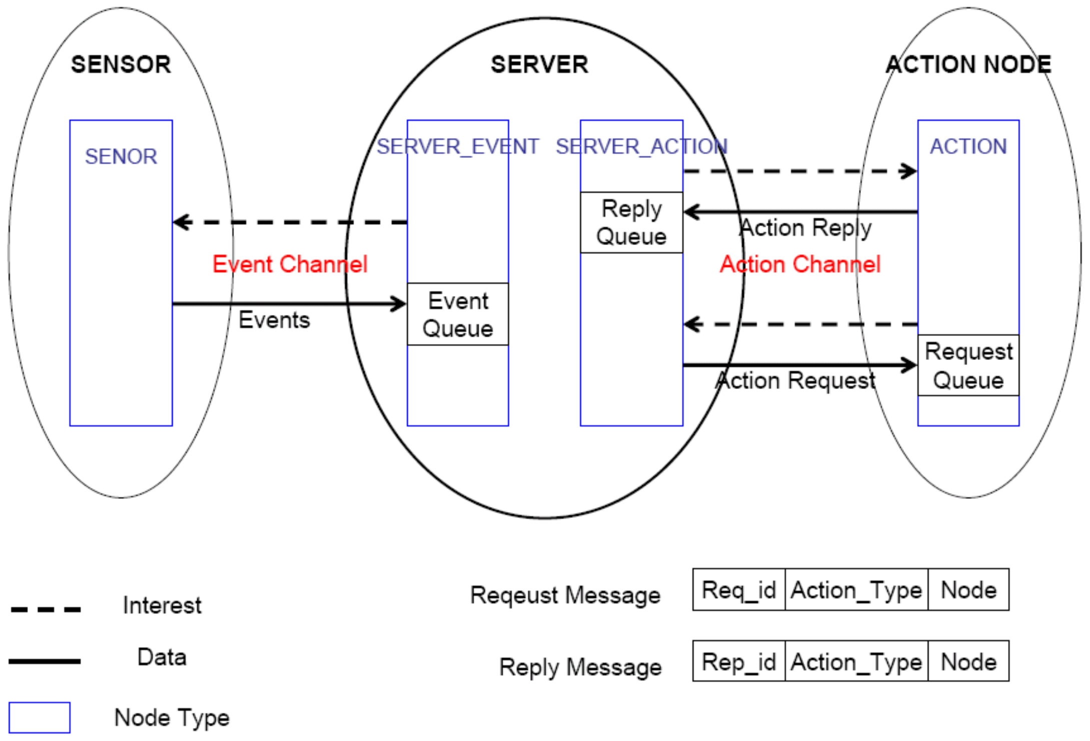


Figure 4.2: Channel Operation

action type and a node ID the right action nodes are selected, since only action nodes subscribe to action type and the node ID could receive the published request message. The reply message also includes an reply ID which is the same as the request ID that the action nodes received. In reply, an adaptation server subscribe interest to receive action reply messages and action nodes publish action reply messages.

4.2.2 Channel Library API

Channel library provides API for applications, particularly event sensors, an adaptation server, and action nodes. It is composed of *Channel* class (Appendix A) and implemented using directed diffusion application API discussed in Chapter 2. Appendix B shows a case example how each application is implemented using this APIs.

- `Channel(int argc, char **argv)`
  - The constructor for channel class.
  - The arguments could be "nodeinfo.txt" file specifying node's characteristics or "config.txt" file specifying node's network configuration [18].
  - This method parses command of the applications from the arguments and stores the information for the applications such as a latitude, longitude, diffusion port and node ID.
- `setupChannel(int node_type, NRAttrVec *input_list)`
  - `node_type` determines whether an application subscribes interest or publishes data. We provide four node types for creating *channel* objects in the implementation. Applications for sensors specify a node type, "SENSOR" that publishes events when they call `setupChannel()`. "SERVER_EVENT" type subscribes interest for receiving events from sensor and "SERVER_ACTION" type publishes action request and subscribes interest for getting action reply as specified in applications for adaptation servers. For the action channel, action nodes need to set the channel with a node type, "ACTION" that subscribes interest for action request and publishes action reply. In the case of an adaptation server, objects for two channels are created with node type, "SERVER_EVENT" for the event channel, and "SERVER_ACTION" for the action channel.

  - `input_list` allows applications to specify more characteristics for events or actions using attribute-value pairs. Channels are named by this set of attribute-value pairs. The real usage for `input_list` serves as an example in Section 4.3.

- `sendData(NRAttrVec *data_attr)`
  - it sends data with attributes, `data_attr`

- `receiveData()`
  - Whenever the node that subscribed interests receive data, the data is placed in an input queue which is the event queue, request queue or reply queue. In input queue, the least recently sent data is placed in the head and the most recently sent data is placed in the tail. This method indicates if there are data in input queue.

- `getData()`:
  - it returns the data at the front and the data is deleted in the input queue which is the event queue, request queue or reply queue.

4.3 Application Implementation

A great variety of applications with requirements and demands for adaptivity can be implemented using an adaptation service. Applications for our adaptation service may consist of sensors, an adaptation server, and action nodes that use the channel library APIs for communication.

Different types of applications for sensors can be implemented using different types of sensed event. They first starts by accessing a channel and defining the event type to send through the channel. Sensors need the module for sensing and collecting user-defined events. When the events are detected, they are sent, using `sendData()`. Our example in Appendix B is a sensor application that detect the events of a pressure and send it to an adaptation server periodically. Other sensor applications detecting and sending a different type of events could be implemented. The events in our example contain a timestamp used for calculating data delays in forwarding the events from the sensors to an adaptation server.

An adaptation server must create two channels: an event channel and an action channel. It receives events through an event channel, and decides to send action request triggered by the events. When it setups an event channel, it may specify various types of events it is interested in receiving. An adaptation server may specify a special group of actions or particular nodes to request actions on an action channel. Table 4.1 shows a simple procedure for an adaptation server. Each application for action nodes can specify a set of actions that it can serve in an adaptation service. For instance, when an adaptation server request an alarm action, only action nodes serving ALARM may reply to the request with their node ID. Action nodes serving other action could not reply to the request. Reply messages include reply ID which is the same as the request ID of the request message that the adaptation server sent.

Assume there are two types of sensors that detect the events of a temperature and a pressure in each, other sensors, and one adaptation server in a sensor network field (Figure 4.3). In the field, we have two action nodes that serves ALARM and REPORT.

```
proc Adaptation server
  setup event channel;
  setup action channel;
  req_id ⇐ 1;
  repeat
    need_action ⇐ false;
    receive data from event channel;
    if (event_received)
      Check events to see if the following action is necessary;
    if (need_action)
      request an action in action channel;
      increment req_id;
    receive action reply
    if (reply_received)
      check reply to see from where the reply comes;
  until END
```

Table 4.1: Adaptation server

If the adaptation server need to receive the events of temperature and pressure, the application for an adaptation server must specify both of events in the *input_list* and call `setupChannel()`. The adaptation server subscribes the interest on an event channel. Each sensor also sets up the channel for the events that it wants to send. The sensor that detects the events of a temperature call `setupChannel()` with an argument, *input_list* specifying the events of a temperature. The sensor that detects the events of a pressure performs the same procedure with the *input_list* specifying the events of a pressure. Then, sensors that detected each events send the events to a adaptation server along an event channel. The other sensors may not send any events on an event channel. An adaptation server also, sets up the action channel to receive actions of ALARM and REPORT by calling `setupChannel()`. The *input_list* for this channel includes the specification for the

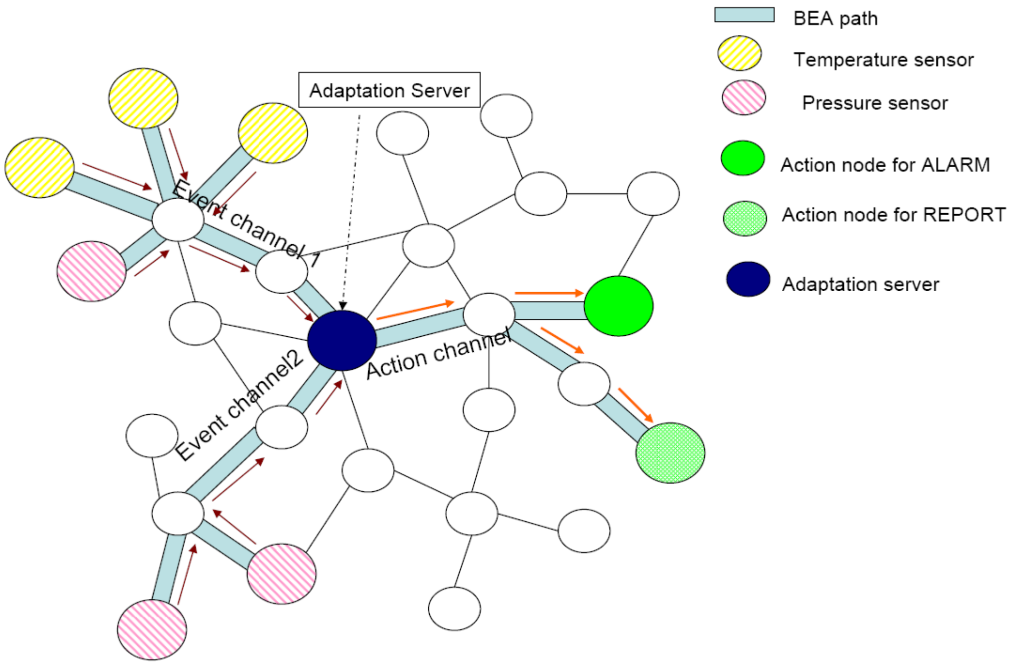


Figure 4.3: Example for adaptation service

actions of ALARM and REPORT. Each action node sets up the action channel using the action specification that it can serve. The action node that serves ALARM calls `setupChannel()` with *input_list* specifying the action of ALARM. The action node that serves REPORT calls `setupChannel()` with *input_list* specifying the action of REPORT. When an adaptation server receives all events that it is interested in, it decides if it needs to request either or both types of actions. The adaptation server sends action request messages along an action channel. The reply message from both action nodes also comes back along the action channel. The source code of our applications can be found in Appendix B.

4.4 Balanced Energy Aggregation Filter

The balanced energy aggregation filter is a filter to handle all exploratory messages. The implementation of this filter is based on the directed diffusion filter API described in Section 2.1. The BEA filter at each node contains important data structure for storing exploratory messages from all its neighbors and finding the lowest energy cost. The data structure which contains the neighbor ID that sent an exploratory data message, the energy cost of the message, and an exploratory data information is called *Energy Entry.* An *energy entry* is an entry in a hash table. This code uses the Tcl hash table, which is used in the directed diffusion core and the gradient filter. The hash key in a hash table for exploratory messages is created from a packet number and a packet ID of the exploratory data message. Each hash entry is composed of the *energy list* which is a linked list of energy entries. This hash table is capable of containing up to 100 entries. When the hash table reached the maximum size, it deletes the ten oldest hash entries.

Figure 4.5 shows the flow chart for processing the exploratory message when BEA filter receives an exploratory message. If the exploratory message comes from local host, this means this node is a source node. Thus, it sets the initial energy cost to 0 and forwards it to the gradient filter. Otherwise, it receives the exploratory message from a neighbor and computes its energy cost. Then, it creates a new `energy entry` with the energy cost and the neighbor ID that sent this exploratory message, as shown in Figure 4.4. If exploratory messages exists in the hash table, the BEA filter searches the *energy list* to find out if it has already received an exploratory message from that neighbor. If this is the first exploratory message from that neighbor, the BEA filter inserts the *energy entry* at the end of the *energy list.* If this is the first exploratory message received from

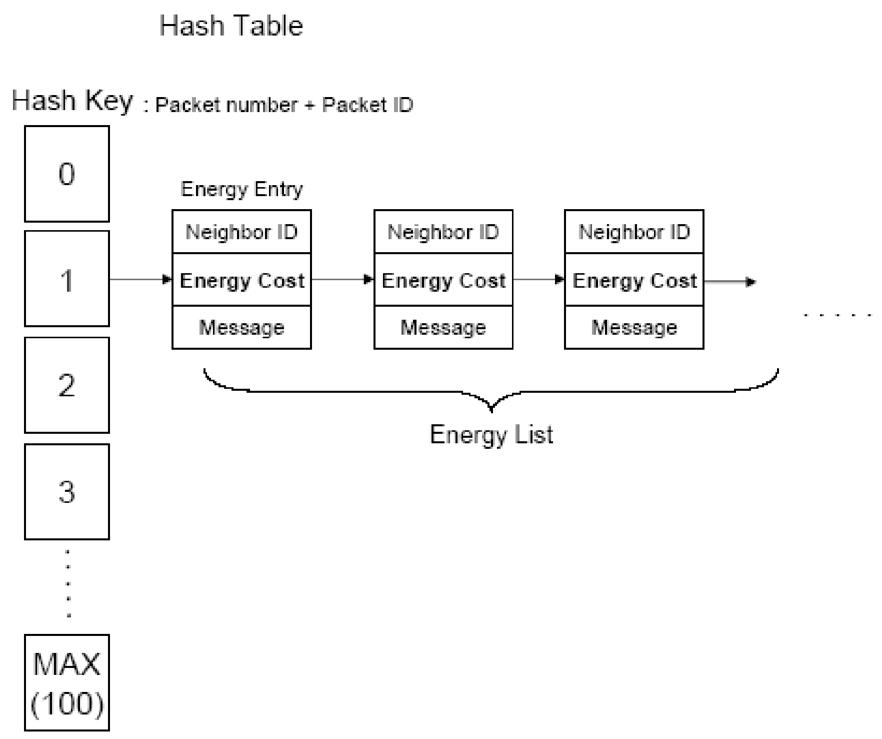


Figure 4.4: Hash table for containing exploratory messages

any neighbor, the BEA filter creates a new *hash entry* and start a exploratory timer to receive exploratory messages from all neighbors. The timeout values is based on the delay for forwarding data, as defined in the directed diffusion protocol. When the exploratory timer for exploratory messages from a source expires, it finds the exploratory message with a lowest energy cost and sends it to a gradient filter (Table 4.2).

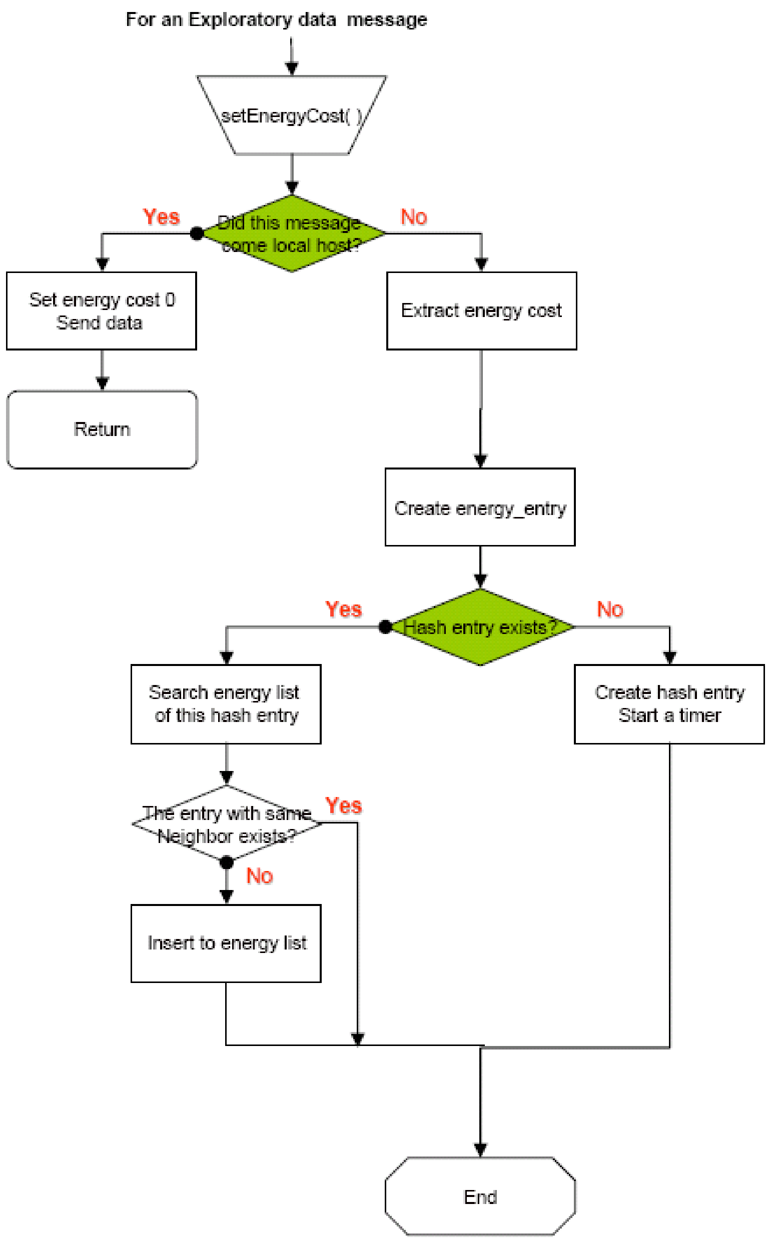

For an Exploratory data message
setEnergyCost( )
Did this message come local host?
Yes
No
Set energy cost 0 Send data
Return
Extract energy cost
Create energy_entry
Hash entry exists?
Yes
No
Search energy list of this hash entry
Create hash entry Start a timer
The entry with same Neighbor exists?
Yes
No
Insert to energy list
End

Figure 4.5: Flow chart for an exploratory message

```
proc FilterTimeout()
  move to the **energy_list** in the next hash entry;
  set first **energy_entry** to **min_energy_entry** in **energy_list**;
  while each **energy_entry** in **energy_list** do
      compare **energy_cost** of the **energy_entry** to **min_energy_entry**;
      if **energy_cost** is less than the **energy_cost** of **min_energy_entry**
      then
         **min_energy_entry** ← this **energy_entry**;
      endif;
  endwhile;

  **energy_cost** ← **energy_cost** of the **min_energy_entry**+ **extra_cost**;
  send the message of the **min_energy_entry**
end FilterTimeout
```

Table 4.2: After timeout of hash entry

4.5 Gradient Filter

The gradient filter plays the essential role of the interest flooding, gradient setup, data propagation, and reinforcement in directed diffusion. It processes messages of types: INTEREST, DATA, EXPLORATORY_DATA, POSITIVE_REINFORCEMENT, and NEGATIVE_REINFORCEMENT. It deals with new messages of each type though exploratory data from many neighbors could be duplicated. We need the BEA filter to process all exploratory messages from all neighbors before a gradient filter process only a new message. The BEA filter is thus set at a higher priority than the gradient filter.

Basically, the gradient filter maintains the routing table. Figure 4.6 shows a routing table. Routing entries are created by INTEREST and POSITIVE_REINFORCEMENT

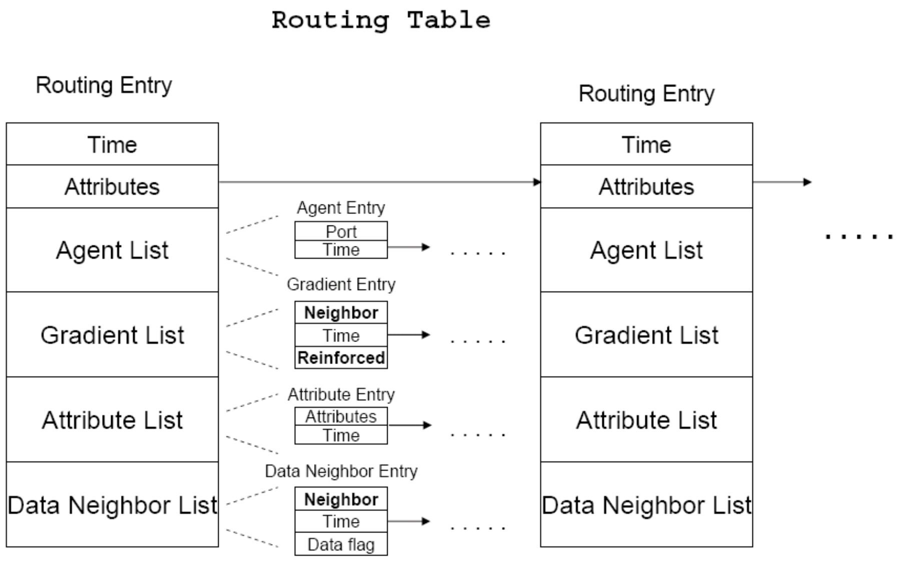


Figure 4.6: Routing Table in the gradient filter

messages. The gradient filter finds a routing entry that matches the DATA and EXPLORATORY_DATA messages. If messages are of the type INTEREST, it searches the routing table to find a matching routing entry and inserts a new routing entry if there is no matching routing entry. If the node that sent this interest is a sink, it updates an *agent list* by adding a new *agent entry* containing a port number on which the sink is running the directed diffusion. It also searches interest subscriptions in an *attribute list* and update the list. The *reinforced* flag of a *gradient list* could be set when a gradient filter receives POSITIVE_REINFORCEMENT messages. When data or exploratory messages come to a gradient filter, the attributes of the messages are examined against *attributes* of the routing entries to find matching routing entries according to the matching rule explained in Section 2.1. If there is a matching routing entry, it updates a *data neighbor list* of the matching routing entry with the neighbor ID that sent the data.

In addition to the basic functions of a gradient filter, we modified the gradient filter to send exploratory data messages with energy cost. A local rule for forwarding an exploratory data in the gradient filter is defined in Table 2.2. We need the following new lists (Figure 4.7) and methods to update the lists in order to find the neighbors that will forward that exploratory message without incrementing the energy cost. The *reinforced list* is the list of neighbors that was reinforced at least once. It is updated when positive

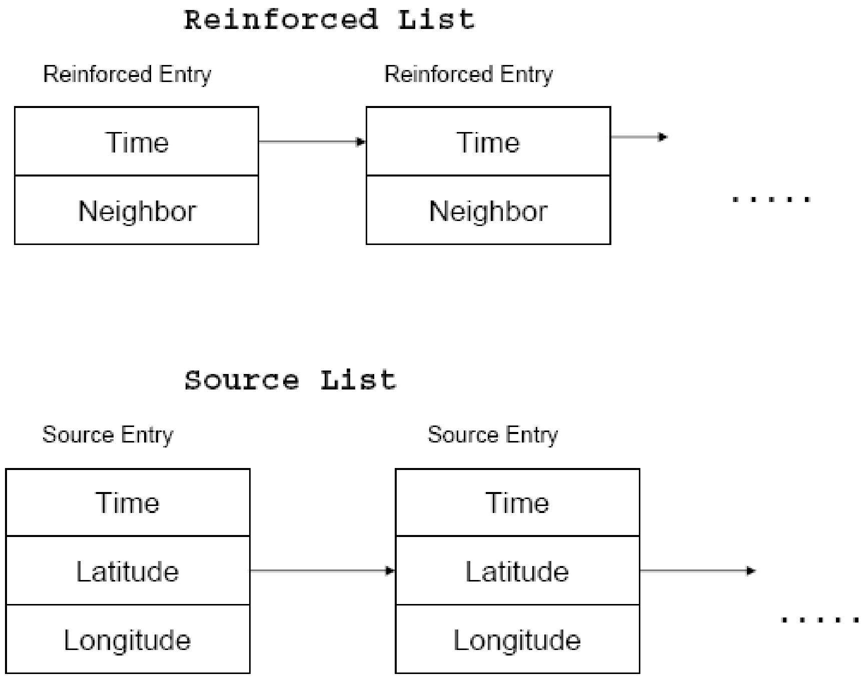


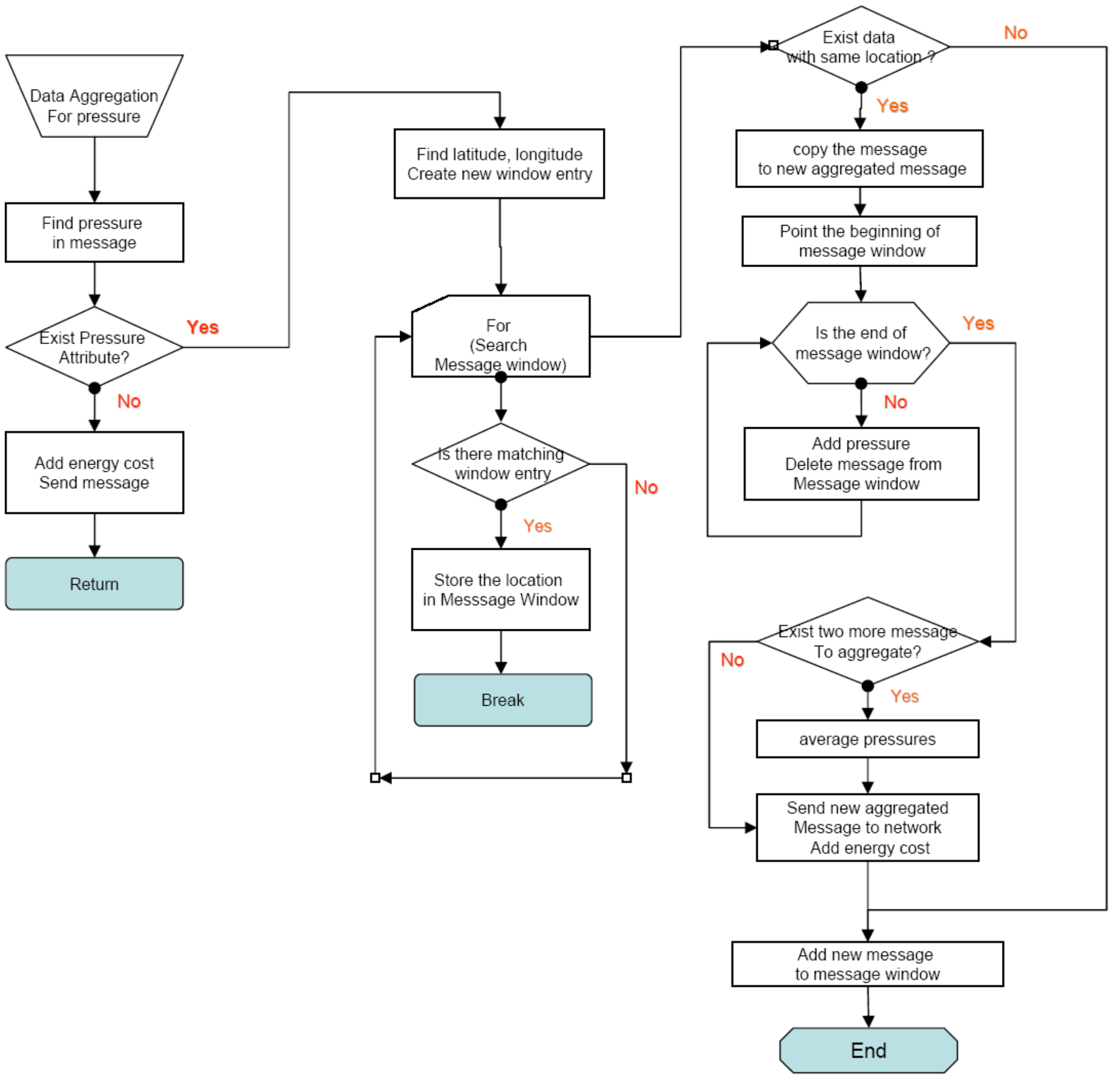
Figure 4.7: Reinforced list and source list added in the gradient filter

reinforcement messages are received. The *source list* contains the location information of sources that sent the events. If there are reinforced paths, they may not be shared paths. Neighbors on shared paths have received data from more than two sources. So, exploratory messages that do not increment the energy cost are forwarded to neighbors on shared paths.

- `void updateReinforcedList(int32_t last_hop)` :
  - last_hop is the neighbor ID that sent the positive reinforcement message.
- `bool checkSourceList(float latitude, float longitude)` :
  - latitude and longitude are the location of sources that sent the events and obtained from message attributes
  - it returns true or false, depending on whether there are information about two or more sources in the source list, otherwise it returns false.
- `void updateSourceList(float latitude, float longitude)`:
  - it updates the sources list

4.6 Data Aggregation Filter

Data aggregation filter is application dependant. It is depending on the target application. For example, suppose that in a controlled pressure environment, the data aggregation filter monitors the pressure value and average them. However, the data aggregation filter could not average other value (i.e. temperature). When it receives other value, it forwards the value without any aggregation to the network. Our aggregation filter averages the pressure readings from several sensors. We create the message window to keep track of all data messages. The entries of the message window has the latitude and longitude information of data sources in order to compare and find data messages from different sources. The data aggregation filter searches the message window. If there exists data messages from the same sources, it adds the pressure values from the beginning of the message window to end of message window and averages it. Aggregated data message are deleted and data messages that newly arrive at the data aggregation filter is inserted at the front in the message window. Figure 4.8 shows more details of the algorithm.



Data Aggregation For pressure
Find pressure in message
Exist Pressure Attribute?
Yes
No
Add energy cost Send message
Return
Find latitude, longitude Create new window entry
For (Search Message window)
Is there matching window entry
No
Yes
Store the location in Messsage Window
Break
Exist data with same location ?
No
Yes
copy the message to new aggregated message
Point the beginning of message window
Is the end of message window?
Yes
No
Add pressure Delete message from Message window
Exist two more message To aggregate?
No
Yes
average pressures
Send new aggregated Message to network Add energy cost
Add new message to message window
End

Figure 4.8: Flow for data aggregation

Chapter 5
PERFORMANCE EVALUATION

In this chapter, we discuss the implementation environment and the results from performance tests. We describes our methodology and compares the performance of the balanced energy aggregation against the path sharing aggregation and the greedy aggregation.

5.1 Development Environment

All implementations are developed using Directed Diffusion code, diffusion-3.2.0 that is provided by ISI (the Information Systems Institute). Since Diffusion-3.2.0 has been implemented using C++, our implementations which access the dynamic libraries compiled from C++ code of Diffusion-3.2.0 are also implemented using C++. Applications that implement sensors, adaptation servers, and action nodes using our channel library, may be implemented in C++.

The testing environment could support emulation or real-world testing. The testing environment could involve real nodes or nodes that closely mimic the real nodes. Emulation allows for the running of target code on non-target hardware that may be more closely controlled and monitored. All our tests are emulated on 48 Linux-based machines. The machines used are faster than the actual nodes (sensors) in the field. Each is 400Mhz with 128 megabytes of ram and has 100Mbps wired connections with three network switches. Though each test machine can run multiple virtual directed diffusion nodes, we mapped one machine to one virtual node.

5.2 Experimental Design

We select four metrics for analyzing the performance of the balanced energy aggregation and comparing it to the path sharing aggregation and the greedy aggregation: average dissipated energy, average delay, system lifetime, and the number of events. The average dissipated energy and average delay were used in earlier work to compare the opportunistic aggregation of the directed diffusion with the greedy aggregation [12]. The metric of system lifetime were defined to show the efficiency of Maximum Residual Energy Path (MREP) routing protocol [21]. **Average dissipated energy** measures the ratio of total dissipated energy per node in the network to the number of distinct events received by sinks. This metric computes the average work done by a node in delivering information to the sinks. **Average delay** measures the average one-way latency observed between transmitting an event and receiving it at each sink. **System lifetime** defined as the time until the first node in the network runs out of energy. Since we want to balance the network traffic as well as achieve substantial aggregation efficiency, the lifetime of each sensor node should be increased. **The number of events** measures the number of distinct events received by sinks during the system lifetime. These metrics which is system lifetime and the number of events indicates the overall lifetime of network.

Theoretical results in Krishnamachari et al 's paper [11] shows that the greatest gains due to data aggregation are obtained when the sources are close together and far away from the sink. Thus, our network configuration is intended for the high-level data aggregation and, also maximizing the system lifetime in our Balanced Energy Aggregation. For studying the impact of the number of sources, we use at least 25 nodes in the sensor network. The number of sources ranges from 2 to 8 sources in increment of 1. We use only one sink. The network configuration shapes like one-eighth pie. All sources are randomly selected from nodes at the left side of the pie and the sink is selected from nodes at the right side of the pie. One of sensors in the left side and the sink forms an existing path and the intermediate nodes are scattered around the path. We also, studied five different network configuration in order to evaluate how network density affects the performance. The network size ranges from 10 to 32 nodes in increments of 5 nodes and the average number of neighbors ranges from 3 to 7. All network configuration in this test have 2 sources and 1 sink.

Events in each source is generated every two seconds. Thus, the aggregation delay is set to 2 seconds. Event messages are modeled as 124 byte packets, including location information (longitude and latitude) of the sources, event value of the pressure, and the header information for the directed diffusion. Since our test performs perfect aggregation, the aggregate event message size is 124 bytes. The threshold for evaluating the impact of the number of sources is 50%. When the remaining energy is less than 50% of the initial energy, the node could be considered overloaded. When the remaining energy is one tenth of threshold, 5%, we regard the node as a dead node. On the other hand, we increased the threshold (70%) for testing in variable network size. By choosing high threshold, we could expect higher system lifetime.

5.3 Evaluation

5.3.1 Impact of the Number of Sources

We evaluate three types of data aggregation algorithm discussed before, namely Greedy Aggregation(GA), Path Sharing Aggregation(PSA), and Balanced Energy Aggregation(BEA). Figure 5.1 shows the system lifetime as a function of the number of sources. Assuming that data are aggregated at one aggregation point near the sources, the balanced energy aggregation has much more lifetime than other algorithms. More network traffic go through the node at the aggregation point and as a result, the node is over-loaded. When the residual energy is less than the threshold, BEA chooses another path. It could reduce the data message going through the previous aggregation point, (except broadcasting message such as exploratory message and interest), and increase

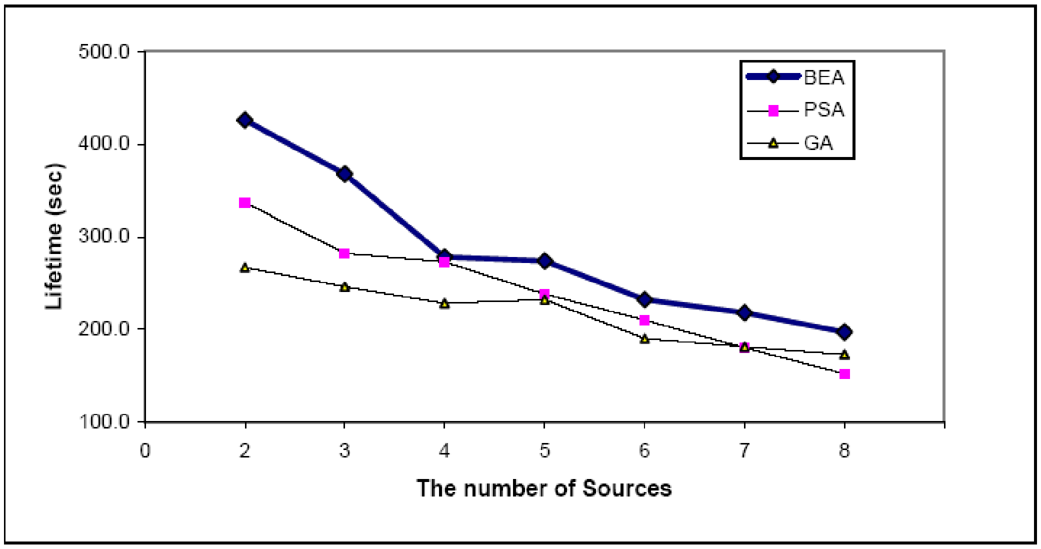


Figure 5.1: System lifetime with variable number of sources

the lifetime of the node. However, path sharing aggregation and greedy aggregation could not change the path before the node is dead. Obviously, the balanced energy aggregation achieves higher system lifetime than the other two algorithms. Comparing the path sharing aggregation with the greedy aggregation, the greedy aggregation that introduces extra incremental messages shows lower system lifetime. The system lifetime for all approaches decreases with the number of sources due to increased network traffic.

As expected, the graph of the number of events received by the sink during the system lifetime shows that BEA has the highest number of events, followed by PSA (Figure 5.2). The number of events also decreases with the number of sources.

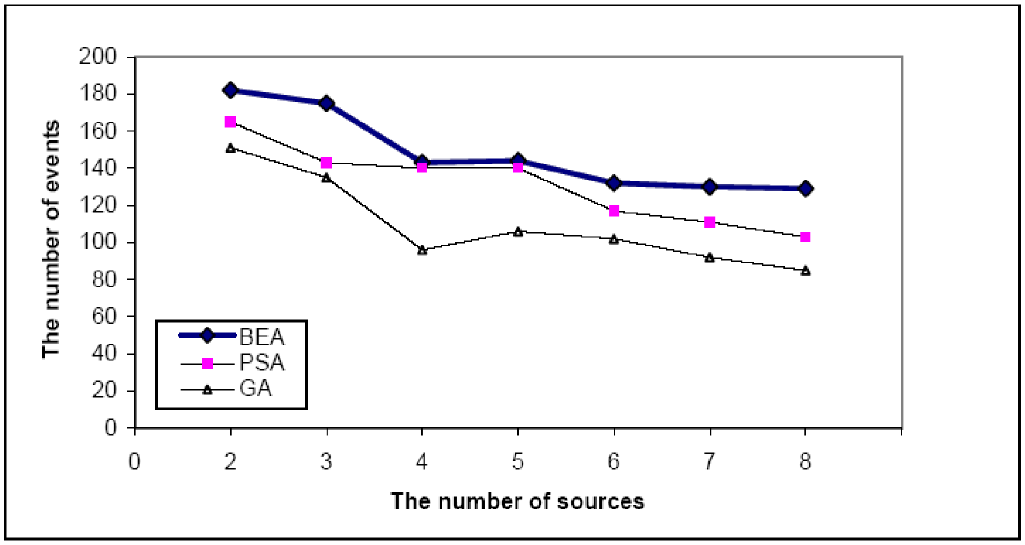


Figure 5.2: The number of Events during the network lifetime with variable number of sources

Figure 5.3 plots the average dissipated energy observed as a function of the number of sources. Although the balanced energy aggregation increase the system lifetime, it achieves this at the expense of increasing the average dissipated energy in the network. However, the increase in average dissipated energy is only slightly higher than PSA and is still generally lower than GA. It is able to do this by reducing the number of transmissions and reducing the number of nodes involved. When nodes sleep, we can save as much as 90% of the energy. Since the dissipated energy is defined as the total dissipated energy per node, the balanced energy aggregation achieves an increase in the system life through the distribution of loads, but does not try to save energy. Since the path sharing aggregation shares paths at the closest point to the sources, it encourages early sharing of paths and reduces energy consumption. The balanced energy aggregation may not perform early aggregation, if the residual energy of the aggregation node falls below the threshold. Despite early path sharing of the greedy aggregation, the incremental messages of the greedy aggregation increase the average dissipated energy.

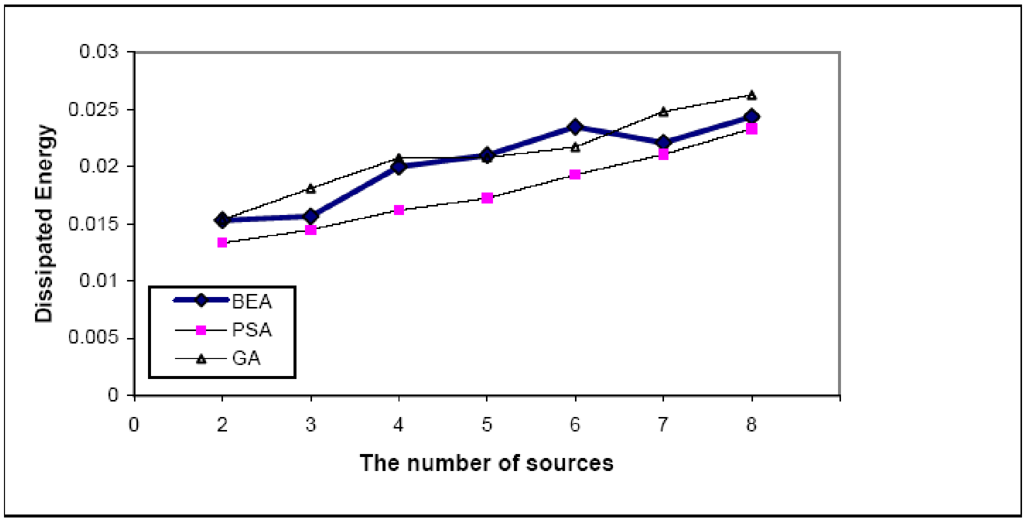


Figure 5.3: Average dissipated energy with variable number of sources

Figure 5.4 shows the average delay as a function of the number of sources. It seems that the average delay may not be different for each aggregation algorithm. This is because all algorithms favor an energy efficient shared path rather than a shared path. When an aggregation node is overloaded, the shared path in BEA would be switched to another path with a new aggregation node. However, the new aggregation node, that is the next closest node near the sources, is still on the shared path and that shared path may not also be the shortest path. Though an average delay in each algorithm shows little difference, we can see the relation between the average dissipated energy and the average delay in all the aggregation algorithms we have compared, as shown in Figure 5.5. The average dissipated energy is inversely proportional to the average delay in all, i.e. the lower the average dissipated energy, the higher the delay of events from the sources to the sinks. This explains why our data aggregation function described in Section 3.3 is efficient in reducing delay.

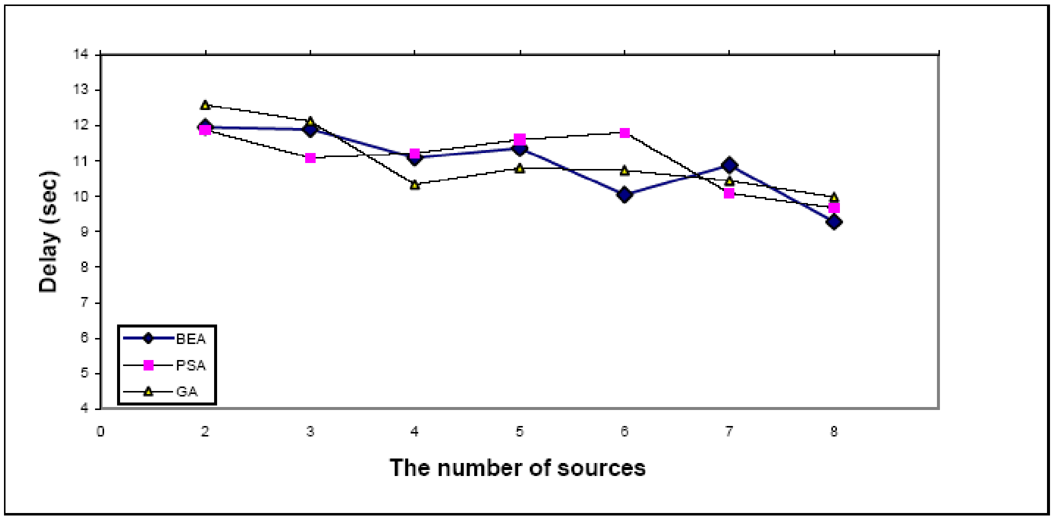
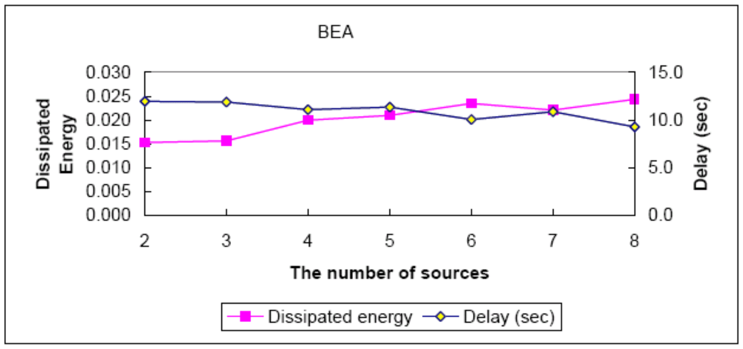
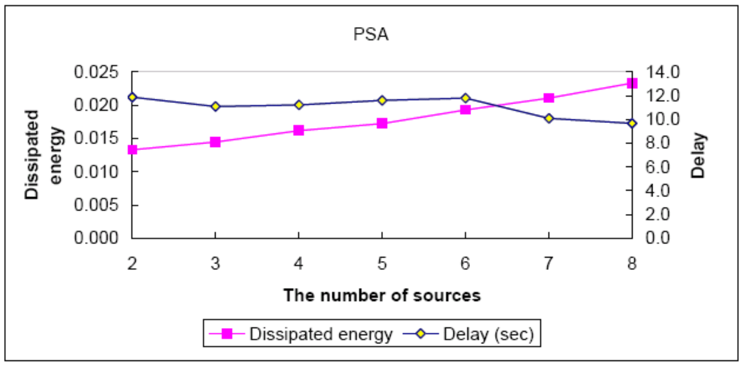


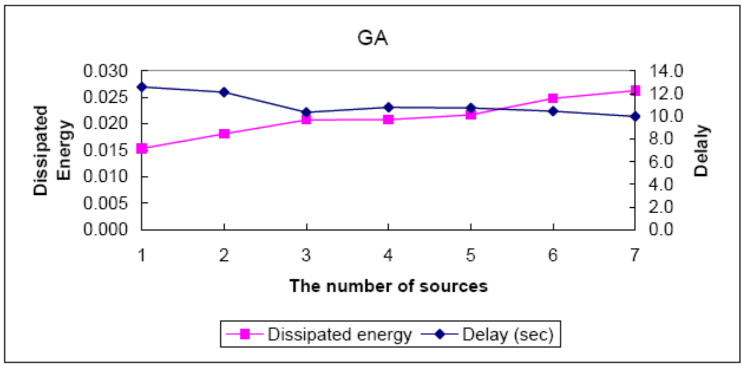
Figure 5.4: Average Delay with variable number of sources



(a) Balanced Energy Aggregation



(b) Path sharing Aggregation



(c) Greedy Aggregation

Figure 5.5: The relations between dissipated energy and delay

5.3.2 Impact of Network Size

Comparisons of the performance with variable network size shows that the balanced energy aggregation has much higher system lifetime than the other two algorithms (Figure 5.6). We used higher threshold(70%) than in the previous experiments for comparisons with variable number of sources (Figure 5.1). The balanced energy aggregation changes the shared path when the residual energy rate in a node at an aggregation point is below 70%. Path switching happens earlier than when the threshold is 50%. Thus, we could save more energy of the aggregation node and increase the lifetime of that node. After the residual energy rate reaches the threshold, it introduces an extra cost which is grater than 1. The extra cost is increased as the residual energy rate is decreased. The increased extra cost increase the energy cost of the neighbor nodes close to sources. When many neighbors near the sources have high energy cost, the source may choose a longer delay path. Balanced energy aggregation results in longer average delay for delivering data from the sources to the sink (Figure 5.7). At a threshold of 70%, we also observed that some node that is not on the original existing path is the first to die, particularly when the network size (the number of neighbors) is increased. Though the nodes on an existing path initially have higher chance of losing their energy than other nodes, BEA may switch path and force the other nodes that are not on the original existing path to forward the data.

As expected, the number of events received by the sink during the system lifetime in BEA is higher than PSA and GA (Figure 5.8). This is because BEA has the higher lifetime than other two algorithms. Also, the average dissipated energy of BEA is slightly higher than PSA and is lower than GA (Figure 5.9).

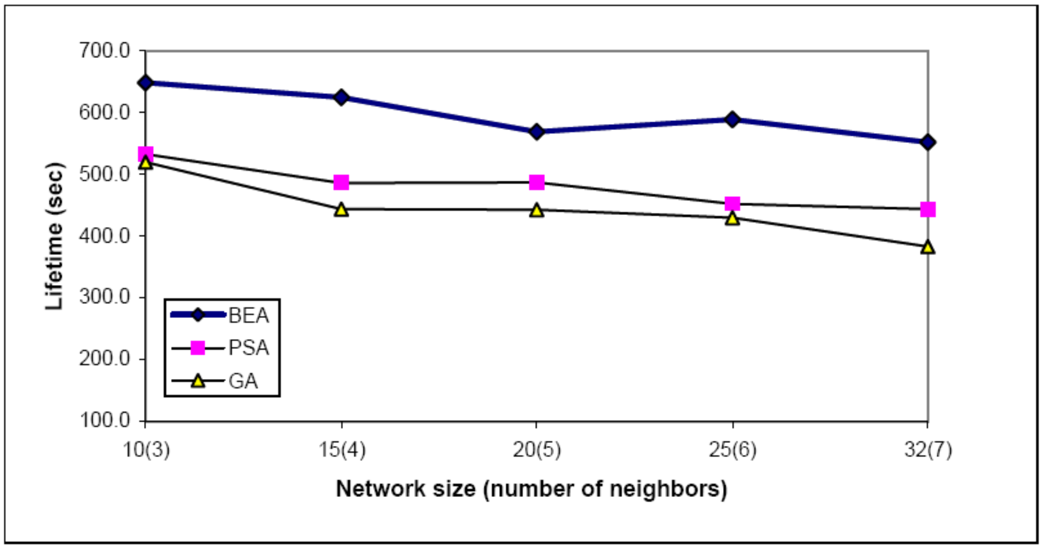


Figure 5.6: System lifetime with variable network sizes

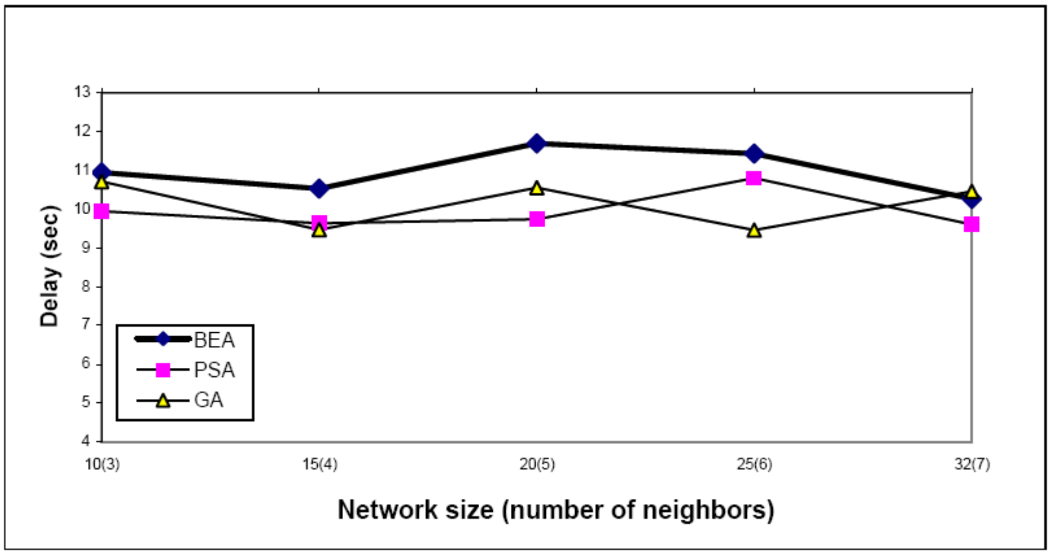


Figure 5.7: Average delay with variable network sizes

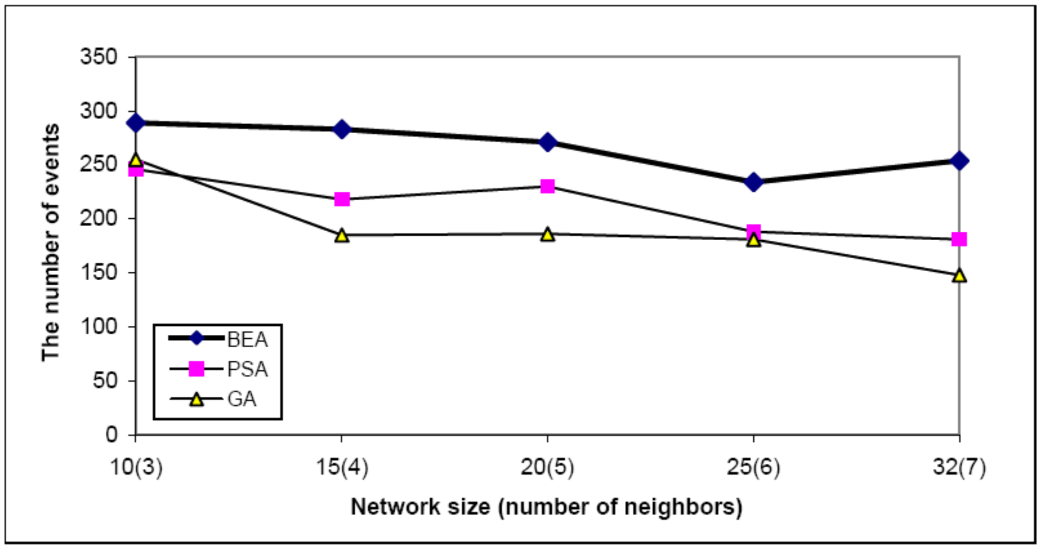


Figure 5.8: The number of events with variable network sizes

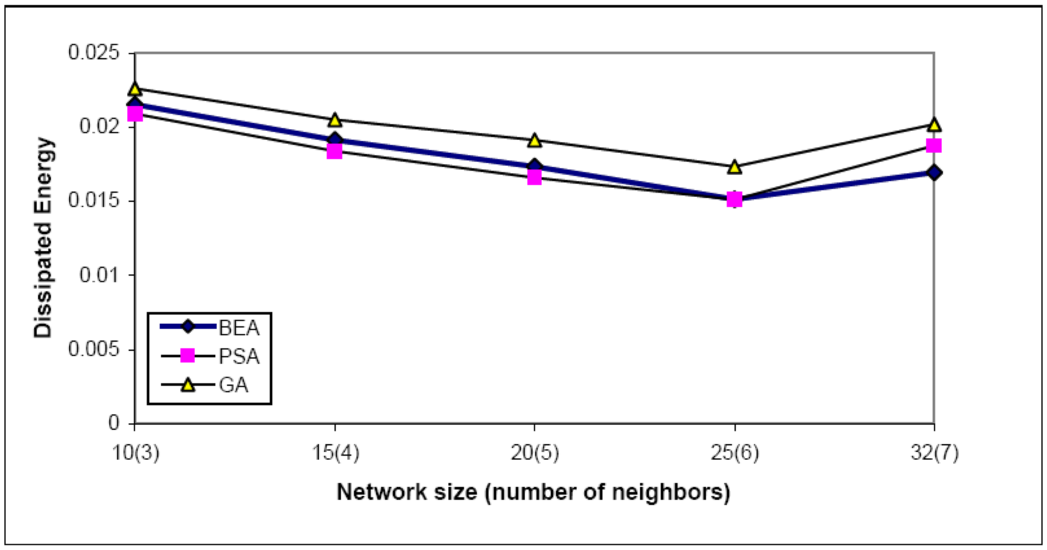


Figure 5.9: Average dissipated energy with variable network sizes

Chapter 6

CONCLUSIONS AND FUTURE WORK

We presented an adaptation service underlying an event-based communication and data aggregation algorithms. Our adaptation service delivers distributed events and react to changes in an environment. It consists of three components: event sensors, adaptation servers, and action node. An event sensor is responsible for collecting and sending events to an adaptation server. An adaptation server receives the events and sends action requests to an action node. An action node executes the requested actions. Events and actions need to be disseminated and routed in ways that increase energy efficiency and network lifetime of the sensor networks. We proposed a novel approach to deliver events or actions by employing a balanced energy aggregation that maximizes the lifetime of the sensor network while limiting the energy consumption. It use mechanisms to save energy through the distribution of loads that switch paths from the sources to the sinks. Results of our evaluations from the experiments shows that the balanced energy aggregation prolonged the network lifetime by balancing loads, compared to the path sharing aggregation and greedy aggregation.

Our programming model is based on a concept of channels that encapsulates properties of the underlying sensor network communication system accessible to an application. Channels was classified into event channels and action channels by their attributes.

There are several outstanding issues that have to be addressed in future research work. First, our balanced energy aggregation need to be tested in various network sizes. In a high-density network with more neighbors, the balanced energy aggregation will be more effective in balancing loads, resulting in an increase in the overall network lifetime of the sensor network. Second, our data aggregation function does not guarantee data accuracy. It was used just as a simple tool for demonstrating energy efficiency and implementing a perfect aggregation whereby the data size of an aggregate is equal to the data size of an individual event. Future work is needed to design an aggregation function that supports data accuracy as well as energy efficiency when it is implemented in complex applications.

In the future, applications of our adaptation service will be implemented to enable adaptation in a more practical environment. We need to design experiments to evaluate more aspects of an adaptation service, such as contributions of methods for link failure detection and animal habitat monitoring. Also, we need to come up with a generic communication protocol to synchronize between all the distributed servers such as lookup servers, composition servers [15], and adaptation servers. Currently, we assume only one adaptation server is used for an adaptation service. A more effective adaptation service will consists of lookup servers, composition servers and adaptation servers, collaborating together to achieve some global and distributed adaptation objectives. The distributed service architecture needs to avoid a centralized control.

Bibliography

[1] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva, "Directed Diffusion for Wireless Sensor Networking," IEEE/ACM Transactions on Networking, Vol. 11, No. 1, February 2003.

[2] Dan Coffin, Dan Van Hook, Ramesh Govindan, John Heidemann, Fabio Silva, "Network Routing Application Programmer's Interface (API) and Walk Through 9.1", July 12, 2003.

[3] Tsung-Han Lin; Huang, P., "Sensor data aggregation for resource inventory applications," Wireless Communications and Networking Conference, 2005 IEEE Volume 4, 13-17 March 2005.

[4] Gupta, G., Younis, M., "Load-balanced clustering of wireless sensor networks Communications," 2005 IEEE International Conference on Volume 3, Issue , 11-15 May 2003.

[5] Tsung-Han Lin, Huang, P. "Sensor data aggregation for resource inventory applications," Wireless Communications and Networking Conference, 2005 IEEE Volume 4, 13-17 March 2005.

[6] Mitchell A. Cohen, Jakka Sairamesh, Mao Chen, "Reducing business surprises through proactive, real-time sensing and alert management," Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services EESR '05, Pages: 43 - 48, June 2005.

[7] K. Mani Chandy, Brian Emre Aydemir, Elliot Michael Karpilosky, S.M. Zimmerman, "Event Webs for Crisis Management," IASTED Conference, 2003.

[8] Chandy, K. M., Aydemir, B. E., Karpilovsky, E. M., and Zimmer- mann, D., "Event-driven architectures for distributed crisis management," 15th IASTED International Conference on Parallel and Distributed Computing and Systems, November 2003.

[9] Meier, R., Cahill, V., "STEAM: event-based middleware for wireless ad hoc networks," Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on 2-5 July 2002.

[10] Eisenhauer, G., Bustamante, F.E., Schwan, K., "Event services for high performance computing," High-Performance Distributed Computing, 2000. Proceedings. The Ninth International Symposium on 1-4 Aug. 2000.

[11] B. Krishnamachari, D. Estrin, and S. Wicker. "The Impact of Data Aggregation in Wireless Sensor Networks," In Proc. of Intl. Workshop on Distributed Event-Based Systems, 2002.

[12] Intanagonwiwat, C., Estrin, D., Govindan, R., Heidemann, J., "Impact of network density on data aggregation in wireless sensor networks," Distributed Computing Systems, 2002. Proceedings. 22nd International Conference on 2-5 July 2002.

[13] Dasgupta, K., Kalpakis, K., Namjoshi, P., "An efficient clustering-based heuristic for data gathering and aggregation in sensor networks," Wireless Communications and Networking, 2003. WCNC 2003. 2003 IEEE Volume 3, 16-20 March 2003.

[14] Alvin Lim, "Distributed Services for Information Dissemination in Self-Organized Sensor Networks," Special Issue on Distributed Sensor Networks for Real-Time Systems with Adaptive Reconfiguration, Journal of Franklin Institute, Elsevier Sciences Publisher, vol. 338, 2001, pp. 707-727.

[15] Xuan Yu, "Implementation of Lookup Service Protocols for Self-Organizing Sensor Networks," Master Thesis, Auburn University, 2001.

[16] Ye Wang, "A Dynamic Adaptation Service Framcework in Self-Organizing Sensor Networks," Master Thesis, Auburn University, 2002.

[17] Qiao Shen, "Energy Consumption Improvements of Sensor Networks Using Directed-Diffustion Protocol," Master Thesis, Auburn University, 2004.

[18] Mark Ivester, "Interactive and Extensible Runtime Framework for Execution and Monitoring of Sensor Network Services," Master Thesis, Auburn University, 2005.

[19] Wendi R. Heinzelman, Hoanna Kulik, and Hari Balakrishnan, "Adaptive protocols for information dissemination in wireless sensor networks," In proceedings of the ACM/IEEE International Conference on Mobile Computing and Networking, pp. 175 185, August, 1999.

[20] Ignacio Solis, Katia Obraczka, "The impact of timing in data aggregation for sensor networks," Communications, 2004 IEEE International Conference on Volume 6, 20-24 June 2004.

[21] Qiling Xie, Chin-Tau Lea, Golin, M.J., Fleischer, R., "Maximum residual energy routing with reverse energy cost," Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE Volume 1, 1-5 Dec. 2003.

Appendices

Appendix A

class Channel

```
class Channel {
    uint16_t    port_;
    int nodeid_;
    float   latitude_;
    float   longitude_;

    NR  *mdr_;
    handle  subHandle_;
    handle  pubHandle_;
    int num_subscriptions_;
    ChannelReceive  *recvCallback_;
    bool    event_received_;

    queue<NRAttrVec *> input_queue_;

public:
    Channel(int argc, char **argv);
    void setupChannel(int type, NRAttrVec *attrs);

    void recv(NRAttrVec *data, NR::handle my_handle);
    handle setupSubscription(NRAttrVec *attrs);
    handle setupLocalSubscription(NRAttrVec *attrs);
    handle setupPublication(NRAttrVec *attrs);

    bool sendData(NRAttrVec *data);
    bool receiveData();

    int getNodeid();
    NRAttrVec* getData();
}
```

Appendix B

Example of the Adaptation Service

B.1 Example for Sensor : pressure_sensor.cc

```
int main(int argc, char **argv) {
    NRAttrVec   input_list;
    NRAttrVec   event_list;
    SensedData  event;
    SensorApp   *sensor;
    Channel     *sensor_channel;

    time_t      now;
    struct timeval  tv;
    NRSimpleAttribute<void *>   *timeAttr;
    NRSimpleAttribute<float>    *pressureAttr;

    // create or open new channel
    sensor_channel = new Channel(argc, argv);
    input_list.push_back(TargetAttr.make(NRAttribute::EQ,
                    TARGET_PRESSURE));
    sensor_channel->setupChannel(SENSOR, &input_list);

    // For check latency
    timeAttr = STimeStampAttr.make(NRAttribute::IS, &tv,
                sizeof(struct timeval));
    event_list.push_back(timeAttr);
    pressureAttr = AppPressureAttr.make(NRAttribute::IS, 0.0);
    event_list.push_back(pressureAttr);

    sensor = new SensorApp
        ("/home/kimeunk/todo/adapServ-2.0.0/test/sensed_data.txt");
    while (1) {
        event = sensor->detection();
        if (event.valid()) { // event detected
            pressureAttr->setVal(event.value); // from Diffusion API
            gettimeofday(&tv, NULL);
            timeAttr->setVal(&tv, sizeof(struct timeval));
            if (sensor_channel->sendData(&event_list)) {
                DiffPrint(DEBUG_ALWAYS, "----------->Sending Data
                    %f\t%d.%d\n", event.value, tv.tv_sec, tv.tv_usec);
                sensor->deleteSensedEvent();
            }
        } else {
            DiffPrint(DEBUG_ALWAYS, "no event!\n");
        }
        sleep(2); // event rate: 2 secs/event
    }

    return 0;
}
```

B.2 Example for Server : pressure_server.cc

```
// to handle with multiple events and actions,
// you should follow operator and valiable
int main(int argc, char  **argv) {
    Channel *event_channel, *action_channel;
    bool event_received;
    bool need_action;
    bool reply_received;;
    int req_id;
    NRAttrVec input_list;
    NRAttrVec action_request;
    NRSimpleAttribute<int> *requestidAttr;


    event_channel = new Channel(argc, argv);
    input_list.push_back(TargetAttr.make(NRAttribute::IS,
                    TARGET_PRESSURE));
    // To get multiple events..specify more
    //input_list.push_back(TargetAttr.make(NRAttribute::IS,
                    TARGET_PING));
    event_channel->setupChannel(SERVER_EVENT, &input_list);

    ClearAttrs(&input_list);
    // create or open new channel for action_request
    action_channel = new Channel(argc, argv);
    // request action to a Group
    input_list.push_back(ActionAttr.make(NRAttribute::EQ, ALARM));
    // request action to a particular node in the Group
    //action_request.push_back(ActionNode.make(NRAttribute::EQ, 2007));
    action_channel->setupChannel(SERVER_ACTION, &input_list);


    if ((fp = fopen("energy_delay", "w")) == NULL) {
        DiffPrint(DEBUG_ALWAYS, "open error Delay file\n");
    }

    req_id = 1;
    requestidAttr = ActionRequestID.make(NRAttribute::IS, req_id);
    action_request.push_back(requestidAttr);
    while (1) {
        need_action = false;
        event_received = event_channel->receiveData();
        if (event_received) {
             need_action = check_event(event_channel);
        }

        if (need_action) {
            if (action_channel->sendData(&action_request))  {
                DiffPrint(DEBUG_ALWAYS, "########-----> Send
                [ACTION REQUEST] with req_id, %d\n", req_id);
                requestidAttr->setVal(++req_id);
            }
        }
        reply_received = action_channel->receiveData();
        if (reply_received) {
            check_reply(action_channel);
        }

        sleep(1); // polling : depend on user application
    }

    fclose(fp);
}
```

B.3 Example for Action node : pressure_action.cc

```
void check_request(Channel *channel, NRAttrVec &action_reply) {
    NRAttrVec *data;
    NRSimpleAttribute<int> *nodeAttr;
    NRSimpleAttribute<int> *requestidAttr;
    NRSimpleAttribute<int> *replyidAttr;
    int req_id;

    while (data = channel->getData())
    {
        requestidAttr = ActionRequestID.find(data);
        if (requestidAttr)
        {
            req_id = requestidAttr->getVal();
            DiffPrint(DEBUG_ALWAYS, "########----->
                RECEIVED [ACTION REQUEST],%d from server\n", req_id);
        }

        replyidAttr = ActionReplyID.find(&action_reply);
        replyidAttr->setVal(req_id);
        if (channel->sendData(&action_reply))
        {
            DiffPrint(DEBUG_ALWAYS, "-----> Action node:
            SENDS [ACTION REPLY], %d to server. \n", req_id);
        }
    }
}

int main(int argc, char  **argv) {
    Channel *action_channel;
    bool request_received;
    NRAttrVec input_list;
    NRAttrVec action_reply;

    action_channel = new Channel(argc, argv);
    // specify Action Group
    input_list.push_back(ActionAttr.make(NRAttribute::IS, ALARM));
    // specify this node in Action Group
    input_list.push_back(ActionNode.make(NRAttribute::IS,
                        action_channel->getNodeid()));
    action_channel->setupChannel(ACTION, &input_list);
    action_reply.push_back(ActionReplyID.make(NRAttribute::IS, 0));
    while (1)
    {
        request_received = action_channel->receiveData();
        if (request_received)
        {
            check_request(action_channel, action_reply);
        } else {
            sleep(1); // polling : depend on user application
        }

    }
}
```