UNIVERSITY OF CALIFORNIA
SANTA CRUZ

EFFICIENT PROTOCOLS FOR POWER-CONSTRAINED HETEROGENEOUS WIRELESS AD-HOC NETWORKS

A dissertation submitted in partial satisfaction of the
requirements for the degree of

Doctor of Philosophy

in

COMPUTER ENGINEERING

by

Ignacio Solis

September 2005

The Dissertation of Ignacio Solis
is approved:

Professor Katia Obraczka, Chair

Professor J.J. Garcia-Luna-Aceves

Professor Roberto Manduchi

Robert C. Miller
Vice Chancellor for Research and
Dean of Graduate Studies

Contents

List of Figures

List of Tables

Abstract

Efficient protocols for power-constrained heterogeneous wireless ad-hoc networks

by

Ignacio Solis

With networks permeating every corner of the technological spectrum the requirements expected from the them have expanded. They haven't grown in the sense of more features, they have grown in the sense of more efficient specialized features. The protocols that were once well suited for all our network needs in the past are not well suited for all our network needs in the future.

These heterogeneous environments have raised a number of challenges for the network society. Interconnection between powerful and week devices needs to be efficient. The omnipresent IP [18] protocol can't deal efficiently with all the possible scenarios. At the lower end of the heterogeneous spectrum we have a set of application and devices that need to be carefully optimized. The proliferation of ubiquitous scenarios have made standardization a sought after path. If the trend of optimizing protocols for unique applications continues, the different clouds of devices won't be able to talk to each other.

We have developed FLIP, a FLexible Interconnection Protocol in charge of connecting devices of varying capabilities. This protocol is well suited to connect powerful devices with all the current capabilities provided by IP [18]. It is also well suited to connect power-anemic devices, like the ones used for sensor networks. For this scenario we have also developed power efficient methods of data collection.

Our Cascading Timeouts reduce delay in the collection of sensed data from a sensor network. We have also designed a scheme for efficient data aggregation based on isoclusters, that is, clusters of nodes that have been grouped because they have a similar sensed value.

Keywords: Sensor Networks, Data Aggregation, Energy Efficient Protocols

Acknowledgments

This thesis has taken a lot of time and effort to put together, and even though I would like to take credit for most of that, none of it would have been possible without the support of a lot of the people around me.

It should be obvious from the fact that she was able to put up with me all these years that she deserves a big thanks on my part. Thank you Katia for giving me guidance when I needed it (she knows I've needed it) and for supporting me academically and personally.

There are many people at school that deserve my sincere thanks. Kumar has been with me since the beginning, including a change of school, always a good friend. All the people from our lab, INRG, and our neighbors at CCRG.

Thanks to all my family, blood related or not, to all my friends, here and afar, and to all the random people that have positively influenced this work.

Chapter 1

Introduction

One of the implications of ubiquitous connectivity is that networks have become more heterogeneous as users have been employing a diverse set of devices ranging from laptops, hand-helds, cellular phones, pagers, and smart badges to stay connected anywhere, anytime. Furthermore, forthcoming applications such as smart environments (homes, offices, buildings, highways, etc.), factory automation, surveillance, environmental and biomedical monitoring will add a whole new set of devices that will need to communicate with one another. Therefore, network heterogeneity will manifest itself in terms of increased diversity in communication medium technology (e.g., as wired, wireless, satellite, and optical links), as well as in the types of devices networks will interconnect. In the near future, internetworks will interconnect not only traditional desktop and laptop computers, but also unconventional devices whose power, processing, and communication capabilities differ widely. These devices will form *clouds*, which will be connected among themselves and with the existing IP infrastructure.

While many of the Internet protocols have proven successful in accommodating the network's exponential growth, they were not designed to handle the degree of device heterogeneity that will characterize future internets. Consider IP: it adds an unnecessary and sometimes prohibitive amount of complexity and overhead, especially in the case of limited-capability devices like the ones found in sensor networks. More recently, protocols specifically tailored for sensor networks have been implemented. Because they are so specialized, these protocols will not be able to accommodate more sophisticated and powerful devices.

1.1 Sensor Networks

Sensor networks are one of the current network research driving forces. They have posed these and many other challenges to overcome. They have become a viable technology due to modern hardware. It is important that network protocols play their role and evolve with it.

Sensor networks are typically data driven, i.e., the whole network cooperates in communicating data from sensors (*information sources*) to *information sinks*. One of the main challenges raised by sensor networks is the fact that they are usually power constrained since sensing nodes typically exhibit limited capabilities in terms of processing, communication, and especially, power. Sensor networks' power limitation is aggravated by the fact that, often, once deployed, they are left unattended for most of their lifetime. Thus, energy conservation is of prime consideration in sensor network protocols in order to maximize the network's operational lifetime.

1.2 Data Aggregation

In-network aggregation is a well known technique to achieve energy efficiency when propagating data from information sources (e.g., sensors) to sink(s). The main idea behind in-network aggregation is that, rather than sending individual data items from sensors to sinks, multiple data items are aggregated as they are forwarded by the sensor network. Data aggregation is application dependent, i.e., depending on the target application, the appropriate data aggregation operator, or *aggregator*, will be employed. For example, suppose that in a controlled temperature environment, the average temperature needs to be monitored. As sensors generate temperature readings periodically, internal nodes in the data collection tree (rooted at the information sink and spanning relevant data sources average data received from downstream nodes and forward the result toward the information sink. The net effect is that, by transmitting less data units, considerable energy savings can be achieved. However, how much energy is saved depends on the type of aggregator employed. For instance, in the running average scenario just depicted, a number of packets containing temperature readings from individual sensors are averaged and result in a single packet of the same size as

the ones that carry individual temperature readings. However, if the only possible aggregator is *concatenation*, i.e., multiple data items are concatenated and transmitted as a single packet, then the sole source of energy savings is more efficient medium access.

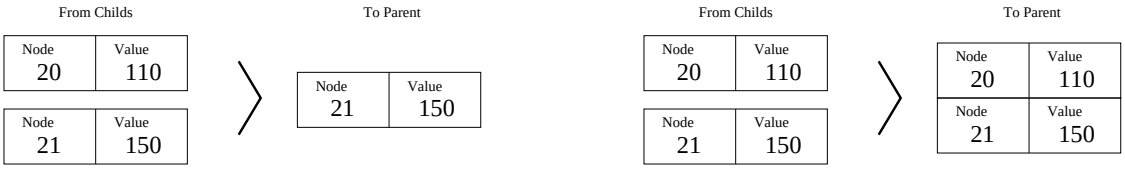
From the information sink's point of view, the benefits of in-network aggregation are that in general (1) it yields more manageable data streams avoiding overwhelming sources with massive amounts of information, and (2) performs some filtering and pre-processing on the data, making the task of further processing the data less time- and resource consuming.

Because of its well-known power efficiency properties, in-network aggregation has been the focus of several recent research efforts on sensor networks. As a result, a number of data aggregation algorithms targeting different sensor network scenarios have been proposed. Directed diffusion [5], TAG [39], eScan [24], and Sensor Protocols for Information via Negotiation (SPIN) [45] are some notable examples.

Monitoring (including monitoring of continuous environmental conditions like temperature, humidity, seismic activity, etc.) is a good example of such applications. One of the constraints imposed by periodic data generation on aggregation algorithms is timing. In other words, how long should a node wait to receive data from its children (downstream nodes in respect to the information sink) before forwarding data it already has received. Note that the tradeoff is between data accuracy and freshness, i.e., the longer a node waits, the more readings it is likely to receive and therefore, the more accurate the information it sends out. On the other hand, waiting too long may result in stale data. Furthermore, if a node waits too long, it may interfere with the next "data wave".

The more complex forms of monitoring require data collection from all nodes including location, this is the essence of environmental monitoring, map generation. In these situations simple aggregation techniques don't work as well. When calculating a maximum the information did not grow in size as it flowed down to the sink. Figure 1.1 illustrates the argument.

In order to achieve good energy savings better methods for aggregating data must be found. eScan [24] proposes to save energy by the use of polygons which get



Aggregation for Maximum Aggregation for Mapping

Figure 1.1: Aggregation methods

merged as the data flows down the aggregation tree. Madden proposes a similar method in his advanced aggregation techniques [23] that uses a grid to define the polygons.

1.3 Contributions

The main contributions of this dissertation are:

- FLIP: A base protocol architecture for use in heterogeneous network scenarios. The requirements posed on the network by different systems can vary greatly. This is normally a source of overhead for the small devices or a source of constraints for the large ones. Some architectures such a sensor networks require tight coupling with between applications and the network, others prefer the more abstracted view provided by systems designed cleanly around the ISO-OSI [49] layering scheme. Since sensor networks layers are tightly coupled it is important that we provide as much flexibility as possible at the network layer. Applications will need to have control of many of the features previously classified at the middle layers of the ISO-OSI [49] layering scheme. The FLIP architecture establishes a flexible packet structure which will allow us to provide the applications with this functionality. The price for this flexibility is a very minimal overhead.

  FLIP, or Flexible Interconnection Protocol, has a flexible header. A meta header defines the fields included in the header itself, allowing an application to limit the header to the data it really needs. Having a standardized header structure will permit various protocols to coexist under the same network without requiring a complex set of protocol schemes.

- Cascading Timers: We have established an efficient way to collect data from all

the nodes in the network. When gathering data from a single source there are certain methods for optimizing the communication in terms of various factors. These have been addressed by the numerous works on routing and traditional communications. Multicast has addressed the distribution of data to multiple sources and had to deal with a new set of challenges. Monitoring sensor networks, where all the nodes produce information present yet again a new set of problems to deal with.

We have studied the effects of timing in data collection and how to make the whole network collect data more efficiently by careful tuning of the timeout parameters. Cascading Timers are our approach to timeouts in data gathering scenarios. These timers are essential to data collection specially when doing aggregation. And due to the fact that sensor networks never have perfect communication, it is important for us to consider packet drops. When all nodes are transmitting it is essential that we consider errors, specially when aggregating data and a single drop can be very costly.

- Isoline Aggregation: Monitoring sensor networks deal with the strain of gathering data from all nodes. Aggregation and careful timing help in this respect. They are however not the only things we can do to ease our burden. When all nodes are producing data that can't be easily aggregated (i.e. location information); we have to find ways to optimize collection. We carefully studied the generation of maps and come up with an aggregation algorithm specially tailored for such a scenario; Isoline Aggregation.

  Isolines are basically the contours of a contour map. Nodes on the field will detect only the relevant data, the isolines, and report when required. Redundant information will be suppressed and not transmitted. Our approach can deal with random node placement and uneven node density distribution without incurring extra overhead.

1.4 Publications

- Solis, Obraczka. "Isolines: Efficient Spatio-Temporal Data Aggregation in Sensor Networks". August 2005, In preparation.
- Solis, Obraczka. "In-Network Aggregation Trade-offs for Data Collection in Wireless Sensor Networks". July 05, Under Submission.
- Solis, Obraczka. "Efficient Continuous Mapping in Sensor Networks Using Isolines". July 2005, Mobiquitous 05.
- Solis, Obraczka. "Isolines: Energy Efficient Mapping in Sensor Networks" June 2005, ISCC 05.
- Solis, Obraczka. "The Impact of Timing in Data Aggregation for Sensor Networks". July 2004, ICC 04.
- Solis, Obraczka. "FLIP: A Flexible Interconnection Protocol for Heterogeneous Internetworking". August 2004, MONET.
- Solis, Obraczka. "A case for a Flexible-Header Protocol in Power Constrained Networks". March 2003, WCNC 03.
- Solis, Obraczka, Marcos. "FLIP a Flexible Protocol for Efficient Communication Between Heterogeneous Devices". July 2001, ISCC 01.

1.5 Thesis Outline

This thesis is organized as follows. In Chapter 2 we will discuss FLIP. The structure of a FLIP header and meta-header. It will also describe GTP, the transport layer equivalent of FLIP. It will demonstrate via various simulations how a flexible protocol header can prove advantageous. In Chapter 3 we will detail Cascading Timers. They will be compared to other similar approaches and evaluated under multiple conditions, including a range of drop probabilities. Chapter 4 will describe our work in mapping for sensor networks. It will present static mapping as well as dynamic mapping. It also explains how Isoline Aggregation deals with high node density deployments

and reduces the possible overhead. Finally Chapter 5 will summarize our contributions and conclude our work.

Chapter 2

The FLIP Protocol - Flexibility and Efficiency

In this chapter, we describe the design and implementation of a network protocol whose main goal is to accommodate devices with varying power, processing, and communication capabilities. The proposed protocol, Flexible Interconnection Protocol, or *FLIP*, will operate among devices in the farthest branches/leaves of an intranet while providing inter-network connectivity with other clouds and with the existing IP-based Internet infrastructure. FLIP's overhead (both in terms of per-packet overhead and protocol complexity) is dependent on the capabilities of the particular device running FLIP and the functionality needed by the application. For "anemic" devices, FLIP's close to optimal overhead not only saves bandwidth, but, more importantly, energy.

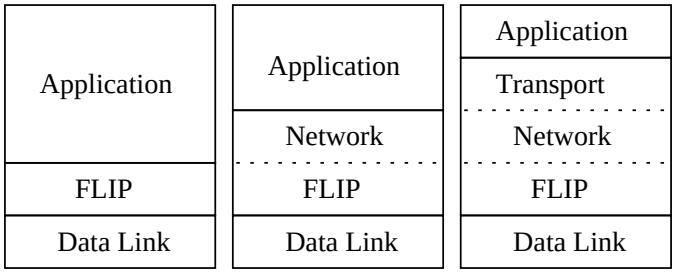


Figure 2.1: FLIP in the protocol stack.

Figure 2.1 shows FLIP's position in the protocol stack. FLIP is designed to run atop a data link layer protocol and provide functionality all the way up to the

application layer, replacing the functionality of network and transport protocols. The FLIP layer can be very "thin", which means that FLIP provides minimum functionality; this is the case of the version of FLIP that very simple devices like sensors would run. On the other hand, FLIP could provide functionality (or a subset thereof) of a "heavy-duty" transport protocol like TCP. It is the application designer's choice what services are required and should be included in FLIP.

Figure 2.2 exemplifies how functionality provided by the FLIP stack can be selected by applications. Some functions such as fragmentation are either selected or not. Others like scope as defined by a Time-to-Live (TTL) field require that a value be specified. In the case of addressing, FLIP provides options for the types of addresses that can be used (e.g., 2- or 4-byte addresses) [1]. Note that a socket-style interface is assumed. Indeed, as discussed in Section 2.2, we implemented a BSD socket interface that provides the application layer with access to FLIP.

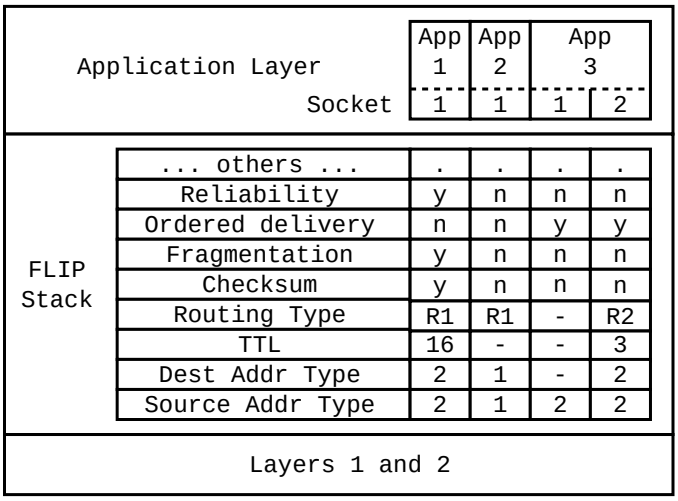


Figure 2.2: The FLIP stack

One of our focus is on how FLIP addresses the challenges posed by networks where most devices are power-anemic. Sensor networks are typical examples: they consist of an arbitrarily large number of sensing devices which rely on relatively short lifetime batteries. Sensor network applications usually imply that sensors will be left on the field unattended for extended periods of time and must conserve energy in order to

[1] Section 2.1.1 provides a more detailed description of FLIP's fields

maximize the overall network's operational time. Furthermore, while it is often assumed that sensor networks exhibit homogeneity (i.e., all sensors are either the same or have similar capabilities/characteristics), this is not necessarily the case as such networks can consist of different types of sensors. Take for example environmental monitoring, one of typical application of sensor networks. It often employs a variety of sensors ranging from "scalar" sensors (e.g., temperature, humidity, etc.), "boolean" sensors (motion, magnetometers, etc.) to "streaming" sensors (e.g., cameras and microphones).

Protocols like IP (including IPv4 [18] and IPv6 [33]), which were originally designed for "wired", fairly homogeneous networks, impose an unnecessary and sometimes prohibitive amount of complexity and overhead, especially in the case of limited-capability devices. Consider a sensing application that sends 1-byte packets. In an IPv4 network, data packets would be 95.2% header (using a 20-byte IPv4 header) and 97.6% in the case of IPv6 (using 40-byte header), which is pure overhead. For wireless, power constrained networks this is certainly wasteful and often too expensive. In such environments, several IP features are usually dispensable. For example, fragmentation will rarely, if ever, be needed in sensor network applications. Therefore, transmitting and carrying IP header fragmentation information is wasteful and if avoided, will result in valuable resource savings.

On the other hand, designing and optimizing a protocol for a single application/network scenario is prone to many problems. Several protocols will likely have to coexist in the same network and device interoperability will be challenging. Besides, in a production network, the cost of redeployment to enable new features might be high, if not prohibitive (e.g., unmanned space mission or a sea-bottom monitoring sensor network).

In heterogeneous environments, FLIP allows devices with varying capabilities to coexist and interoperate under the same network infrastructure. Due to its extensible headers, FLIP facilitates protocol evolution and deployment of new features. We demonstrate FLIP's power conservation capability in a number of scenarios.

In Section 2.3, FLIP is used to provide IPv4 and IPv6 functionality. Section 2.4 evaluates how well FLIP matches the needs of *directed diffusion* [5], while still being power-efficient. *Directed diffusion* (described in more detail in Section 2.4.1) is a

communication paradigm designed for data gathering applications in sensor networks. Using an optimized FLIP architecture we were able to save more than half the energy consumed by the unoptimized use of *diffusion*. We then consider in Section 2.5 a sample sensor network application, namely running average calculation of sensed data. We use temperature as the data being reported by sensors and develop a simple application-level data gathering protocol. We implement this data gathering application using two different protocol header paradigms: (1) FLIP's adaptive header and (2) static headers represented by two models, namely complete and minimal headers. Our simulation results show that FLIP outperforms static headers by as much as 12% while providing full functionality. Finally, in Section 2.6, we add data aggregation to the data gathering protocol and show its energy-savings effects. Adding such new features is part of protocol evolution and can be easily accomplished in the case of a flexible-header protocol like FLIP. Employing data aggregation resulted in energy savings of as much as 30%. Sections 2.1 and 2.2 describe FLIP's basic design principles and a reference implementation under Linux, respectively [2]. Related work is discussed in Section 2.7 and Section 2.8 presents our concluding remarks.

2.1 FLIP Design Principles

The overhead and complexity of a protocol is directly related to the functionality the protocol provides. Recall that FLIP's main goal is to accommodate a range of devices with varying capabilities and yet provide the functionality required by applications. Thus FLIP allows application programmers to select just the functionality they need, without incurring the overhead associated with functions they do not need. Furthermore, the ability to select a subset of protocol functions allows FLIP to accommodate a range of devices from very simple sensors to desktop computers. For instance, if the application needs packets to age, then the application programmer can "turn on" FLIP's Time-To-Live (TTL) field. At each hop, the packet's TTL value will be decremented and examined, if it reaches 0 the packet is discarded; otherwise, the packet is forwarded. Users can also "turn on" the length field, whose value will be calculated as part of composing a packet.

[2] A preliminary design of FLIP was presented in [15]

In its simplest form, FLIP does not include end-to-end reliability or ordering. It might not even perform routing. This is because, in some scenarios, routing is done by the application using special information such as nodes' geographic positioning or remaining power. Some of these scenarios may use small, very simple devices which would only be encumbered with routing. These are just end devices and do not have the required capability to perform routing functionality effectively. For example, in the case of simple sensors, they may just perform one-hop broadcasts to send out their readings each time. A nearby, more capable node can then collect these readings and route them towards the destination.

2.1.1 The Network Layer

FLIP packets are composed of a *meta-header*, header fields and the payload. The *meta-header* indicates which header fields are present in the packet and consists of an array of bits, or bitmap. If a header field is included in the packet, then the corresponding bit in the meta-header is one, otherwise, it is set to zero.

In order to minimize the bitmap's overhead, we split it into one-byte pieces. Each byte contains a continuation bit that indicates if more bitmap pieces follow. Figure 2.3 shows an example of the FLIP meta-header. Note that the continuation bit is the first bit of each byte. This ensures that, in the 2-byte version of FLIP's extra simple packet (ESP) (FLIP's ESP mode will be described below), the payload occupies contiguous bits.

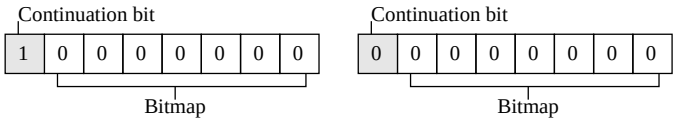


Figure 2.3: FLIP meta-headers

Consider a scenario that only requires the *length* field, whose presence bit lies in the first byte of the meta-header. Then, the packet will only have to carry the first byte of the meta-header, which will have the bit corresponding to the length field on, and the others, including the continuation bit, off.

In some cases, the target application need only send small amounts of data

with no header information. Sensor network environments are a good example of such a scenario: sensors simply broadcast data related to what they are sensing. In these cases, even a 1-byte meta-header to indicate that no header is needed is too expensive. For instance, if sensors broadcast 1-byte data, then 1-byte headers result in 50% overhead. To address these scenarios, FLIP offers the *extra simple packet*, or ESP. We designated the second bit of the meta-header, that is, the one following the continuation bit in the first byte, to be the ESP bit. If this bit is set, it indicates the packet at hand is an ESP. The use of the continuation bit in the ESP allows for 1- and 2-byte ESPs. While a 1-byte ESP, that is, one with the continuation bit off, contains 6 data bits, a 2-byte ESP allows for 14 bits of data (all the 8 bits of the second byte will be counted as data). Figurer 2.4 depicts both ESP cases.

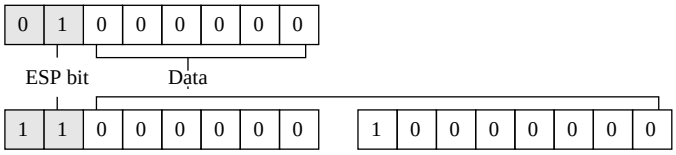


Figure 2.4: Extra Simple Packet (ESP)

FLIP's ESP addresses the need for a real "barebone" protocol, which will be used by applications that need to send small pieces of data with no overhead. FLIP's regular meta-header bitmap covers the more general cases where some fields are required and some are not, thus optimizing average use.

As shown in Figure 2.5, FLIP's current meta-header bitmap spans 3 bytes, including three continuation bits and the ESP bit. The last meta-header byte has been left unspecified as it will be used for adding new features as part of FLIP's evolution.

We should point out that the ordering of the fields was determined so as to optimize packet overhead for very simple applications and devices. More complex devices and applications can normally amortize the cost of having more meta-header bytes. Fields are ordered so that the most commonly needed ones appear in the first meta header byte(s). More infrequently used fields appear last so that the corresponding meta-header byte does not have to be included when these fields are not used. The definition of each FLIP header field follows.

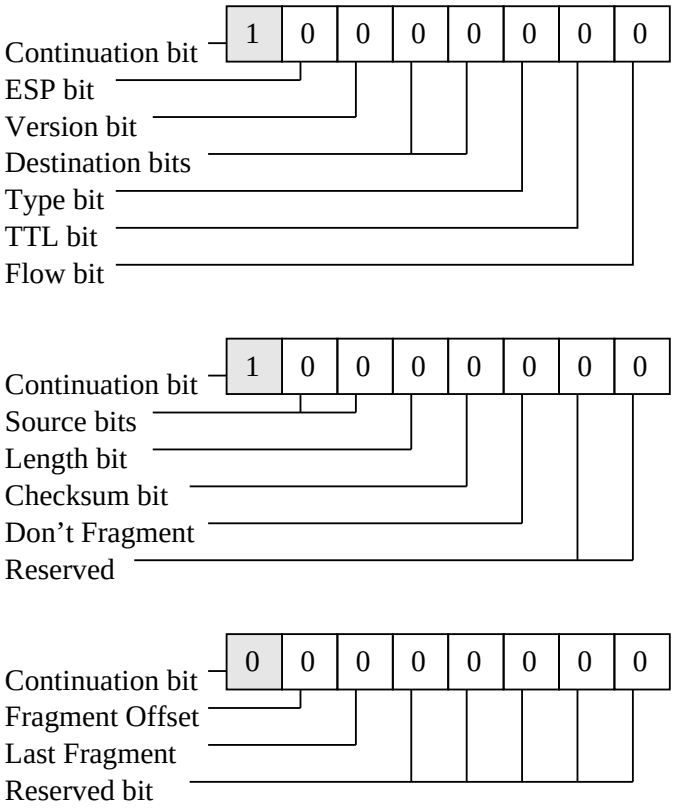


Figure 2.5: The FLIP meta-header bitmap

- **Version** is 1 byte in length. The 4 higher order bits represent the version field. The current FLIP version is 0. The 4 lower order bits represent the priority field. If a packet lacks the version field, version 0 and priority 0 are assumed.

- **Destination** is a variable-length field. The corresponding meta-header field is composed of 2 bits, whose value determines the size of the destination field. If the bitmap bits are set to:

  - 00 indicates the destination field is not present.
  - 01 indicates we have a destination field of 2 bytes in length carrying a FLIP address.
  - 10 indicates it is a 4-byte destination address.
  - 11 indicates the destination field is of variable length.

In the case of a variable length address, the first byte indicates the size of the field, which could range from 5 to 255. Values of 0 to 4 are reserved for future use, such as geo-location. IPv4 [18] addresses correspond to 4-byte FLIP addresses and IPv6 [33] addresses to variable length addresses of size 16.

- **Type (Protocol)** is 1 byte in length and indicates the protocol type. This matches the IPv4 field by the same name and IPv6's next header field.
- **Time to Live (TTL)** is 1 byte in length and is typically used to limit the scope of a packet. It may define the scope in number of hops, i.e., at every hop the TTL is decremented and when it reaches 0, the packet is discarded. Applications may also define the scope of their packets in terms of other metrics, such as geographic area, etc.
- **Flow** is 4 bytes in length. As the name implies, this field is intended for flow identification. Flow identifications are useful to implement features such as support for flow-based quality of service (QoS). Packets belonging to a given flow will be subject to QoS parameters negotiated for that flow.
- **Source** is a variable-length field and its length is determined by 2 bits in the meta-header exactly the same way as the destination field.
- **Length** is 2 bytes, which means that the maximum packet size is limited to 64 KBytes.
- **Checksum** is 2 bytes in length and checks the packet payload. It is calculated similarly to the IP Checksum.
- **Don't Fragment** is a flag which means it does not require the corresponding header field. It explicitly informs the forwarding nodes not to fragment this packet.
- **Fragment Offset** is a 2-byte field indicating this fragment's offset with respect to the original packet.
- **Last Fragment** is a flag used to indicate this is the last fragment of a packet.
- **Reserved** are still to be defined fields.

A sample of a FLIP packet is depicted in Figure 2.6. The shaded area represents the header, and the remainder, the payload. In this example, the meta-header is 1-byte long (continuation bit set to 0) and signals the presence of the version field, a 2-byte destination address, and the type field. All other fields, including the source address, are not included in the packet

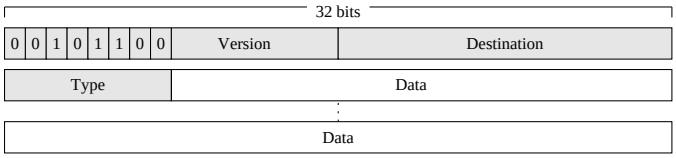


Figure 2.6: FLIP sample packet

For increased flexibility, FLIP also allows for *user-defined* header fields. If the continuation bit of the third meta-header byte is set, it indicates that user-defined header fields are included in the packet. Each user-defined header field definition is one byte in length: the first bit is the continuation bit, and the remaining 7 bits are defined and interpreted by the application.

An example of a user-defined field is the velocity of the source, in case the source is currently moving. The destination may use this information to compute its velocity relative to the source to evaluate the "stability" of its connection to the source. User-defined information may also include a list of hosts this packet has traveled through. A node might use this information for packet processing or routing. Another example is current energy level. In power-constrained environments (e.g., sensor networks), this information might be useful to determine the current power limitations of a certain route.

FLIP header fields are usually fixed size. For example, *TTL* is always 1 byte and *length* is always 2 bytes long. Addresses are a special case. Assigning 2 bits to the meta-header address field allows FLIP addresses to be of variable length. 2-byte addresses are adequate for scenarios where addresses need not be globally unique (i.e., system-unique or locally-unique addresses suffice). Addresses that are 4-byte long give us effective compatibility with IPv4 networks. Variable length addresses can be used to emulate other addressing schemes, including IPv6, Ethernet, or even hierarchical

address types.

In some cases, source and/or destination addresses may be omitted. If a packet does not include destination address, it is assumed to be a broadcast packet. If a source address is not present, it is assumed to be irrelevant or somehow implied by the packet.

When present, fragmentation information is handled in a similar way to IP. A *don't fragment* flag indicates that this packet should not be fragmented. The *fragment offset* is a two-byte field, and the *last fragment* flag is again a single flag. These two fields are included in the third byte of the meta-header, leaving two unused bits in the second byte. This is because not many packets are fragmented, and when they are, it normally means they are large, so we can amortize the cost of the extra meta-header byte. We anticipate that fields that might be needed in the future may be of more frequent use and hence it would be more efficient to use the space in the second meta-header byte. A clear example is security-related fields, which we have currently not included.

FLIP allows application developers to customize its header by selecting fields required by the target application. Allowing direct manipulation of header fields by the application layer can be considered a violation of layered system design. However, exposing network-layer features to higher layers allows for protocol optimization, which is especially critical in power-constrained environments such as the ones in which FLIP will likely be more widely used. Our reference implementation of FLIP in the Linux 2.4 kernel, which is described in the next section, provides access to header manipulation functions through the `socket()` and `set/getsockopt()` interfaces.

2.1.2 The Transport Layer

As previously discussed, FLIP provides the basic substrate on which to build network- as well as transport-layer functionality. In this section, we present an example transport protocol, we call GTP or Generic Transport Protocol, built atop FLIP. GTP's design is based on the same principles as FLIP, i.e., generality with flexibility. In other words, GTP provides support for a variety of transport-level functionality yet allows the application developer to select only the functions required by the target application. To this end, it also employs customizable headers through a meta-header describing the transport-layer header fields.

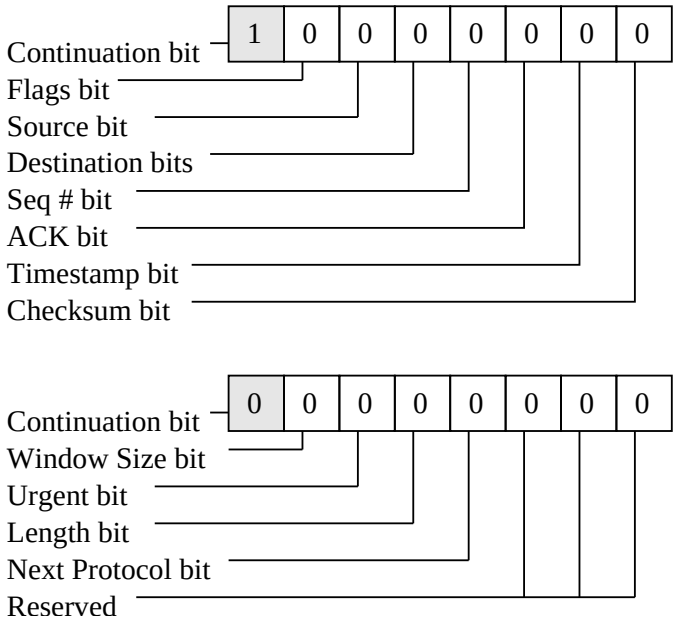


Figure 2.7: GTP meta-header

Figure 2.7 depicts GTP's meta-header and its fields. GTP's header includes fields used by existing transport protocols, specifically TCP [19], UDP [17], and RTP [13]. Similarly to FLIP, the most general fields were placed in the first byte to optimize for more common use. It is noteworthy that FLIP packets can contain various transport protocol data units (TPDUs), which can come in handy when performing data aggregation.

GTP's header fields are described below.

- **Flags** is a variable-size bitmap field which, similarly to the meta-header, can grow dynamically through the use of continuation bits. Currently, only the first byte has been defined. Figure 2.8 shows the composition of GTP's flag field and the description of each flag follows.
  - **Extended mode** indicates that this packet uses extended addressing, in which case the source and destination fields are four bytes long as opposed to two bytes.
  - **SYN** is the normal synchronization flag used for three-way handshake at

connection establishment.

  - **FIN** is the flag used to end a connection.
  - **Reset** is used to reset a connection to an initial state.
  - **Push** informs the receiver to pass the received data to the application without waiting for the internal buffer to fill.
  - **Marker** is a application-level mark on a stream. It can be used for example, to mark frames in a video stream.
  - **Padding** informs the receiver that this data unit was padded to the next 4-byte boundary.

- **Source** is a 2-byte field used for addressing multiple sources within the same host. It is the equivalent to a TCP/IP source port. On extended mode, this is a 4-byte field.
- **Destination** is similar to Source.
- **Sequence #** is a 4-byte field that determines the position of this packet in the data stream. The position is by default in bytes.
- **ACK** is a 4-byte field used to acknowledge the reception of data from the other end of the connection. Its value is in bytes.
- **Timestamp** is a 4-byte field with a relative value. Timestamps are relative to each other and can be scaled by the application.
- **Checksum** is a 2-byte field used to check the integrity of the data unit.
- **Window Size** allows the receiver to inform the source the amount of data it can receive. It is a 4-byte field (to avoid having to use scaling factors a la TCP).
- **Urgent** is a 2-byte field to indicate to the receiver this 2-byte field is carrying urgent data.
- **Length** defines the size of the data unit and is 2 bytes long. This is used normally for multiple data units in the same packet, since the size of a single data unit can be inferred from the packet length.

- **Next Protocol** is a single byte that specifies the type of the next data unit.
- **Reserved** are fields that have not yet been defined.

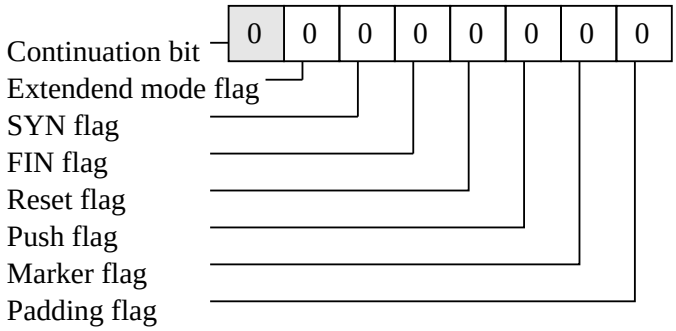


Figure 2.8: GTP Flags field

The use of a flags field as opposed to including the flags as part of the meta-header (a la FLIP) is due to the fact that (1) there are more transport-layer flags and (2) they will only be used in some packets. Thus adding them to the meta-header would have meant that the meta-header's average size would have increased. With the current design, packets that do not require *SYN* and *FIN*, for example, do not need to carry the extra bits.

We should also point out that some of TCP's flags are implicit in GTP's meta-header. Hence there is no need for an extra flag for *ACK* or *urgent data*. Note that in the case the packet is not carrying an acknowledgement nor urgent data, the meta-header bits corresponding to these fields will be 0 or not present.

Figure 2.9 shows a sample FLIP/GTP packet. The shaded area corresponds to the FLIP and GTP headers. In this example, the FLIP meta-header is 2 bytes and indicates the presence of 2-byte destination and source addresses, plus the protocol type (which should be set to GTP), the TTL and length fields. GTP's meta-header is also 2 bytes long. It signals the presence of source and destination addresses, as well as sequence number, acknowledgment and window size information.

Like FLIP, GTP's design makes use of flexibility to address heterogeneity and accommodate devices with different capability. Yet, it provides a variety of transport-level functions that can be combined to address the application's needs. Section 2.3,

which evaluates GTP for providing different transport-level functionality, demonstrates that GTP's ability to include only the functions required by the target application leads to higher efficiency when compared to static protocols like TCP and UDP.

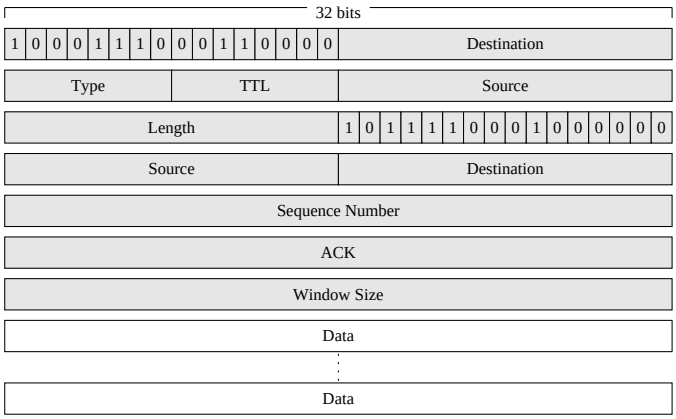


Figure 2.9: FLIP/GTP packet

2.1.3 FLIP and Heterogeneity

One of the main goals in designing FLIP was to construct a protocol that allows a diverse set of devices to speak to each other in an efficient manner using the same protocol suite. As an example, consider deploying an ad hoc network consisting of thousands of different types of sensors (temperature-, humidity-, and motion sensors, as well as microphones, cameras, etc.) for environmental monitoring in a remote location.

Simple sensors such as temperature would use FLIP's ESP packets to report their readings. Data collection nodes would gather sensed data received from local sensors. Using a more complex FLIP header, they would form data collection structures (trees, meshes) and reliably convey data they collect to information sinks. For energy efficiency, data would be aggregated as it flows from collection points to sinks.

For energy conservation reasons, sensors such as cameras and microphones would be kept in "stand-by" mode most of the time. As soon as an event is detected (e.g., wind sensors notice winds picking up above a certain threshold), cameras and microphones in that area would receive a "wake-up" signal from the local data collecting node and would start collecting information to register a possible weather phenomenon

(e.g., dust devil, tornado, etc.). The "wake-up" signal could use FLIP's priority field to indicate that this information needs to be forwarded by intermediate nodes with higher priority. Cameras and microphones are attached to more powerful nodes that can keep up with their data generation rate and perform local processing in order to avoid overloading sinks with too much information and consuming too much network resources. Some cameras and microphones in the same neighborhood could also perform information fusion among them to decide whether they should send information to sink(s), and if so, what kind of information representation to use (e.g., if the probability of event occurrence is deemed low, send only the base layer of the compressed stream from the camera that is closest to where the action is). These real-time streams could be sent using FLIP/GTP's unreliable stream functionality a la RTP.

Scientists on the field equipped with hand-held devices could also be collecting sensed information in real-time. They could then communicate among them sharing information and/or exchanging files using FLIP/GTP's reliable end-to-end delivery functionality. They might also need to communicate with their collaborators connected to the wired Internet; this would be accomplished via translation gateways, which convert FLIP/GTP packets to TCP/IP and forward them onto the wired infrastructure.

Clearly, all exchanges in this heterogeneous scenario could be carried out using different protocols to handle the different types of communication. The main advantage of using the FLIP stack is that it provides a single, yet efficient protocol architecture which can be used for simple data gathering, point-to-point data communications as well as more complex exchanges.

2.2 Implementation

As proof of concept, we implemented a barebone version of FLIP in the Linux 2.4 kernel. We generated a patch for the kernel which allows the inclusion of FLIP at compile time or as a loadable module. Linux is making its way into devices of various kinds and capabilities; having a Linux implementation of FLIP will allow us to conduct live experiments in heterogeneous environments.

Below the FLIP code lies the device code, specifically the device output/input queues, through which FLIP sends/receives data. When data is received, the receiving

device passes it to the FLIP layer which queues the data on the corresponding socket.

FLIP uses the BSD socket abstraction to interface with applications. In order to send and receive data using FLIP, application programmers will use the same set of socket system calls they would use to handle TCP/IP communication endpoints. For instance, to create a FLIP socket, all they have to do is request a socket of family `AF_FLIP`.

```
char* buf = "Hello World";
__u16 addr;

s = socket(AF_FLIP,SOCK_RAW,FLIP_NO_ESP);
addr = htons(1000);
setsockopt(s,SOL_FLIP,FLIPO_DESTINATION,
           &addr,sizeof(addr));
write(s, buf, strlen(buf));
```

Figure 2.10: Sample application code

Figure 2.10 illustrates the FLIP API. In this example, a FLIP socket of type `SOCK_RAW` (allowing the programmer to modify most of the fields) is defined. The `CAP_NET_RAW` capability is required to use the socket if capabilities are being used. FLIP sockets will eventually be able to support datagram and stream once transport layer functionality is implemented. `Socket`'s last parameter is used to select ESP or non-ESP mode. With the current implementation, the programmer cannot change from one mode to another once the socket is created.

The programmer can then use `setsockopt()` to set the necessary header fields. For instance, address fields (source and destination) identify a given communication end-point. If a socket is assigned an address, that socket will only receive packets with that address in the destination field. The address of a FLIP traffic source is set through the `FLIPO_SOURCE` option. If no address is assigned to a socket, FLIP will not set the source address on outgoing packets from that socket. In the example of Figure 2.10, the destination field is defined as a 16-bit FLIP destination and is set with the `FLIPO_DESTINATION` option.

Since ESP packets have no headers, and thus no destination or source addresses specified, ESP sockets always receive all packets.

The `getsockopt()` call is used to read header definitions for a certain socket, as well as to read the header fields of incoming packets on that socket. As previously pointed out, to achieve flexibility and efficiency, our design exposes the network layer to the application programmer.

In order to speed up packet header construction, we cache header information for every socket that has been defined. Dynamic header fields, which change from packet to packet, are computed on the fly before the packet is sent. *Packet length* and *checksum* are examples of dynamic header fields.

In the current implementation, we use Ethernet and 802.11b as the MAC layer protocols. Like RF wireless access, Ethernet assumes a shared broadcast medium. A unique protocol number was selected as Ethernet's *next protocol* field. Using these underlying MAC protocol means that FLIP packets must be at least the size of the minimum frame payload. Consequently, in this implementation, we cannot take full advantage of FLIP's ESP mode.

2.3 Evaluation

In this section we compare FLIP's functionality and overhead against a more "traditional" protocol, namely IP. We also evaluate FLIP in the context of a sensor network environment.

2.3.1 FLIP and IP

We should point out that both FLIP and IP were designed to address different goals and target environments. Thus comparing them is not really fair to either. While the IP layer provides a fixed set of functions, FLIP's functionality and overhead are application-dependent. In other words, the application determines which fields are to be included in the FLIP packet header. Therefore, applications send just what they need, avoiding the cost of transmitting and processing unnecessary information.

Take for example an application that sends out data in 1000-byte chunks. Using IPv4, the overhead would be 20 bytes (corresponding to the IPv4 header), which is

Table 2.1: FLIP-IP comparison in terms of packet size

Functionality	Protocol		
	IPv4	IPv6	FLIP
Full IPv4 (header only)	20	N/A	24
Full IPv4 (1b payload)	21 (2000%)	N/A	24 (2300%)
Full IPv4 (1000b payload)	1020 (2%)	N/A	1024 (2.4%)
Typical IPv4 (header only)	20	N/A	17
Typical IPv4 (1b payload)	21 (2000%)	N/A	18 (1700%)
Typical IPv4 (1000b payload)	1020 (2%)	N/A	1017 (1.7%)
Full IPv6 (header only)	N/A	40	44
Full IPv6 (1b payload)	N/A	41 (4000%)	45 (4400%)
Full IPv6 (1000b payload)	N/A	1040 (4%)	1044 (4.4%)
Dest. & Source (header only)	20	40	10
Dest. & Source (1b payload)	21 (2000%)	41 (4000%)	11 (1000%)
Dest. & Source (1000b payload)	1020 (2%)	1040 (4%)	1010 (1%)
Dest. only (header only)	20	40	3
Dest. only (1b payload)	21 (2000%)	41 (4000%)	4 (300%)
Dest. only (1000b payload)	1020 (2%)	1040 (4%)	1003 (0.3%)

not significant given the size of the payload. However, if hosts are just sending 1-byte heartbeat messages (e.g., either their address or some form of identification), then 20 bytes of header would seem unacceptable. Fields such as fragmentation information, ToS, or even packet length (in the case of fixed-size packets) would be adding unnecessary overhead and wasting network, and even more importantly, device resources (such as power). If IPv4 is used, the message would be 21 bytes long, where only 1 byte is payload. The corresponding barebone FLIP packet could be 5 bytes long: 1 meta-header byte, a 4-byte destination address, and 1-byte heartbeat, which results in a 400% increase in efficiency (when compared to the IPv4 packet).

When comparing the functionality of IP and FLIP, we need to examine the issue of header compatibility. IP header fields are easily mapped into FLIP fields. Indeed, FLIP was designed with IP-compatibility in mind. It is fully compatible with IPv6. To emulate IPv6 functionality, the version, flow, length, protocol (for next header), and TTL (for hop limit) header fields need to be enabled. Moreover, 16 byte addresses for source and destination should be selected. The overhead of using FLIP instead of IPv6 is four bytes: two bytes for the meta-header, one extra flow id byte (FLIP's flow id is 4 bytes long while IPv6's is only 3) and the size specification of the address. This

additional overhead is relatively low: it results in only 10% header size increase.

IPv4 emulation varies a little bit. If we provide "full IPv4 functionality", including fragmentation, we would need to choose the same fields as IPv6 plus checksum and fragment offset. We would use the 4-byte flow field as IPv4's id. This would waste 2 bytes. IPv4 options can be included as FLIP user defined fields. The overhead of using FLIP to emulate a header like this would be 4 bytes: three meta-header bytes, two extra bytes in the flow field, a smaller priority (ToS) field and no header length field. When providing "typical IPv4 functionality", there is no need for fragmentation or flow id; the version field can also be omitted. This results in a header of 17 bytes for the typical case of IPv4 functionality.

In homogeneous environments (e.g., where all devices are capable of speaking IP), FLIP's flexibility is dispensable, and thus even a small increase in overhead may be unwarranted. However the main point in comparing FLIP's and IP's functionality is to show that FLIP can be used by very simple devices with minimum overhead, and, at the same time, provide IP-style functionality when needed with minimum cost.

FLIP's main drawback, when compared to IP (or any "fixed-header" protocol), is associated with the fact that header parsing becomes a more involved task. Clearly, higher header processing overhead implies that it takes longer to forward packets. Regarding protocol implementation, communication between layers becomes more complex since now varying-size data has to be passed between layers. Furthermore, allowing users to modify protocol header fields raises implementation correctness issues.

Table 2.1 shows a comparison between FLIP and IP (IPv4 and IPv6) in terms of packet size. Each column is associated to one of the protocols, namely IPv4, IPv6, and FLIP. The rows list the required functionality. "Destination and Source" uses 4-byte addresses for the the destination and the source only, while "Destination" includes only a 2-byte destination address. The cells show the packet size. The number in parenthesis is the size of the header compared to the payload (given as a percentage). We consider 3 payload sizes: 0- (or header only), 1-, and 1000 bytes. For instance, in the case of the 1-byte payload, IPv4 uses a 20-byte header. Thus the header to payload ratio is $(200/1) = 2,000\%$.

As previously pointed out, the purpose of this table is to showcase FLIP's

Table 2.2: GTP - TCP/UDP comparison in terms of packet size (in number of bytes)

	Data size	TCP/UDP	GTP
Setup Packet	0	40 (20 + 20)	34 (17 + 17)
Reliable Packet	50	90 (20 + 20 + 50)	82 (17 + 15 + 50)
Unreliable Packet	50	78 (20 + 8 + 50)	74 (17 + 7 + 50)

flexibility-overhead tradeoff when compared to fixed-header protocols. FLIP can provide both functionality of "traditional", more complex internetworking protocols, such as IPv4 and IPv6, at reasonably low cost, as well as functionality of a barebone protocol incurring minimal overhead.

Table 2.2 compares GTP (running atop FLIP) to TCP/UDP (atop IPv4). The rows represent different types of exchange: connection setup (SYN), reliable and unreliable data packets. The first column shows the data size, that is, the payload. The following columns show the packet size for TCP/UDP and for GTP. The number in parenthesis is the breakdown of header and payload sizes.

The FLIP header size of 17 bytes supports typical IPv4 functionality requirements as previously derived. The setup packet includes flags, source and destination address, sequence number, checksum and window size, resulting in a GTP header size of 17 bytes. For reliable exchanges, the GTP header includes the ACK field in addition to source, destination, sequence number, and checksum. Note that, when compared to the connection setup case, flags and window size (assuming it does not change during the connection) are not included, resulting in a header size of 15 bytes. The header for unreliable exchange includes source, destination and checksum. GTP meta-header requirements are 2 bytes for the setup packet (need to include flags) and 1 byte for the last 2 cases.

We should reiterate that the goal of FLIP/GTP is not to replace the TCP/IP network architecture but to extend its scope to interconnect heterogeneous devices among them and to the existing IP infrastructure. The comparison in Table 2.2 shows the benefits of using a flexible, customizable protocol suite in heterogeneous network environments. Essentially, FLIP/GTP is able to provide the same functionality as TCP(UDP)/IP at lower cost. This is due to FLIP/GTP's ability to include only the functionality needed by target applications.

2.3.2 FLIP-IP Integration

FLIP's goal is **not** to replace but rather **extend** the scope of IP to interconnect clouds of varying capability devices to the existing IP infrastructure.

FLIP and IP can co-exist and inter-operate using different integration strategies. One way of integrating the two protocols is through simple encapsulation. For example, in order to interconnect FLIP-capable islands across an IP infrastructure, FLIP tunnels can be used. Upon leaving a FLIP cloud, FLIP packets are encapsulated into IP datagrams by a FLIP-IP gateway. When reaching the FLIP-capable network destination, IP-FLIP gateways restore the original FLIP packets, stripping off the IP envelope. An alternate mechanism is to tunnel IP traffic through FLIP networks. IP datagrams could be encapsulated in a header indicating a "IP-in-FLIP" type and an address.

In fact, we foresee that, even though FLIP-IP encapsulation will likely be more common, both tunneling mechanisms will be needed in heterogeneous networks and will be used complementary to one another.

2.4 Sensor Networks

Sensor networks are one of FLIP's key target application domains. In most sensor network scenarios, the goal is energy conservation as sensing devices rely on batteries with relatively short lifetime. Typically, sensor network applications imply that sensors will be left on the field unattended for extended periods of time and must conserve energy in order to maximize the whole network's operational lifetime.

Sensor devices, and implicitly sensor networks, are data driven in the sense that the whole network cooperates on the task of communicating data from sensors to end users. In these kinds of scenarios, FLIP optimizes communication among nodes by only transmitting required information with minimum protocol overhead. For instance, FLIP's ESP provides application programmers with a considerably lightweight packet. ESPs can be used in scenarios such as coordination between peers in radio range or transmission of small data chunks (e.g., readings from temperature, humidity-, and motion sensors). The inclusion or exclusion of destination and source fields could

determine the scope of the data as routable or one-hop (such as "running out of battery" or "hello" messages). FLIP's different address types allow proposals such as the address-free architecture [20] to coexist with more traditional addressing schemes.

In order to evaluate how FLIP addresses the needs of sensor network applications, we selected as a case study the directed diffusion architecture [5]. Directed diffusion is a communication paradigm for sensor networks which establishes *interests* for specific data (e.g., number of cars that flow through busy intersections during rush hour). Relevant data flows towards nodes that expressed interest in named information. Routing is done by the application, which aggregates data when possible. Since these applications are very involved with the network, a transport layer is not used.

Directed diffusion's original header is 22 bytes long. If IPv4 was used to implement directed diffusion, it would incur an overhead of 9 bytes, and would have to carry *packet number* information in the payload. In the case of IPv6, the overhead would increase to 29 bytes. Using FLIP, the overhead would be only 2 bytes, corresponding to FLIP's meta-header.

2.4.1 Directed Diffusion and FLIP

Directed diffusion [5] is a communication paradigm especially targeted at data-centric sensor networks. In such environments, nodes collaborate to get the data from its source(s) across the sensor network to the data sink(s). A sink node sends an *interest* for a certain data. This interest will be broadcast to the whole network. As a result, a *gradient* will be set up along the path. If a node has relevant information to that interest, it will send the data along the gradient back to the sink.

Originally, directed diffusion was implemented as a very specialized application-level protocol. Diffusion implementors' main goal was to develop a working architecture for data-driven sensor networks, rather than build a generic network-layer protocol. In this section, we evaluate the tradeoff between FLIP's flexibility and efficiency as the network-layer protocol underlying diffusion.

We implemented "diffusion-over-FLIP" in two ways. In the first approach, we constructed diffusion's complete header using FLIP and evaluated the resulting protocol's overhead when compared to "plain" diffusion. In other words, we left every

diffusion field intact and did not perform any sort of optimization. Secondly, we implemented diffusion from scratch assuming FLIP as the underlying protocol. In this case, we optimized where possible.

2.4.2 Diffusion over FLIP

We use the diffusion packet definition from their 3.0 beta release [40]. Figure 2.11 shows a sample diffusion packet. We then map each diffusion field to the corresponding FLIP field. Most diffusion fields can be directly translated to FLIP header fields: `last_hop` is mapped to `source`, `next_hop` to `destination`, etc. More specialized diffusion fields are carried in the payload. `number of attributes` and `sender port` are two examples. The resulting FLIP packet is 2 bytes longer than the original diffusion packet since we have the overhead of the meta-header.

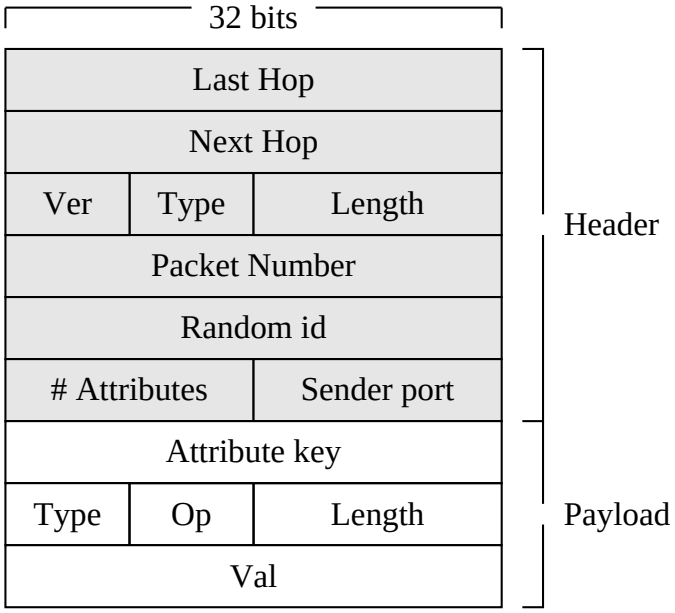


Figure 2.11: Diffusion packet for an `int` attribute

The goal of this exercise is to show that it is relatively simple for an existing application to adopt FLIP as its underlying network protocol without incurring excessive overhead.

2.4.3 Optimizing Diffusion with FLIP

The second approach to evaluating FLIP in the context of diffusion is to address the following question: how would one re-design diffusion assuming FLIP as the underlying protocol? We consider diffusion's different packet types: `interest`, `reinforcement`, and `data`. As expected from a fixed-header protocol, all packet types use the same packet header. The question then becomes: can one take advantage of FLIP's flexible headers to optimize diffusion's exchanges?

In the case of `interest` packets, the header only needs to carry `source`, `flow (packet id)`, and `type` fields. This customization reduces `interest` headers to 11 bytes, including the meta-header overhead. The payload portion of this type of packet can be reduced to 10 bytes for simple interests. That is, interests that have only one attribute and that can deduce the type of data from the `attribute key`. The total packet size for interests will be 21 bytes, in contrast to 36.

In addition to `interest` header fields, `reinforcements` also require a destination field because they reinforce a specific path. This results in 25-byte packets as we are using 4-byte addresses.

Data packets flow in the opposite direction to interests. Similarly to `reinforcements`, they carry both `source` and `destination` fields because they need to leave a trail for reinforcements. `Data` and `reinforcement` packet headers end up being the same. In their payload, we are able to save one byte used for specifying query on attributes, making it 9 bytes long for an `int` attribute type interest. The total packet length will be 24 bytes. Figure 2.12 shows the resulting optimized diffusion packets. Shaded and unshaded areas denote header and payload, respectively.

2.4.4 Simulation Results

To evaluate these FLIP-based diffusion variants, we modified the original diffusion code in the `ns-2` network simulator [12]. We ran a data gathering experiment with the diffusion algorithm, a sink sends out an interest and one node responds with information (data source). In our experiments, we use sensor networks consisting of 300 nodes scattered across a 2000 x 2000 meter area. 802.11 is the underlying MAC protocol. The energy values for radio transmission and reception are based on the original

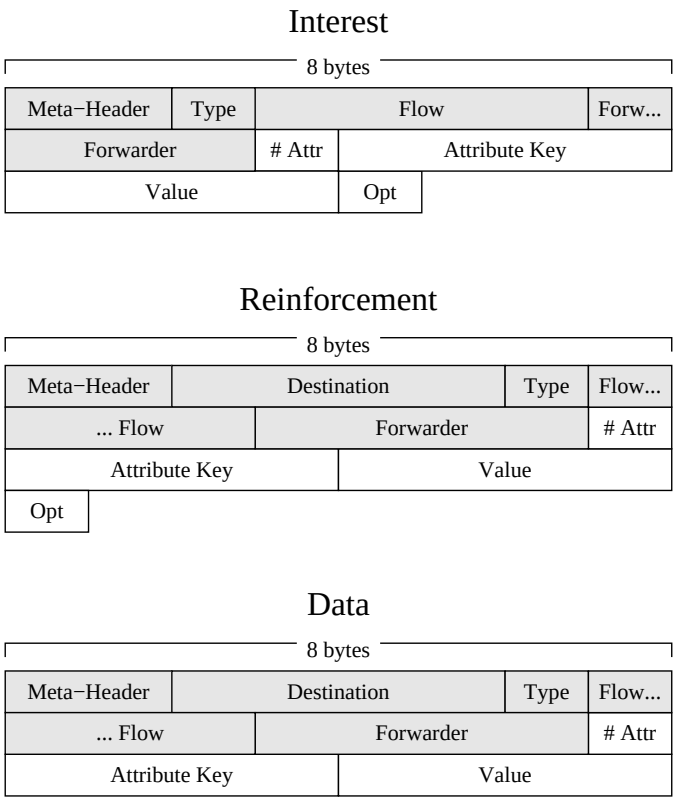


Figure 2.12: FLIP-optimized diffusion headers

diffusion evaluation values, i.e., 395mW in reception mode and 660mW when transmitting. Nodes remain static in the sensor network and have a 250m transmission range. To accentuate the difference between diffusion variants we reduce idle energy dissipation to 0, which suppresses the effects of lower layer (data link and physical) overhead. Node failures were not considered.

Figure 2.13 shows average node energy over time. Nodes' starting energy is 1.0 Joules. Data points represent averages over 10 runs with different random topologies. We use one sink, one source and a requested data rate of 10 packets/second. The graph's "step" shape is due to how the diffusion algorithm operates: it resends interests every 5 seconds. Simulations were run for 21 seconds.

As expected, *diffusion over FLIP* consumes slightly more energy than diffusion since the packets are 2 bytes longer. However, difference in energy consumption between the two protocols is relatively small.

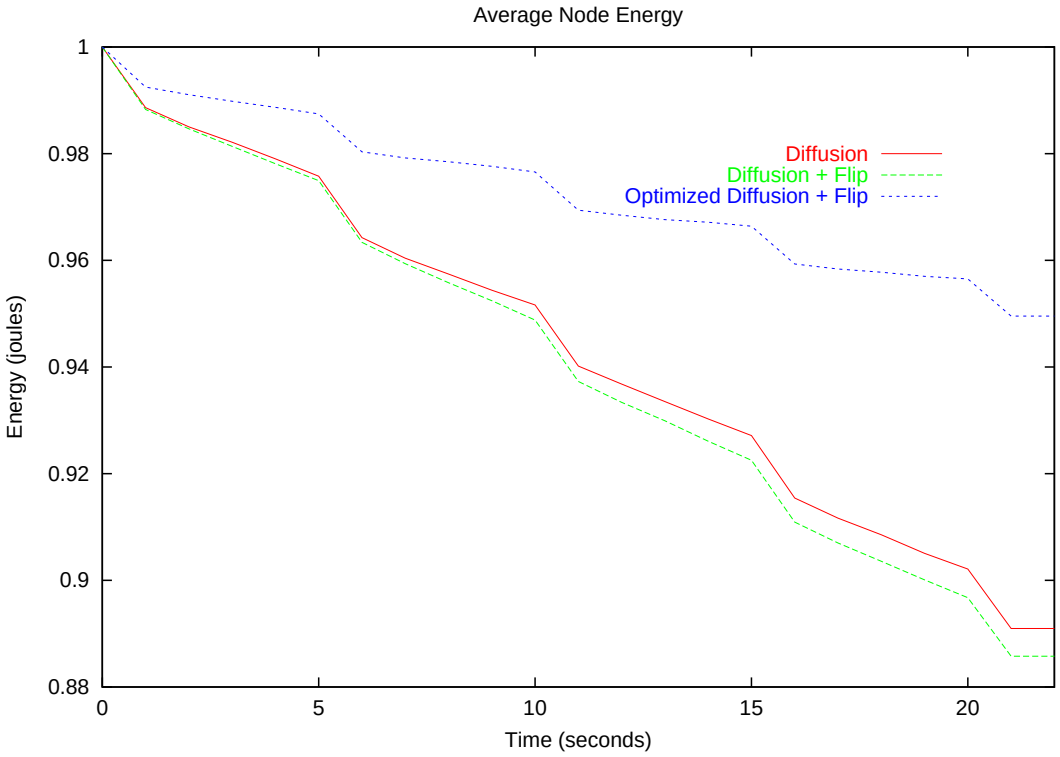


Figure 2.13: Energy levels over time for different diffusion variants.

Optimized diffusion on the other hand yields considerable energy savings when compared to the other diffusion variants. It consumes less than half the energy for the same period of time. This means that FLIP-optimized diffusion could double the lifetime of a sensor network when compared to "plain" diffusion. Table 2.3 summarizes energy consumption results for the different diffusion variants.

We should point out that, when implementing the original diffusion protocol, diffusion developers were likely not trying to implement a completely optimized protocol.

Table 2.3: Diffusion variant energy consumption

	Energy consumed	packet size
Diffusion	0.109	36
Diffusion + flip	0.114	38
Optimized Diffusion	0.050	varies (21 - 25)

The original diffusion header includes extra functionality that our optimized header does not provide as it is not required by this diffusion implementation. Of course, if this functionality is required by future diffusion variants, it can easily be incorporated by FLIP.

It is also noteworthy the fact that FLIP's optimization of diffusion which results in three different types of packets to implement diffusion's interest, reinforcement, and data exchanges also showcases FLIP's ability to accommodate heterogeneity.

2.5 Effects of Flexible Headers

In this section, we demonstrate FLIP's energy efficiency in the context of another data gathering application for sensor networks. The specific scenario we consider is temperature running average calculation.

We designed a simple data gathering protocol which works as follows. A requesting node sends a query for a certain variable, for example temperature. Each node then sends back an answer reporting their current temperature measurement. The requesting node can then perform some calculation over the requested data. This calculation might be something like finding the average over each round of reported temperature values. This is a simplified example since this average would not take into account node location. Nodes perform two basic exchanges: the query that originates at requesting nodes, and the resulting replies. The `TTL` is decremented at each hop. We will explain the use of the `TTL` in Section 2.6. We describe the different packet formats below.

2.5.1 Header Models

We compare FLIP's flexible headers with two static header models: *minimal* and *full* headers. Figure 2.14 show the three header models considered.

In FLIP, the query packet header consists of `source` (2 bytes), `TTL` (1 byte), and `type` (1 byte). The response header includes `destination` (2 bytes) and `type` only. Query and response header sizes (including meta-headers) are 6- and 4 bytes, respectively. The payload in the two cases is 2- and 4 bytes long. Query packets carry

the query id; in the case of response packets, the data being reported is also included. This makes the packets 8 bytes long. Ring nodes, defined below, reset FLIP's meta-header bit corresponding to the TTL field.

Minimal static headers are 6 bytes long. They consist of the union of all FLIP header fields, i.e., `source`, `destination`, `type`, and `ttl`. The corresponding query packet is 8 bytes, like in FLIP. But responses are 10 bytes long.

The full header mode includes all fields typically present in "traditional" network-level protocols: `version`, `source`, `destination`, `type`, `TTL`, `size`, `CRC`, and `sequence number`. The total header size is 15 bytes which makes queries 17- and responses 19 bytes long.

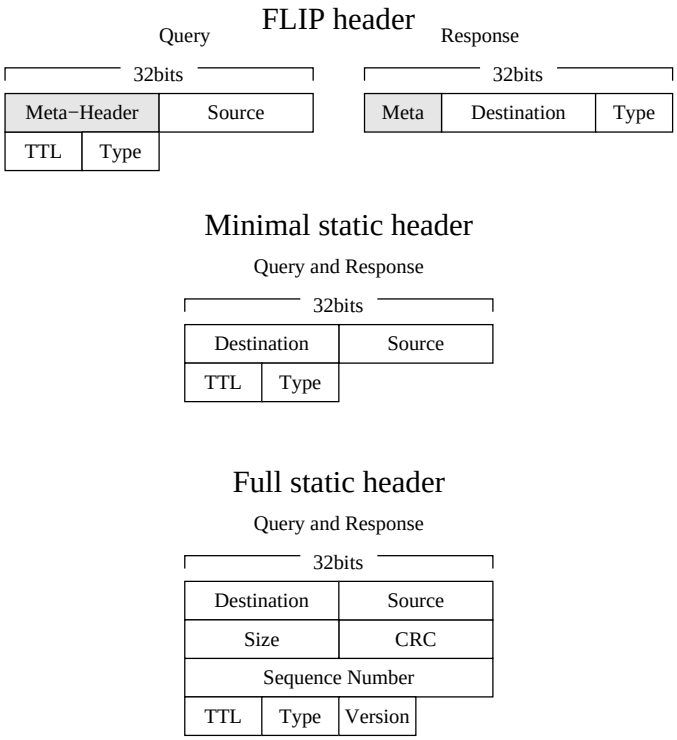


Figure 2.14: Header models

In this particular application scenario, since flows in different directions need to carry different information, FLIP's flexible headers are able to minimize protocol overhead, while not limiting protocol functionality. As our simulation results show, FLIP yields the highest energy efficiency even when compared to the minimal header

model case.

2.5.2 Simulation Results

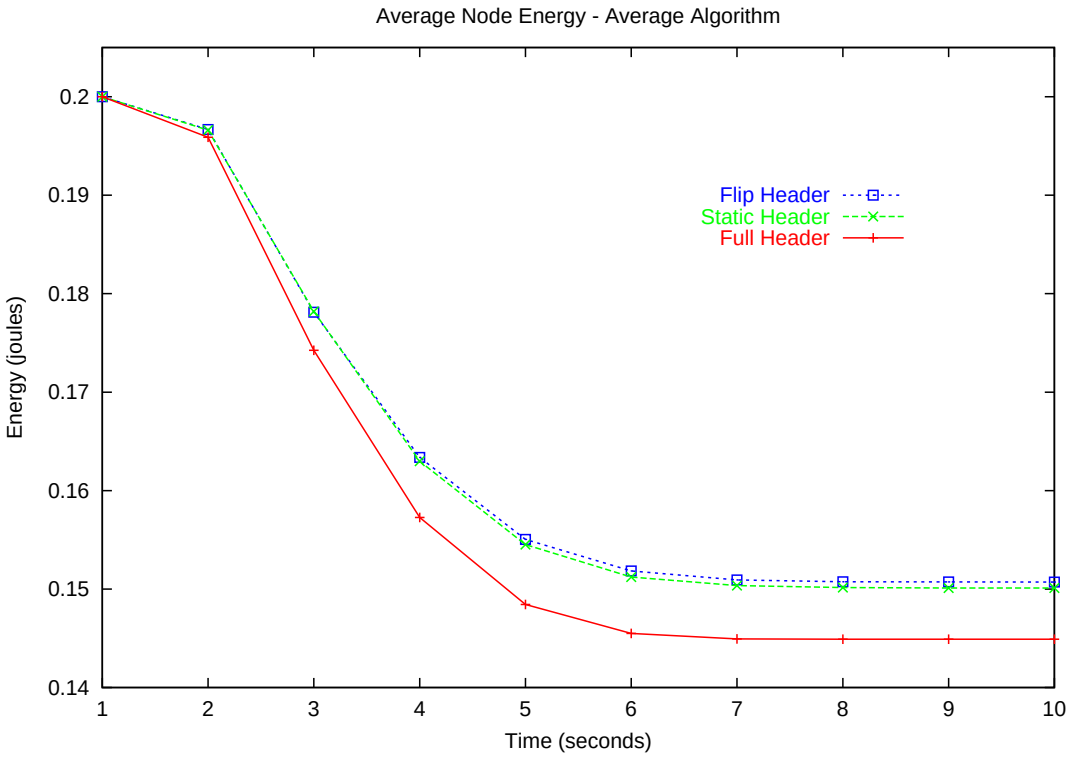


Figure 2.15: Data gathering (temperature averaging) energy consumption

The graph in Figure 2.15 shows how average node energy varies over time for the temperature averaging application. Similarly to the directed diffusion experiments, we use a 300-node sensor network spanning a 2000 x 2000 meter area. We also use `ns-2`'s 802.11 MAC protocol and the same radio energy consumption parameters, i.e., 395mW, 660mW, and 0 for receive, send and idle, respectively. As before, these parameters are based on the empirical values used in the original evaluation of diffusion [5]. Reported data points are averages over 10 runs. Initial node energy is 0.2 Joules. The experiment consists of running the average calculation algorithm once, where a node sends a query and waits for the responses. There are no delivery guarantees of any kind nor recovery mechanisms addressing node failures.

Table 2.4: Total energy consumption for the data gathering application

	Energy consumed	Query size	Response size
FLIP header	0.0493	8 (7)	8
Minimal static header	0.0499	8	10
Full static header	0.0551	17	19

In the initial part of the experiment (between 0 and 2 seconds) average energy consumption is low since nodes are only sending the query packet away from the requester node. As more and more nodes reply to this packet and forward the responses, energy consumption increases. After a few seconds the energy levels off as packets arrive at the requester or are lost due to collisions. The number of readings collected by the requester were similar for all three header models with an average of 266 readings out of (maximum) 300.

Table 2.4 summarizes the energy consumption results for the different header models. The minimal static header model consumed 1.2% more energy than FLIP, while full headers consumed 11.8% more. Both the minimal header protocol and FLIP provide exactly the same functionality, which is optimized for the data gathering application. The full static header model on the other hand includes the usual fields present in "traditional" network-layer protocols. This extra, but dispensable, functionality results in almost 12% additional energy consumption when compared to FLIP. For applications that need some or all the functionality provided by a full header model, FLIP could easily add the required fields.

One can argue that it is possible to use a different static header for each protocol exchange. To this end, the protocol still needs a way to differentiate between the different packet types. For example, nodes can use the packet type field to decide how to process a packet. However, we claim that if protocol designers are willing to make packet processing more complex, they will be better off using FLIP, which is fully customizable. As demonstrated by our results, FLIP's meta-header provides an efficient way to define which fields are included in the header.

2.6 Data Aggregation

Previous sections showed that FLIP's flexible headers are an effective power-conservation mechanism. The goal of this section is to showcase FLIP's flexibility as a way to incorporate new protocol functions seamlessly. As an example, we modify the data gathering protocol described in Section 2.5 to include data aggregation. We demonstrate that FLIP's ability to incorporate new functionality may lead to a more (power-)efficient protocol.

We assume applications where information from nodes closer to the requester is considered more important than information from farther away nodes. An example application that falls in this category is monitoring a controlled chemical reaction (e.g., temperature), where data from sensors close-by to where the reaction is taking place is more critical than data from sensors farther away. This means that information from close-by nodes should be received as soon and as accurately as possible. Information from more distant nodes is not so critical and can be delivered later. The objective then becomes to optimize energy efficiency while still delivering important data in a timely fashion.

Our data aggregation mechanism works as follows. The requester node defines an area it considers important. It does so by setting the TTL of the query packet, which defines the hop count of the *importance area*. At every hop, the TTL is decremented by one. Once it reaches zero, it means the packet left the importance area. After this point the packet no longer needs the TTL field since it already knows it is far away from the requester. Figure 2.16 shows a sample scenario. The central (gray) node is the requester and the dashed circle defines its transmission range. The shaded area is an approximation of a 2-hop importance area. The black, or *ring* nodes delimit the importance area.

Nodes reply as soon as they get a request. Nodes inside the importance area will forward these replies immediately so they reach the requester as soon as possible. Nodes outside the importance area will reply and forward other nodes' replies at their leisure. In our experiments, outside nodes also reply immediately. However, instead of forwarding immediately, ring nodes aggregate replies into a single packet which they forward to the requester when new data is received.

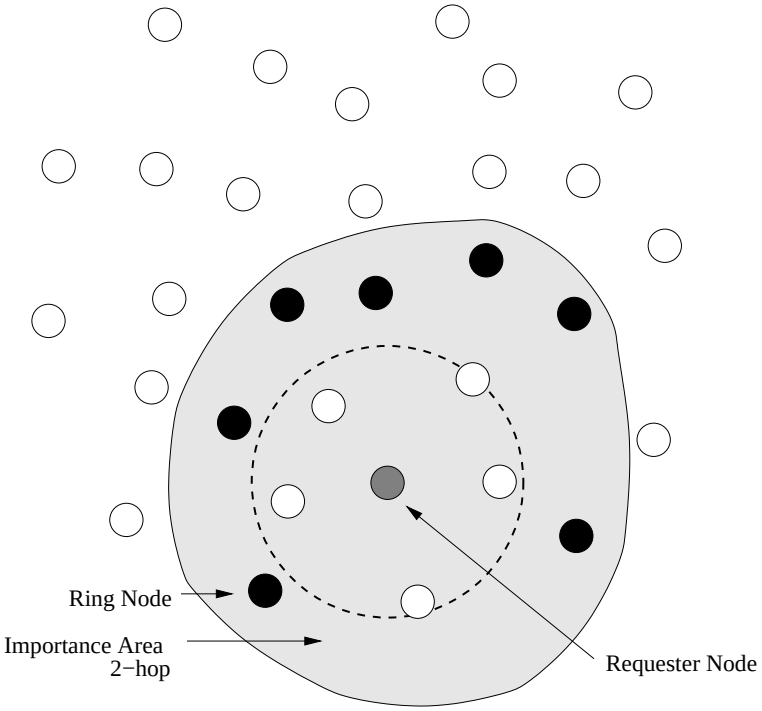


Figure 2.16: Sample ring aggregation scenario

Data aggregation at ring nodes compensates for energy consumption at nodes within the requester's importance area. Of course the tradeoff is that information from farther away nodes is delayed. However, since this information is not considered critical, the added delay is tolerated. The same aggregation technique can be applied to other data-driven applications such as hierarchical mapping algorithms which require accurate information from close-by nodes and are tolerable to less accurate readings from distant nodes.

In these experiments, we used the same simulation parameter values as described in Sections 2.4.1 and 2.5. The radius of the importance area (number of hops between requester and ring nodes) was set to 4. Periodic messages from ring nodes to requester are sent every 0.5 seconds.

Figure 2.17 shows the effects of data aggregation when applied to our average computation protocol. Note that the resulting energy level graphs for data gathering with and without aggregation exhibit similar shape. However, data aggregation results

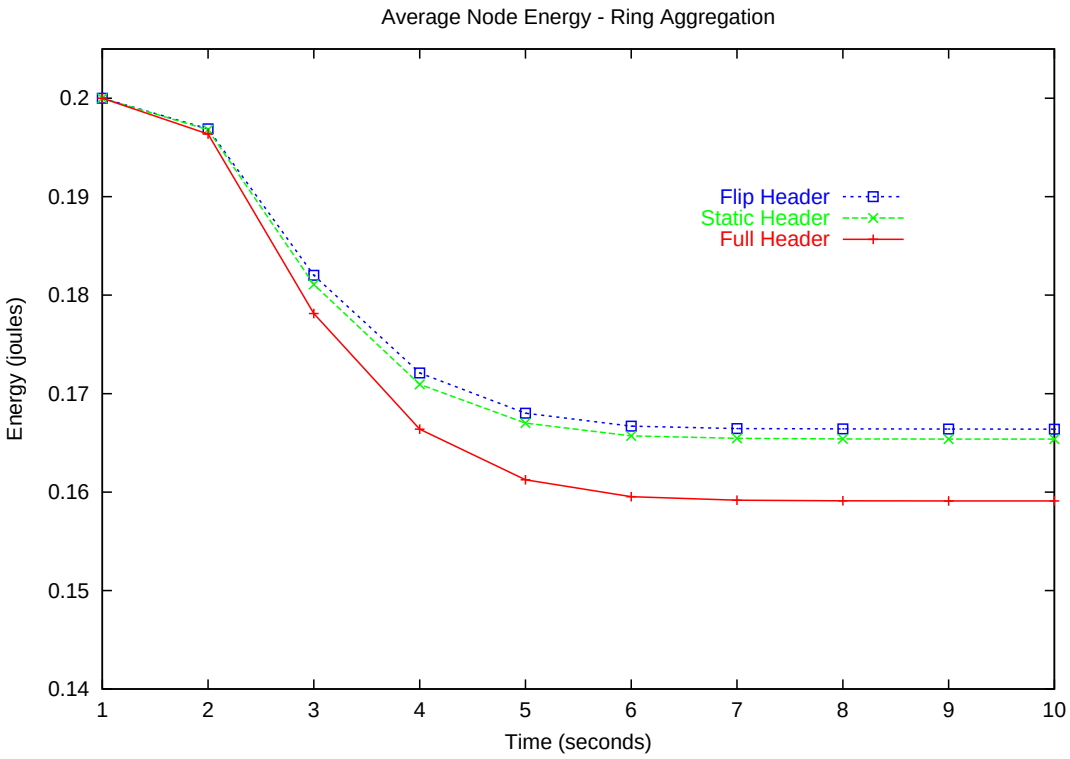


Figure 2.17: Data gathering (temperature averaging) energy consumption with data aggregation.

in lower power consumption as nodes inside the importance area end up sending fewer packets when aggregation is used.

Table 2.5 shows total energy consumption with aggregation for the different header models. FLIP's energy savings is 1.1% higher than the minimal static header model and 6.0% higher than full headers. Thus FLIP is able to achieve comparable energy efficiency to a fully optimized protocol and still offer functionality provided by full header models. We should also point out that, for all header models, the requester collected similar number of readings (averaging 284 out of the original 300 readings transmitted), which measures the data accuracy obtained by aggregation. Figure 2.18 demonstrates the energy savings obtained by aggregating data by comparing the energy consumption of propagating data with- and without aggregation. Aggregation, in this experiment, uses FLIP as the underlying protocol.

We used aggregation as an example functionality that can be incorporated into

Table 2.5: Total energy consumption for data gathering application with aggregation for different header models.

	Energy consumption		Energy
	no aggregation	aggregation	savings
FLIP header	0.0493	0.0336	31.8%
Small Static header	0.0499	0.0346	30.7%
Full Static header	0.0551	0.0409	25.8%

an existing protocol and showed that it has considerable impact on energy conservation. This aggregation example also demonstrates how FLIP allows higher-layer protocols to add and remove functionality as needed. In the case of the ring aggregation example, when `TTL` field was no longer needed, it was removed, decreasing the header overhead.

Protocol designers have to make choices. If they choose to optimize the protocol in excess, they might make it very hard (if no impossible) to add future enhancements/functionality as the protocol evolves. On the other hand, if they try to provide complete functionality (as in the full static header), they will undoubtedly incur unnecessary overhead in most or all cases. FLIP permits a balance between the functionality provided by full header models and optimized overhead achieved by minimal headers.

2.7 Related Work

Communication protocols for wireless networks have been an active area or research and include efforts such as Packet Radio [43], GloMo [6] and the IETF's Mobile Ad-hoc Networks (manet) working group [16]. In the early 90's, several efforts focused on the concept of "ubiquitous computing" [47]. Some examples include a number of projects at Xerox PARC [48] and the Daedalus/BARWAN project [7] at UC Berkeley. More recently, some research has turned to embedded systems and sensor networks. To our knowledge, FLIP is the only initiative to develop a protocol to interconnect heterogeneous devices. Almeroth et al. [25] introduced the main concepts behind FLIP.

AT&T Laboratories Cambridge (former Olivetti Research Labs) has lead several related initiatives including: Piconet [8] and its low-range radio network, an infrared (IR) network [14] connecting active badges and IR-base sensors. Their efforts to

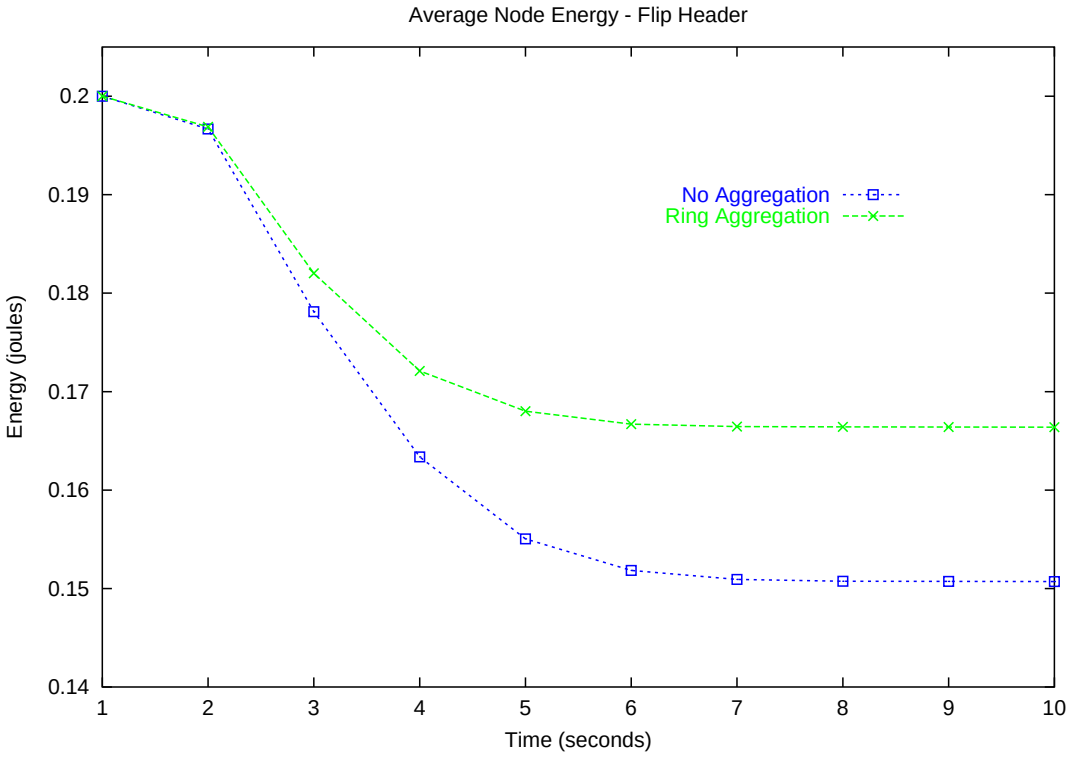


Figure 2.18: Effect of data aggregation on energy consumption

develop a low-power protocol stack [10] focuses on optimizing the MAC layer for low-bandwidth, low-power systems. They have also developed services atop these networks, including the Active Badge location system [46], and the Active Floor [3].

In the Scalable Coordination Architectures for Deeply Distributed Systems (SCADDS) project [40], nodes loose their individuality and the focus lies in the data generated by the whole system. In this context, the *directed diffusion* [5] architecture was developed to convey data from information sources (e.g., sensors) to information sinks. Directed diffusion uses its own protocol which is specially tailored to its needs. In Section 2.4.1, we use FLIP to implement diffusion and demonstrate FLIP's energy efficiency when compared with an existing diffusion implementation.

The Dynamic Sensor Networks (DSN) project [41] is also designing and implementing a protocol specially tailored for their sensor network application. DSN aims to take advantage of GPS in sensor networks and therefore their MAC-layer protocol uses

a GPS-based TDMA while their network-layer protocol uses GPS for spatial addressing and routing. The WINS project, Wireless Integrated Network Sensors [11], describes a basic sensor network environment and presents a solution based on layered processing.

Research such as the Address Free Architecture [20] is related to FLIP as it proposes new approaches to providing network-layer functionality, in this case addressing. They describe an architecture where nodes select probabilistically unique addresses in order to uniquely identify data flows at any point in time.

There is also the BlueTooth [1] consortium effort whose primary goal is to develop low-cost, low-power radios with link ranges on the order of a few meters. The goal is to implement this technology into cheap chips to be plugged into computers, printers, mobile phones, etc.

There has also been work on header compression. The recently proposed Unified Header Compression Framework [22] aims at creating a standard way in which protocols in general can define header compression. Previous work [29][30] targeted specific protocols such as TCP/IP. Unlike FLIP, current header compression schemes require persistent data exchange between endpoints. A full-header packet establishes state and then subsequent packets can be compressed. Another limitation of current compression schemes is that they are intended for mostly point-to-point communication.

2.8 Conclusions

This chapter described the design and implementation of FLIP, a network protocol whose goal is to accommodate varying capability devices. FLIP uses customizable headers to satisfy, with minimal overhead, the requirements of a wide-range of applications and devices. We implemented FLIP under Linux and used the BSD socket abstraction to make FLIP available to application programmers.

We evaluated FLIP in a number of scenarios. First we compared FLIP's overhead and functionality against (IPv4 and IPv6). We showed that when providing IP functionality, FLIP incurs relatively small overhead (1 and 3 bytes respectively), yet provides close to minimal overhead in scenarios that require less functions than what IP provides (specially when carrying small payloads). We presented the Generic Transport Protocol, or GTP, a flexible transport layer protocol that runs atop FLIP. We com-

pared GTP to TCP/UDP and showed that it yields increased efficiency when providing transport level functionality for different application needs. GTP's efficiency is a direct consequence of its ability to provide only the functions needed by target applications.

We also evaluated FLIP in the context of sensor network environments. In particular, we used FLIP to implement the directed diffusion communication paradigm. In the first set of experiments, we performed direct translation between diffusion and FLIP header fields. We observe a slight increase in the resulting protocol's overhead due to FLIP's meta-headers. We argue, however, that even though using FLIP is slightly more energy consuming, it would pay off if there is the need to interconnect different types of devices with different capabilities. We then re-designed diffusion assuming FLIP as the underlying network protocol. Using FLIP's flexible headers, we were able to provide just the required functionality incurring minimal protocol overhead. Simulation results show that *optimized diffusion* can be 50% more energy efficient than original diffusion.

Data gathering applications in sensor networks were the other scenario we used to evaluate FLIP. We designed a simple protocol to perform running average calculation and compared the efficiency of FLIP's flexible header against static headers. We used two static header models: *full* headers include most fields present in "traditional" network-layer protocols, while minimal headers, which are optimized for the target application, only carry required fields. We showed that FLIP is more energy efficient than both header models. It outperforms optimized static headers by a small margin and still has the additional advantage of being able to accommodate other devices if needed. FLIP is able to match the functionality of the full header model and yet yields 12% higher energy efficiency.

As networks become more heterogeneous, FLIP's flexibility allows devices of widely varying power, communications, and processing capability to be networked together. We also showed FLIP's ability to evolve seamlessly and include new protocol functionality as needed. We demonstrated that FLIP's ability to incorporate new functionality may lead to a more (power-)efficient protocol. To this end, we enhanced the running average calculation protocol by adding data aggregation. When compared to the original version of the average calculation protocol, data aggregation reduced the

system's overall energy consumption by as much as 30%. The addition of this feature required the use of the `TTL` field. `TTL` (or any other fields) could be easily incorporated into the FLIP header. Had any of the static header protocols not been implemented with this feature from the start, they would not have been able to take advantage of such an enhancement.

This highlights the fact that, good design (including plans for protocol evolution) is of extreme importance. However, no matter how much protocol designers plan, they are not able to predict all possible features a protocol should have. One good example is IP evolution. IP designers predicted that IP's (IPv4) address space would likely last for several more decades. Now we know this is not the case and to fix that limitation, IPv6 was born. Given that the Internet became a complex, intricate communication infrastructure whose uninterrupted operation is critical, deployment and compatibility with IPv4 are the big challenges faced by IPv6. Flexible protocols such as FLIP enables application-specific optimization leading to maximal protocol efficiency, and yet allows seamless protocol evolution.

In the following chapters we will use FLIP as the underlying network protocol, optimizing our exchanges to take advantage of the flexibility it gives us. Future work on FLIP includes defining a generic routing structure and implementing it for use in devices like the Motes developed by USC/ISI and UCLA. This will allow the devices to take advantage of the FLIP packet structure flexibility. Our work on FLIP has had conference and journal publications [15][34] [35].

Chapter 3

Data Collection - Optimizing through aggregation and timing

This chapter explores in-network aggregation as a power-efficient mechanism for collecting data in wireless sensor networks. In particular, we focus on sensor network scenarios where a large number of nodes produce data periodically. Such communication model is typical of monitoring applications, an important application domain sensor networks target. The main idea behind in-network aggregation is that, rather than sending individual data items from sensors to sinks, multiple data items are aggregated as they are forwarded by the sensor network.

Through simulations, we evaluate the performance of different in-network aggregation algorithms, including our own *cascading timers*, in terms of the trade-offs between energy efficiency, data accuracy and freshness. Our results show that timing, i.e., how long a node waits to receive data from its children (downstream nodes in respect to the information sink) before forwarding data onto the next hop (toward the sink) plays a crucial role in the performance of aggregation algorithms for applications that generate data periodically. By carefully selecting when to aggregate and forward data, *cascading timers* achieves considerable energy savings while maintaining data freshness and accuracy. We also study in-network aggregation's cost-efficiency using simple mathematical models.

Since wireless sensor networks are prone to transmission errors and losses can have considerable impact when data aggregation is used, we also propose and evaluate

a number of techniques for handling packet loss. Simulations show that, when used in conjunction with aggregation protocols, the proposed techniques can effectively mitigate the effects of random transmission losses in a power-efficient way.

Our hypothesis is that timing models play a crucial role in the accuracy and freshness delivered by data aggregation. In this chapter, we study how different timing schemes affect the performance of in-network aggregation algorithms. Based on their timing model, we classify existing periodic data aggregation protocols into three categories, namely: *periodic simple*, *periodic per-hop*, and *periodic per-hop adjusted*.

Periodic simple aggregation works by having each node wait a pre-defined period of time (referred to as *timeout*), aggregate all data items received, and send out a single packet containing the result. As discussed in Section 3.6 below, the directed diffusion [5] sensor network communication paradigm belongs to this category. Aggregation mechanisms in the *periodic per-hop* category have nodes send the aggregated packet as soon as they hear from all their children. A maximum timeout interval equal to the data generation period is used in case reports get lost. Finally, *periodic per-hop adjusted* uses the same basic principle of *periodic per-hop* but schedules a node's timeout based on its position in the distribution tree (rooted at the information sink and spanning all reporting- as well as appropriate intermediate nodes). Our own *cascading timers* aggregation mechanism falls within this category, and, when compared to other existing *periodic per-hop adjusted* algorithms, presents benefits such as not requiring clock synchronization among nodes and minimizing timer scheduling overhead. *Cascading timers* schedules a node's timeout based on the time it takes for a packet to travel a single hop, or the *single hop delay* and the number of hops to reach the sink. We also study how the value selected for the *single hop delay* impacts the performance of *cascading timers*.

In summary, contributions include: (1) development of *cascading timers* aggregation for periodic data generation applications including a detailed analysis of *cascading timers*' dependence on per-hop delay, (2) trade-off analysis of in-network data aggregation using simple analytical models (3) comparative performance study of different aggregation algorithms using extensive simulations, and (4) development of different loss recovery mechanisms and study of how they impact performance of data aggregation under lossy environments.

For evaluating the performance of the different in-network aggregation mechanisms, energy efficiency, data accuracy and freshness, and communication overhead are used as performance metrics. We should also point out that, unlike previous evaluation studies targeting sensor network protocols, a wide range of network scenarios including different information sink placement strategies are used.

The remainder of this chapter is organized as follows. The next section discusses related work including existing periodic data aggregation mechanisms. In-network aggregation with *cascading timers* is described in Section 3.1. Other aggregation types are mentioned in Section 3.2. Section 3.3 investigates in-network aggregation's cost-efficiency using simple mathematical models. Section 3.4 describes the simulation experiments we conduct to compare the performance of different in-network aggregation algorithms, including the experimental setup used, results obtained, as well as the impact of the per-hop delay on the performance of *cascading timers*. Techniques for handling packet losses and their performance are introduced in Section 3.5. Section 3.6 discusses related work. Finally, Section 3.7 presents our concluding remarks and directions for future work.

3.1 Cascading Timers Aggregation

As previously discussed, our *cascading timers*[36] aggregation algorithm targets periodic data generation applications in which nodes produce data at regular periods. A given node aggregates data received from its children into a single data item, which is then forwarded upstream towards the information sink[1]. Application scenarios that fit well within this communication model include monitoring of continuous environmental conditions like temperature, humidity, seismic activity, etc. While we focus on the single information sink scenario, the proposed technique can be applied to multi-sink scenarios.

Some of *cascading timers*' design goals include:

- Simplicity: given that sensor network nodes are typically anemic devices regarding energy, processing, storage, and communication capabilities, designing simple aggrega-

[1] As explained in more detail below, data is aggregated over a tree rooted at the information sink

tion algorithms is key.

- Efficiency: generate close to minimal control overhead. Again, this is a critical requirement in the resource-constrained environments our algorithms target.

- No clock synchronization: *cascading timers* does not require clock synchronization among nodes. No matter how efficient clock synchronization mechanisms become, they will require additional message exchange among nodes and thus incur additional energy consumption. Efficient synchronization algorithms [21] have emerged recently and might allow other tradeoffs.

- Routing protocol independence: no specific underlying routing protocol is assumed.

Similar to most periodic aggregation mechanisms, *cascading timers* starts by having the sink broadcast the initial request to all nodes. This initial request triggers a simple tree establishment protocol which sets up reverse paths from all nodes back to the sink or root of the tree. Upon receiving the request message, nodes send a reply back to their parent in the tree. Each node can then deduce how many children it has. Nodes assume a broadcast medium and forward data using one-hop broadcasts. In order to avoid collisions, transmissions are scheduled using a small staggering delay. The setting of the staggering interval will be discussed in detail in Section 3.4.3.

Note that tree establishment is essentially the overhead incurred by *cascading timers* and most other in-network aggregation mechanisms. Even if no aggregation is employed, a distribution tree is typically used to collect data from information sources to sinks.

In *cascading timers*, instead of having nodes schedule randomly their timeout, i.e., the time interval they wait to receive data from their children before forwarding the next data aggregate, a node's timeout is set based on the node's position in the data distribution tree. Thus, a node's timeout will happen right before its parent's. This causes the so-called "cascading" effect: data originating at the leaves is clocked out first, reaching nodes in the next tree level in time to be aggregated with data from other leaf nodes and locally generated data, and so on. The net effect is that a "data wave" reaches the sink in one period. This is the main reason behind *cascading timers*'

ability to achieve power efficiency and yet deliver fresh data at sink nodes.

Timeout scheduling is part of the distribution tree setup protocol and is triggered by the initial request from the sink. The sink's request contains a "hop count" field which gets incremented as the request travels toward the leaf nodes. Using this hop count information, nodes can estimate their distance, in time, to the sink and schedule their timeout to produce the cascading effect.

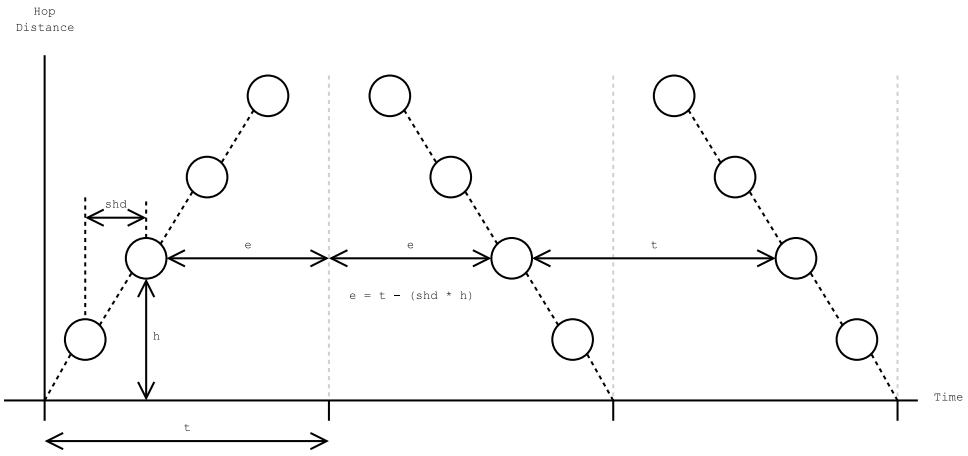


Figure 3.1: *Cascading timers* timeout calculation

Figure 3.1 shows graphically timeout calculation in *cascading timers*, where t is the data generation period, h is a node's distance to the sink in number of hops, and *shd*, the *single hop distance*, is the delay to traverse one hop. Once the request packet is received, a node schedules its timeout to happen after $2e$. Subsequent timers will continue to be scheduled every t interval.

Note that a node's timeout depends on the *single hop distance*, or *shd*. We investigate this dependence in detail in Section 3.4.3 and show how it affects the performance of the algorithm. As previously pointed out, *cascading timers*' timing scheme is parallel to the ones employed by both TAG and Covergecasting. According to our taxonomy, all three mechanisms are classified in the *periodic per-hop adjusted* category. In our simulations, we compare the different aggregation techniques. Our *cascading timers* and TAG represent *periodic per-hop adjusted* algorithms.

Our implementation of *TAG* tries to follow their algorithm as closely as possi-

ble. The data generation period is equivalent to TAG's epoch. We divide the generation period by an estimate the number of hops. To avoid collisions, nodes transmit at a random uniformly distributed time within the slot corresponding to their height on the aggregation tree.

3.2 Other Periodic Aggregation Mechanisms

Below we describe in detail the other classes of aggregation algorithms we use in our comparative study. As baseline, we employ no in-network aggregation when sending data from information sources to the sink. As previously pointed out, even in the no-aggregation case, we employ a distribution tree rooted at the information sink and spanning all (relevant) data sources. As packets flow from the leaves to the root, nodes simply forward them along the tree.

3.2.1 Periodic Simple

Nodes in *periodic simple* aggregation wait a pre-defined amount of time, aggregate all the data received in that period, and send out a single packet. The aggregation period is equal to the data generation period, which, for most our simulation experiments, is set to 1 second.

This class of aggregation protocols represents the basic mechanism used by Directed Diffusion [5] considering that all nodes have relevant data to send. Based on feedback (or reinforcements) from the sink, every node uses a specific gradient which determines the rate at which data is sent to the sink. Note that nodes are not necessarily synchronized when "clocking out" data.

3.2.2 Periodic Per-Hop

According to *per-hop simple* aggregation, once all data items are received from a node's children in the distribution tree, an aggregated packet is produced and sent onto the next hop. Each node uses a timeout for sending out packets in case their children's response is lost. The timeout is equal to the data generation period since once that time is up, we will be expecting and producing new readings.

3.3 Tradeoff Analysis

In this section, we study the performance tradeoffs raised by in-network aggregation. Using simple mathematical models, we conduct a cost-efficiency analysis of data aggregation.

3.3.1 Energy Efficiency

Recall that data aggregation's main goal is to achieve energy efficiency. It does so by reducing the number of packets transmitted. Ideally, when aggregation is employed, only a single packet is sent by each node per data collection period, or round. Thus the number of packets sent per round, or $AggPkt/Round$, is given by Equation 3.1.

$$AggPkt/Round = n \tag{3.1}$$

$$NoAggPkt/Round = \sum_{i \in N} d_i \tag{3.2}$$

$$= \sum_{d=1}^{Max(d)} d * N_d \tag{3.3}$$

Without aggregation, each node will send a packet that will be forwarded to the sink. Each hop traversed by a packet counts as one packet being sent, hence, the cost of getting a reading from a node equals its height on the data distribution tree. The number of packets sent, or $NoAggPkt/Round$ is thus given by Equation 3.2, where d_i is the depth of node i and N is the set of participating nodes. Alternatively, we can compute the number of packets transmitted per round as a function of the number of nodes at every tree level. This is described by Equation 3.3, where d is the number of tree levels and N_d is the number of nodes at depth d.

Clearly, in an average scenario, no-aggregation will send more packets per round. No-aggregation's best case scenario is when the data distribution tree has maximum depth ($Max(d)$) equal to 1. The number of packets sent per round without aggregation is equal to n (Equation 3.2). This confirms that in "shallow" distribution trees, the benefits of aggregation are not as significant. But it never performs worse than no-aggregation.

On the other hand, in deep trees, aggregation has significant impact in reducing the number of packets transmitted. The worst case scenario for no-aggregation is a "line" topology, a tree where every node has only one child. In the case of a line topology with depth n, the summation in Equation 3.3 results in $\frac{(n)(n+1)}{2}$ packets per round.

Essentially, these are the lower and upper bounds for the number of packets sent when no-aggregation is used. As demonstrated in Section 3.4.2, all the aggregation algorithms studied are able to achieve ideal energy efficiency by sending only one packet per node per round. The difference in performance between them lies in the data accuracy and freshness they achieve.

We assume that an aggregated packet will have the same size as a non-aggregated one. This is the case of operations like computing averages, selecting the maximum or minimum value, etc. This implies that, as packets flow toward the sink, they will not grow in size. Hence, our energy efficiency metric computes number of packets-, rather than number of bytes sent.

On the other extreme, if aggregation by concatenation is employed, i.e., data items are concatenated as they traverse the sensor network, there will still be savings on the number of packets transmitted, but the number of bytes sent will not be reduced. Nevertheless, aggregation will still be advantageous in terms of medium acquisition and scheduling. *Cascading timers* will also yield improved data freshness due to its low delay.

3.3.2 Complexity

Note that tree establishment is essentially the overhead incurred by *cascading timers* and most other in-network aggregation mechanisms. Even if no aggregation is employed, a distribution tree is typically used to propagate data from information sources to sinks.

In terms of communication complexity, tree establishment costs n packets as each node disseminates the original query that triggers formation of the tree. If the protocol also generates a reply from every child, there will be additional $n-1$ packets since every node, except the root, will be a child. Tree establishment's total cost is then $2n-1$ packets.

Nodes reply back to their parents when they receive the original query so that each node knows how many children it has. This information is used to optimize the algorithms considered by allowing a node to know when it has received the readings from all its children. If this optimization is not performed, tree establishment cost is reduced to n packets, which is equal to the cost of tree formation for no-aggregation. Note that nodes can still find out how many children they have as the algorithm runs. In both cases, the algorithm is able to handle new nodes joining the tree as well as existing nodes leaving/failing.

In order to adapt to topology changes, tree re-establishment is performed, which will cost n (or $2n - 1$ if nodes reply back to their parents) packets. Of course, localized tree re-establishments can be performed so as to reduce overhead.

All algorithms we present here (including no aggregation) incur these tree formation costs. Hence, under the conditions used in the experiments reported in Section 3.4, the extra cost of using in-network aggregation over no-aggregation could be $n - 1$ additional control packets if we want nodes to know how many children they have right from the start of the algorithm.

In terms of storage complexity, depending on the aggregation operator used, readings from a node's children may need to be stored locally. In our example application, aggregated data can be stored as a single data item which will be sent by the node on timer expiration or when all its children's readings are received. This requires at most one data item size storage unit. In the case of aggregation by concatenation, a node needs as many data item size storage units as the number of children it has. For instance, a scenario with a single 64-bit data value plus 64-bit for control information, assuming an average number of children of 32, would require a total of 512-byte storage. The computational complexity is also trivial for most cases where simple aggregators (e.g., calculating the minimum, maximum, average, sum, etc.) are employed.

3.4 Simulations

For our comparative study of the different in-network aggregation algorithms, we ran extensive simulations using the `ns-2` network simulator [44].

3.4.1 Experimental Setup

In the experiments we conducted, 100 nodes were randomly placed in a $500 * 500m_2$ area. Nodes' transmission range and data rate are set to 100 meters and 115 Kbps, respectively. 802.11b's broadcast mode is used as the MAC-layer protocol and FLIP [35] as the network protocol so we can take advantage of its optimized headers. Based on values used by commercially available radios, we set transmission and reception power levels to 24.75 and 13.5 milliwatts, respectively. Idle power consumption was set to 0.675 milliwatts to reflect an optimized MAC layer, i.e., MAC protocols that switch to low-power radio mode whenever possible.

In order to avoid collisions, nodes stagger their transmissions using a small random interval. This is important when performing data collection over a tree, especially when nodes try to send at scheduled intervals based on their depth in the tree. The maximum staggering value used was 0.03 seconds. Nodes pick a uniformly distributed random timer between 0 and this value before sending. In Section 3.4.3, we discuss the setting of the staggering interval in more detail.

Nodes are stationary and no transmission errors were simulated for the first set of results. However, packets can still be lost due to collisions. Mechanisms to handle packet loss and their performance are reported in Section 3.5 below. Simulations were run for 20 seconds with data being generated every second (round). Although establishing the distribution tree can be initiated by the data request from the sink, in our simulations the tree was formed at time 1 second and data collection was triggered by the sink at time 3 seconds. We present steady state results, that is, measurements taken during the second half of the simulation (during the last 10 seconds).

Data points were obtained by averaging over twenty different runs using different seeds to perform random node placement. As will be evident in our results, information sink placement can greatly affect the performance of tree-based aggregation algorithms. For this reason, we ran experiments using three different sink placement strategies: corner, center, and random placement. Placing the sink in corners means that the resulting collection trees will be deeper. Center placement minimizes tree height.

Performance metrics we use include **energy consumed**, **data accuracy**,

data freshness, and **overhead**. While energy consumed measures the algorithm's energy efficiency, data accuracy and freshness account for its effectiveness in terms of conveying as much information as possible to the sink in a timely manner.

In these experiments, we do not model the actual values being sensed by the nodes, how fast they are changing or in what manner. Therefore, accuracy is measured as the ratio of total number of readings received at the sink to the total number of readings generated. We assume lossless aggregation, that is, no data is discarded. Examples include computing the minimum, maximum, as well as counting. In these scenarios, total accuracy is achieved when the sink can "calculate" an answer that involves one reading from every node per round.

Freshness is computed as the difference between the round a data item is generated and the round it is received at the sink. Overhead measures the communication complexity of the in-network aggregation algorithms.

3.4.2 Results

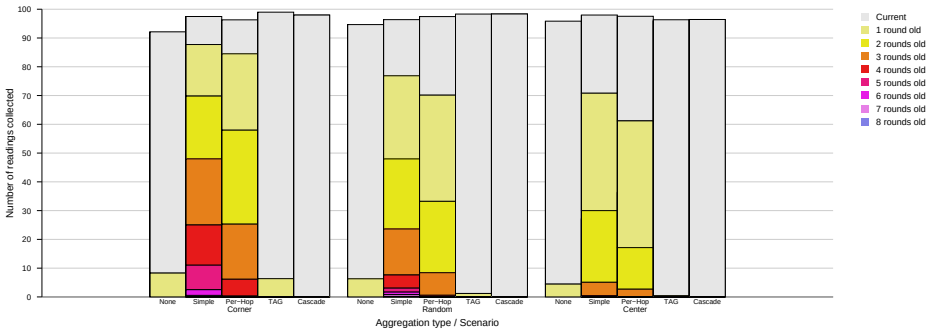


Figure 3.2: Data accuracy and freshness

Figure 3.2 shows the freshness and accuracy of the aggregation algorithms. *Periodic simple* is labeled as "Simple", *periodic per-hop* as "Per-hop", *TAG* refers to our implementation of TAG's aggregation mechanism, and "Cascade" represents our *cascading timers*. The sink placements evaluated were corner, random and center. The total number of readings collected (bar height) depicts accuracy and the bar divisions

(shades), freshness. For comparison purposes, as baseline we use the no-aggregation strategy (labeled as "None").

In terms of data accuracy, we observe that there is not a big difference in performance when comparing the different aggregation mechanisms. No aggregation and *TAG* exhibit a high percentage of fresh data. *Cascading timers* practically eliminates old data. *Periodic simple* exhibits the largest range of data ages; this is because nodes simply send data periodically, thus it can take up to D periods for the readings to arrive in the worst case, where D is the diameter of the network.

Our implementation of *TAG* is a simple one. We do not calculate the maximum number of hops at runtime and so in some paths the real hop count is higher than our estimation of the network diameter. This leads to TAG having some 1 round old readings. For more accurate calculation we would have required extra message exchanges which we chose not to implement for this experiment.

Even though most data aggregation studies often do not account for sink placement, we observe from Figure 3.2 that sink placement has an impact on data freshness. Even for the no-aggregation case, where packets are forwarded immediately after they are received, placing the sink in the center yields fresher data.

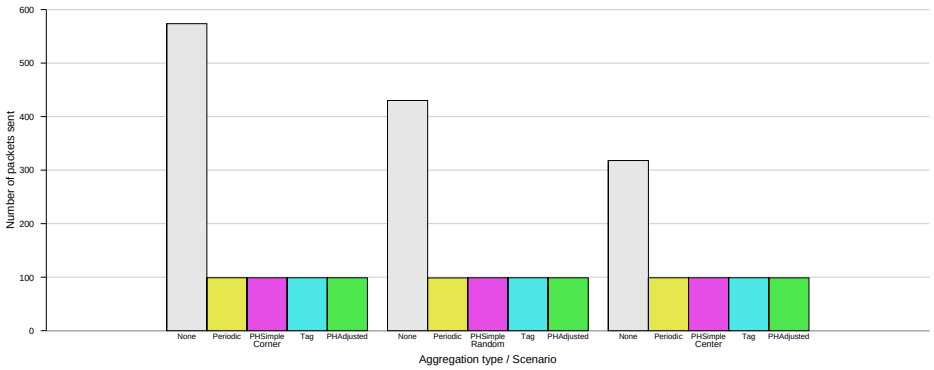


Figure 3.3: Number of data packets transmitted per round

From Table 3.1, which shows the energy consumed by the different algorithms, we observe that, for our experimental setup, energy consumption can be reduced to a

	None	Simple	Per-Hop	TAG	Cascade
Corner sink	0.1418	0.0485	0.0483	0.0467	0.0421
Random sink	0.1302	0.0486	0.0484	0.0464	0.0419
Center sink	0.1134	0.0487	0.0487	0.0454	0.0404

Table 3.1: Energy consumed by the different in-network aggregation algorithms

third when data aggregation is used. Note that all aggregation schemes exhibit similar energy efficiency. These values will be affected by the choices of radio and MAC layers.

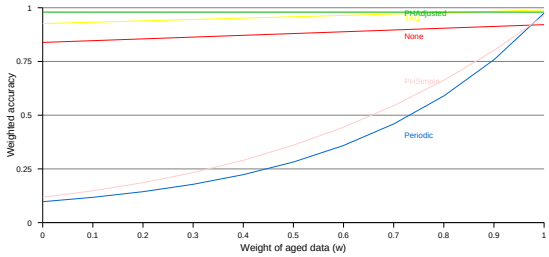
As an another way to compare the performance of the aggregation algorithms with respect to freshness, we introduce a metric that accounts for a data item's age. We call this metric *weighted accuracy*. The motivation behind measuring weighed accuracy lies in the fact that while some applications are interested in historical data, others may only want the most up-to-date information. This is the case of real-time monitoring, where information sinks are only interested in the latest data sensed. For the latter type of applications, aggregation algorithms should not delay data delivery beyond a certain threshold.

To compute weighted accuracy, readings received in the same period they were produced have a weight of 1. Older readings are assigned an exponentially decaying weight: the older the reading, the less weight we assign to it. The expression for weighted accuracy is thus given by:

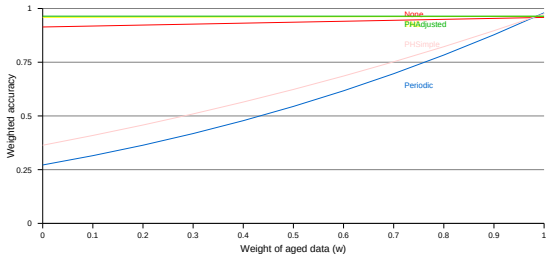
$$weighted\_accuracy = \sum_{i \in I} r_i w^i$$

Where I is the set of ages of the readings, r_i is the number of readings of age i per period and w is the weight. Readings from the current period have an age of 0 and therefore a weight of 1.

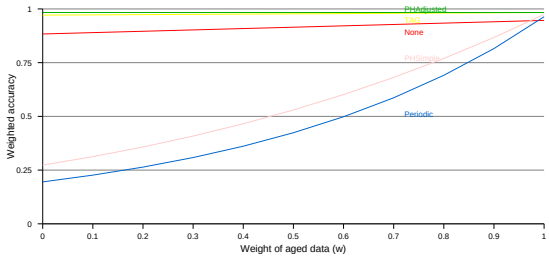
The graphs in Figure 3.4 show the performance of in-network aggregation according to the weighted accuracy metric. As expected, no aggregation, TAG and *cascading timers* exhibit the best weighted accuracy. *Cascading timers* has an edge, specially when weights of old readings are low. *Periodic simple* and *periodic per-hop*



Corner Sink



Center Sink



Random Sink

Figure 3.4: Weighted accuracy

perform poorly if old data has low weight. Their performance increases considerably as we assign higher weight to older information. Under corner sink placement, *periodic simple* and *periodic per-hop* start lower since it takes more periods for the data to arrive. The opposite is true when the sink is in the center.

The delay of a reading provides an alternate way to measure data freshness. We measure the average delay (in seconds) for a given reading as the time interval between when the reading is originally produced by a node until the sink processes all readings generated. Since we are generating results based on all readings we have to wait until all of them are gathered, hence the delay has to factor this in. Table 3.2

	None	Simple	Per-Hop	TAG	Cascade
Corner sink	0.590	2.843	2.080	0.593	0.367
Random sink	0.565	2.047	1.544	0.418	0.286
Center sink	0.545	1.523	1.247	0.298	0.201

Table 3.2: Average reading delay

presents the average delay per reading for our experiments. *Cascading timers* performs the best since this is the very metric it tries to optimize.

Cascading timers exhibits good performance on longer data collection periods as well. For example, in the case of a 10-second collection period using random sink placement, *cascading timers* achieves average delays similar to the ones in table 3.2, $0.28s$. N0-aggregation's delays range around the 5.0 second mark, while *TAG* has an average of $4.23s$ delays. These results are due to the fact that no aggregation just sends at uniformly distributed times and *TAG* staggers transmissions as spread out as possible due to its method of dividing the period into slots. In both of these cases the delay will increase as the period increases.

In summary, as expected, our results show that in-network data aggregation can achieve considerable energy savings. *Periodic per-hop adjusted* aggregation (specifically our *cascading timers* algorithm) is able to maintain the same *freshness* and *accuracy* as when no aggregation is used. This is an impressive result considering the constraints imposed by periodically generated data. Furthermore, while *TAG* exhibits reasonable performance for shorter collection periods, *cascading timers* performs consistently well across a wide range of collection intervals. We study the behavior of the different aggregation timing models in more detail in Section 3.4.4.

3.4.3 Estimating the Single Hop Delay

As discussed in Section 3.1, *periodic per-hop adjusted* aggregation algorithms schedule a node's timeout, i.e., the time a node is due to "clock out" the current data aggregate, as a function of the node's position in the distribution tree. More specifically, in our *cascading timers* aggregation, timeout is a function of the *single hop delay* or *shd*, an estimate of the time a packet is expected to take to traverse one hop in the data

collection network, including processing time. Given *shd*, we can then estimate the time it takes for a packet to traverse the path from the source to the sink by multiplying *shd* by the number of hops traversed.

In this section, we study how *shd* impacts the performance of *cascading timers*. Note that *shd* depends on current network conditions such as load, channel quality, etc. If the network is heavily loaded, packets might take longer to traverse one hop. This may result in nodes timing out and readings getting lost. The performance of *periodic simple* and *periodic per-hop* aggregation schemes will also be impacted when under heavy load.

Recall that, in order to avoid collisions, we stagger node transmission in relation to one another using a small random interval. Thus, *shd* has basically a deterministic- as well as a non-deterministic component. While propagation- and transmission delay make up *shd*'s deterministic component, the staggering interval and queuing delay are responsible for *shd*'s non-determinism. *shd* can then be computed using the expression in Equation 3.4,

$$shd \quad = \quad sd + td + pd + qpd \tag{3.4}$$

where, *sd* is the staggering delay of the packet, *td* and *pd* correspond to the transmission and propagation delays respectively and *qpd* accounts for both queuing and processing delays.

Cascading timers uses max_shd as given by Equation 3.5, which is obtained by using the maximum possible staggering delay and an upper bound (Up) of the rest of the components. The maximum *sd* is chosen by us. The propagation and transmission delays are a function of the network architecture and the maximum size of the packets. The processing and queuing in this scenario are almost negligible since as soon as packets arrive they are aggregated and the processing for this is minimal.

$$max\_shd \quad = \quad Max(sd) + Up(qpd + td + pd) \tag{3.5}$$

Figure 3.5 presents *cascading timers*' weighted accuracy for different values of *shd* ranging from 0.01 to 0.3. The weight used was 0.5. For good performance we chose the maximum staggering interval to be 0.05. This is a good enough compromise between

large enough *sd* to avoid collisions and small enough to allow the last readings to get to the sink within the collection interval. Our original experiments used an *sd* of 0.03. When studying the values of the *shd* and the *sd* later we found the value of 0.05 to be slightly more optimal for our scenario and density.

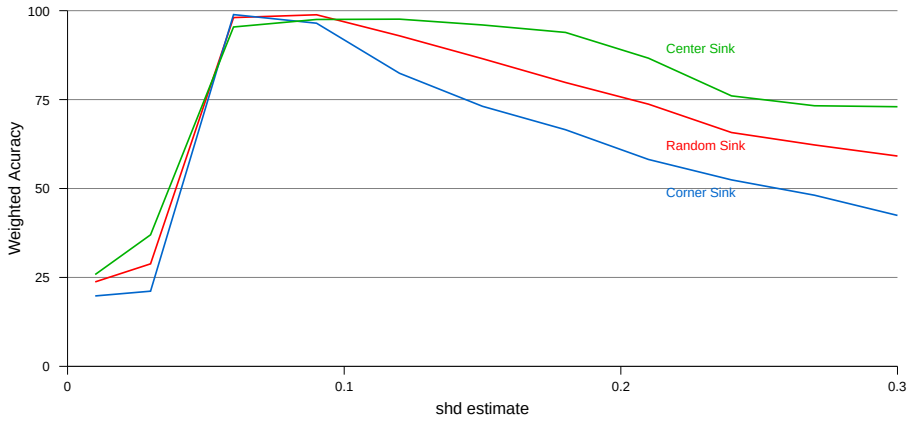


Figure 3.5: Weighted accuracy for different *shd* values

For *shd* smaller than the maximum random staggering interval (left-hand portion of the graph), we observe that the algorithm's performance in terms of accuracy deteriorates considerably. This is to be expected: data is aggregated and sent even before children nodes try to send since their stagger timer hasn't expired. When *shd* is slightly higher than the staggering interval, performance is maximized. This is because child nodes will have enough time to send in their data items to the corresponding parent.

For large values of *shd* the algorithm's weighted accuracy starts to drop. This is expected since the time it takes for readings from farther nodes to reach the sink using large *shd* ends up being longer than the collection period, and hence they are worth less according to weighted accuracy.

Setting the staggering interval is an important tradeoff. While large staggering intervals are effective in avoiding collisions, they yield higher delays. As previously

pointed out, the value of *sd* also depends on network density. In dense networks larger staggering intervals are needed to avoid collisions. Another important consideration is the type of medium access control (MAC) protocol employed. Contention-based MAC protocols, such as CSMA (which is what we use in our simulations), are prone to collisions and node transmissions need to be staggered in time. However, other types of MAC, such as scheduled access protocols, are collision free and thus do not require staggering of node transmissions, probably at the expense of delay.

We also measured the values of the real *shd* over the course of our simulations and observed very small variations. This is expected, since traffic flows over the same data collection tree and the offered load is essentially constant.

3.4.4 Data Collection Interval

This section discusses the impact of the data collection interval's length on the performance of periodic aggregation timing models. Intuitively, the smaller the collection period, the more critical the timing model used by in-network aggregation.

Periodic simple and *periodic per-hop* have no timer organization, and send data randomly during the data collection interval. This basically creates a data path where readings from nodes will take approximately as many collection intervals as hops to reach the sink. The longer the collection interval, the longer delays these data paths will incur.

In the case of *no-aggregation*, data is sampled at random times within the interval and then sent. Data reaches the sink as quickly as possible since it is forwarded with no waiting. Following our *cascading timers* model, data is transmitted near the end of the collection period depending on the position in the aggregation tree. This is independent of the period's length. All the data arrives with a small delay at the sink right before the period ends.

We define two different performance metrics associated with the delay a reading takes to reach the sink. The first definition focuses on how long the reading takes to reach the sink from the time it is sampled. We call this metric the *sample-to-sink* delay. The second metric looks at the time it takes from when a reading is sampled until **all** readings are received by the sink (*sample-to-all*). While the former metric targets

applications that are mostly interested in the latest readings, the latter is useful when the application performs some computation which requires all readings.

As previously discussed, according to the *sample-to-sink* metric, *periodic simple* and *periodic per-hop* aggregation will not perform well. Furthermore, in larger networks, the extra hops packets take to reach the sink will add to the delay. *No-aggregation* will incur the smallest delay possible, having packets hop their way to the sink. Assuming the staggering delay is uniformly distributed from 0 to *sd*, packets under *no-aggregation* will take on average $(\frac{sd}{2} + C) * hops$, where *hops* is the number of hops data has to traverse and C accounts for transmission, propagation, and queuing delays.

Cascading timers will exhibit a *sample-to-sink* performance of almost double that of *no-aggregation* since the *shd* used is based on the maximum, and not the average staggering delay (*sd*) like *no-aggregation*. This means that the delay will be $(sd + C) * hops$. Double the *sample-to-sink* delay might seem like a big disadvantage; however, given that the staggering interval is small compared to the data generation period, the *sample-to-sink* difference between *no aggregation* and *cascading timers* is relatively small.

The *sample-to-all*($StA()$) delay provides a metric for the overall freshness of the data the sink receives. In *no aggregation* data gets to the sink throughout the generation period. Thus, the average time a reading will have to sit idle at the sink is $\frac{period\_length}{2}$, that is, from when it arrives at the sink, until it is tallied at the period's end. *Cascading timers*, on the other hand, sets nodes to transmit at the end of the period, so the time the readings have to wait at the sink is always close to 0 (see Figure 3.1).

$$StA(NoAgg) = n((\frac{sd}{2} + C)D + \frac{pl}{2}) \tag{3.6}$$

$$StA(Cascading) = n((sd + C)D) \tag{3.7}$$

Equation 3.6 and 3.7 give us the average *sample-to-all* delay over all nodes (n) for *no-aggregation* and *cascading timers* respectively. D represents the average node depth and *pl* denotes the period length. As shown in Equation 3.8, for *cascading timers* to have a lower overall *sample-to-all* delay than *no-aggregation* the period length (*pl*)

has to be larger than the maximum staggering delay (sd) times the average node depth (D).

$$\begin{aligned} StA(NoAgg) &> StA(Cascading) \\ n((\frac{sd}{2}+C)D+\frac{pl}{2}) &> n((sd+C)D) \\ (\frac{sd}{2})D+\frac{pl}{2} &> (sd)D \\ pl &> (sd)D \end{aligned} \tag{3.8}$$

In the worst case scenario, i.e. a line topology, the average node depth will be $\frac{n}{2}$. For any large sample period, pl will be larger that $\frac{sd*n}{2}$. Using our simulation parameters, 100 nodes and a 0.03 sd, $StA(Cascading)$ would be smaller than $StA(NoAgg)$ for any period length greater than 1.5 seconds. Again we note that this is the worst case scenario. Using *no aggregation* under that scenario will probably incur collision problems due to the amount of traffic at the nodes close to the sink.

For the other topology extreme, i.e. a single hop tree with an average depth of 1, it is easy to see that *cascading timers* will have lower delays for all values of the period length (greater than the maximum staggering delay). The main drawback would be that, if there are many nodes, the traffic at the end of the period may cause too many collisions. This could be potentially solved by increasing the staggering delay. If the staggering delay is increased to match the period length then $StA(NoAgg) = StA(Cascading)$.

3.5 Packet Losses

As previously discussed, timing models are critical for the performance of in-network aggregation. Indeed, efficient aggregation may result in packets carrying several readings. Packet losses can, thus, considerably degrade the accuracy and timeliness of data aggregation. In this section, we study the effect of packet losses when collecting and aggregating periodic data. We also propose three different mechanisms to handle loss and evaluate their performance.

Since we target applications that generate data periodically, we avoid recovery

mechanisms that introduce delay. If data recovery and retransmission take too long, nodes would already be producing the readings for the following round, error recovery would interfere with the propagation of new data. Take for example a traditional error recovery scheme where negative acknowledgments (NACKs) are used. Under the proposed aggregation protocols, every node knows how many children it has. Therefore, if the node times out waiting to hear from some of its children, it could then send a NACK to request them to retransmit their data for the current round. However, this would interfere with the next round of data. Since target application scenarios don't require perfect data delivery, we are willing to sacrifice accuracy in favor of freshness.

We use a proactive approach to error recovery along the lines of error correcting schemes such as Forward Error Correction (FEC) [32]. Since the typical data items produced in the target application scenarios are generally small (a few bytes), we decided to use a very simple form of error correction mechanism, namely send a packet multiple times. In future work, we plan to investigate other, more sophisticated error correcting codes that can pay off in the case of more complex data.

3.5.1 Handling Losses

Our loss model is simple: drops are independent and the loss probability p is fixed for all links and is kept constant throughout the simulation. A packet is successfully transmitted with probability $(1 - p)$. While this is a simple scheme, it can be used to get a general picture. We will look at spatial and temporal correlated losses in future work.

As mentioned earlier, we deal with losses by pro-actively sending data multiple times. This strategy of course creates a tradeoff between consuming energy to send redundant packets and increasing the probability of delivery. Redundant transmission mechanism might not be suitable for all scenarios; in particular, when data is generated sporadically (rather than periodically) normal error recovery using acknowledgements and retransmissions will likely be more cost-effective.

We use three different redundant transmission schemes, namely *double-send*, *max-send*, and *adaptive-send*. While *double-send*, as its name implies, sends every packet twice, *max-send* sends every packet as many times as readings are aggregated in the

packet. This requires the protocol to include an "aggregation counter" in the data packet. The reasoning behind this strategy is that packets carrying more readings are more valuable and we therefore want to increase their chances of getting through. *Adaptive-send* is a more complex algorithm, whose goal is to achieve certain delivery guarantees (expressed by number of acceptable losses) for a given loss rate. Equation 3.9 describes the relationship between the number of acceptable losses l and the drop probability p, where a and t are the number of aggregated readings in a packet and the number of transmissions, respectively. The number of transmissions to achieve l under loss probability p is then given by Equation 3.10.

$$l = a \cdot p^t \tag{3.9}$$

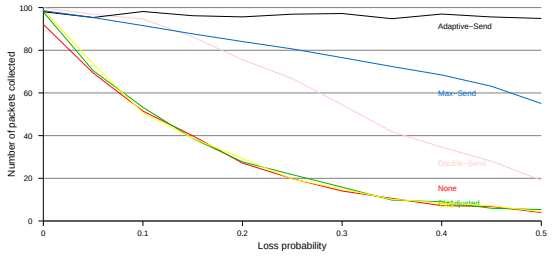
$$t = \log_p \left(\frac{l}{a} \right) \tag{3.10}$$

Note that, since we aim for hop-by-hop guarantees, these equations apply to a single link. In other words, if $l = 0.05$, *adaptive send* will perform the necessary number of retransmissions to guarantee a 95% delivery guarantee over a particular link given that link's loss rate. The overall network's delivery guarantee may be slightly different (higher or lower) depending on the interdependencies between the aggregates in the packets as they flow towards the sink. As our simulation results show, the overall reliability achieved by *adaptive-send* (shown in Figure 3.6 as data accuracy) is typically very close to $1 - l$.

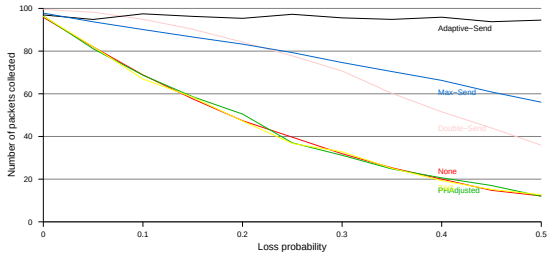
In a real network, the loss probability can be either known a priori (from previous experience operating the network) or estimated over time. Underestimating the loss rate yields delivery rates lower than required; conversely, loss rate overestimation results in more redundant packets.

3.5.2 Results

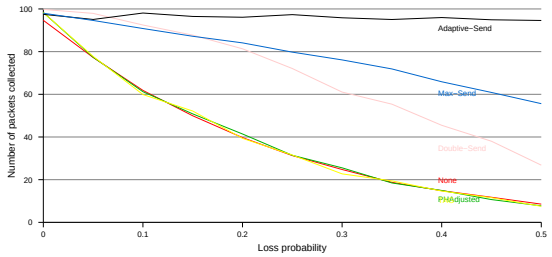
We repeated the simulation experiments described in Section 3.4 to study the performance of the three proposed loss handling strategies when employed by *cascading timers* aggregation. Loss probabilities range from 0.05 to 0.5 in steps of 0.05. We kept



Corner Sink



Center Sink

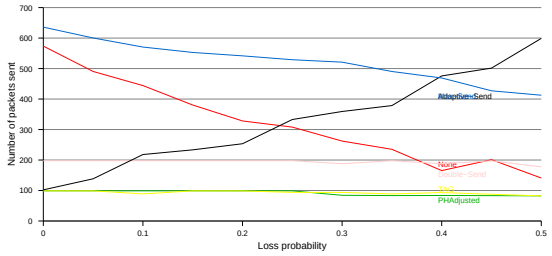


Random Sink

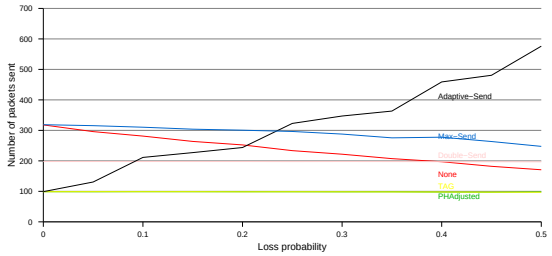
Figure 3.6: Data accuracy under loss

all other parameters the same. In the case of *adaptive-send*, the acceptable loss rate was set to 0.05 (which implies relatively high delivery guarantees). We present the results for different sink placement scenarios in Figures 3.6 and 3.7. For comparison purposes, we plot results for *cascading timers* aggregation and no aggregation when they employ no error recovery.

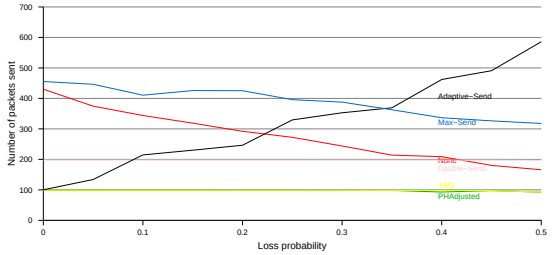
The results obtained for the different algorithms bring out some interesting points. Figure 3.6 shows data accuracy (percentage of readings received by the sink)



Corner Sink



Center Sink



Random Sink

Figure 3.7: Packets sent under loss

achieved by the different mechanisms under different loss conditions. These results confirm our expectations: *adaptive-send* is the best performer with close to perfect accuracy even under high loss. *Max-send* delivers more than 50% of the readings, while *double-send* starts with a very good delivery but quickly degrades under high loss conditions.

At low loss rates we notice that *double-send* performs slightly better than *max-send* and *adaptive-send*. This is attributed to the fact that it sends every packet twice, including packets with one reading, irrespective of loss rate. *Adaptive-send* and *max-send* send packets with one reading only once. When packets with one reading

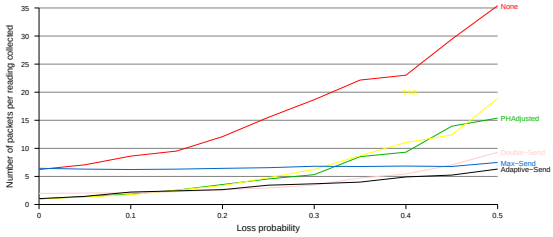
get dropped, they are not recovered. Packets that would have included the dropped reading will now be transmitted fewer times, decreasing their chance to get to their next hop, which decreases the next packet and so on. After a certain drop probability , *adaptive-send* starts sending packets with single readings twice. This seems to indicate that taking good care of packets, even when they only have one reading collected is important.

It is interesting to note that under the loss model and scenarios used, drops have similar impact on data collection with and without aggregation. We observe that the curves for *None* and *cascading timers* (without error recovery) look very similar. Data accuracy decays very quickly in both cases. We plan to study this phenomenon further using different topologies and network diameters.

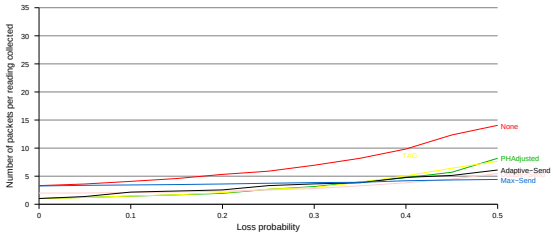
To evaluate the overhead incurred by these error recovery schemes, we plot total number of packets sent in Figure 3.7. *Double-send* sends approximately 200 packets per round for all loss rates as it always sends a packet twice. *Cascading timers* (without error recovery) also sends a constant number of packets (around 100 in this case) since it does not adjust its behavior for different loss conditions. At the higher error levels we notice that 100 packets are not sent per round. This is due to the fact that because of packet drops, at tree construction time some nodes don't become part of the tree.

It is somehow expected that *none* and *max-send* would exhibit similar behavior since *max-send* should transmit approximately the same number of packets as *none*. However, this is not the case: for *none*, the loss of packet means that nodes in the path to the sink will now transmit one less packet. In *max-send*, if all the transmissions of a packet are lost, subsequent packets will be transmitted fewer times since they will have less readings. The decline of *max-send* is slower than *none* because all of the transmissions of a packet have to be lost for subsequent nodes to transmit less. *Adaptive-send* increases the number of packets sent as loss probability increases since the number of transmissions is a function of the number of aggregated values and the drop rate.

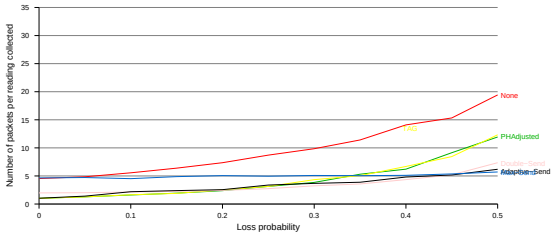
In some scenarios, rather than maximizing data accuracy, it may be more attractive to strike a balance between number of packets sent (or energy consumed)



Corner Sink



Center Sink



Random Sink

Figure 3.8: Packets sent per reading collected: randomly-placed sink

and accuracy. Figure 3.8 plots the number of packets sent per reading collected. The optimal case of course is one packet sent per reading. *Cascading timers* lets us achieve that goal when there are no losses. *Adaptive-send*, which is implemented atop *cascading timers*, yields good performance under all conditions due to its adaptive nature.

A corner sink means routes are longer, *none* and *max-send* due to this, specially at low drop rates. It is also the case that for the longer paths the single reading packets of *none* have a greater probability of getting dropped, hence the poor performance in packets per reading.

3.6 Related Work

Protocols for sensor networks have sparked considerable interest in the network research community. In this context, data aggregation rose as a technique for improving sensor network protocols' energy efficiency. We briefly describe some previous and ongoing research efforts in order to put our work in perspective.

Directed diffusion [5] has been proposed as a data gathering protocol for sensor networks. It targets the monitoring of events which are typically sensed only by a few nodes. An example scenario is tracking animal herds in a given geographic region. Diffusion's communication paradigm is based on information sinks broadcasting requests, or *interests*, for relevant data. Nodes producing relevant information respond and *data paths* are formed. Data is aggregated when a node is part of various data paths.

Diffusion's aggregation is based on a report gradient, which defines how many reports to send per time unit. Therefore, according to our classification, diffusion falls in the *periodic simple* category. Every node can potentially perform aggregation; however, nodes in the shortest path from information sources to the sinks do most of it. Diffusion adapts well to node failures by keeping state of the interest throughout the network. When a path fails the neighboring nodes remember alternate paths. Some of their more recent work addresses the impact of node density in this type of data collection scenario [4].

eScan [24] is an energy monitoring scheme that collects energy readings from every participating node. Their scenario is somewhat similar to the ones we target, i.e., every node maintains an energy value that is reported to a collection sink at the edge of the network. However, rather than generate periodic reports, new data is reported only when the energy of a node changes beyond a certain threshold. Aggregation is performed as data flows to the sink by merging reports of similar energy values into *energy range polygons*.

The initial eScan work does not handle latency in propagating data; they also assume a perfect MAC layer and instead of using time they use data generation events to drive their simulations. Rather than being an alternative to eScan, our aggregation techniques can be incorporated by eScan to, for example, improve data freshness.

SPIN [45], Sensor Protocols for Information via Negotiation, is a protocol for

data collection and dissemination. In SPIN, all nodes have pieces of named information that they want to send to the rest of the nodes. Data transfers are first negotiated based on the names of items. Only requested items are exchanged. This avoids the cost of sending the data needlessly but incurs the overhead of engaging in the negotiation phase. Note that SPIN's communication model is based on a gossip-style approach. The resulting protocol is very similar to NNTP [26] for propagation of news over the Internet. Essentially, it uses point-to-point communication among pairs of nodes to eventually convey data to all interested participants. SPIN does not really use an explicit aggregation mechanism; aggregation is performed implicitly during initial negotiation between two nodes using the meta-data to decide whether actual data will be exchanged.

TAG [39], or Tiny AGreggation, is a sensor network querying system. It employs a SQL-like syntax and uses aggregation as the query is processed within the network. When a query involves an *epoch*, requiring readings to be collected periodically, TAG uses the *periodic adjusted* aggregation approach. It subdivides the epoch into slots. The length of a slot is given by the epoch length divided by n, the maximum number of hops separating data generation nodes from the sink. Following *periodic adjusted* aggregation operation, slots are assigned to nodes in decreasing order, n, $n-1$, $n-2$, ... , as the query propagates through the network. Nodes transmit in their slot, hence, the out-most nodes will transmit first and nodes closest to the sink, last. As in any time-slotted mechanism, clock synchronization among nodes is required so that nodes transmit in their designated slots. TAG also takes advantage of time slotting to switch idle nodes' radios off.

Convergecasting [42] performs aggregation as it collects data periodically from all nodes to a single sink. Like TAG, its data aggregation mechanism also falls in the *periodic adjusted* category. It assigns aggregation slots as the query percolates the sensor network, trying to assign nodes to different slots in order to avoid collisions. Once the algorithm finishes assigning slots, that is, when the query setup reaches the edge of the network, the order of the slots is inverted to reflect a data collection tree. A similar concept is used by the work reported in [9].

3.7 Conclusions and Future Work

This chapter explored in-network aggregation as a power-efficient mechanism for collecting data in wireless sensor networks. Our focus was on applications where a large number of nodes produce data periodically which is consumed by fewer sink nodes. Such communication model is typical of monitoring scenarios, one key application of sensor networks.

Using simple analytical models, we present a trade-off analysis of in-network data aggregation. Through simulations, we evaluate the performance of different in-network aggregation algorithms, including our own *cascading timers*, and characterize the tradeoffs between energy efficiency, data accuracy and freshness. Our results show that timing, i.e., how long a node waits to receive data from its children (downstream nodes in respect to the information sink) before forwarding data onto the next hop (toward the sink) plays a crucial role in the performance of aggregation algorithms in the context of periodic data generation. By carefully selecting when to aggregate and forward data, we achieved considerable energy savings (as much as 5 times less traffic) while maintaining data freshness and accuracy.

Finally, in order to perform well under packet loss conditions, we developed three different proactive error recovery techniques which are suitable to periodic data generation (as they incur no additional delay). They are essentially based on proactively transmitting packets multiple times. Simulations showed that the proposed techniques were able to maintain high accuracy even under high loss conditions. Among the proposed mechanisms, *adaptive-send*, which adjusts the number of times a packet is sent based on the number of readings aggregated in the packet and an estimate of the loss probability, yielded the best performance.

As future work, we plan to use more sophisticated loss models to evaluate our recovery mechanisms under different loss conditions. We will also develop techniques to handle node failures. We also plan to investigate aggregation algorithms that target different scenarios. For example, instead of periodic data generation, explore scenarios in which data is reported only when it changes significantly. Another direction is to explore cluster-based aggregation, where clusters are formed based on a set of constraints and then data is aggregated within clusters.

Chapter 4

Isoclusters - Grouping by value

In this chapter, we describe a novel aggregation technique that targets spatially-correlated data. In particular, we address applications that are continuously monitoring varying conditions of a given geographic region (e.g., temperature, rain fall, radiation, etc.) and, as a result, generate a "contour map" of the sensed variable.

The proposed algorithm takes advantage of the spatial correlation of data in these monitoring scenarios, i.e. the fact that nearby nodes often sense similar readings. Energy efficiency is achieved by, instead of having all nodes send their readings to the sink, having only a few nodes report. Ideally, only nodes with important information will report. Our approach defines the important information to be the isolines of a map.

Isolines are basically *isopleths* (from the Greek *iso* - same and *pleth* - value), a line composed of points of the same value. When these lines are drawn on a map we get a contour map, like the one shown in Figure 4.1. Areas encompassed by isolines lie within a certain value range and we call them isoclusters.

Energy efficiency is achieved by, instead of having all nodes send their readings to the sink, having only a few nodes per isocluster report to the sink. For instance, if we were to generate a contour temperature map, an isotherm, the temperature ranges would be defined and then nodes would be grouped into areas that exhibit temperatures within the defined ranges. To construct an isotherm we do not need to collect data from all the nodes in the region being monitored; it is sufficient to find the isolines, draw them on our map and "color in" the corresponding areas. This is how energy efficiency is attained.

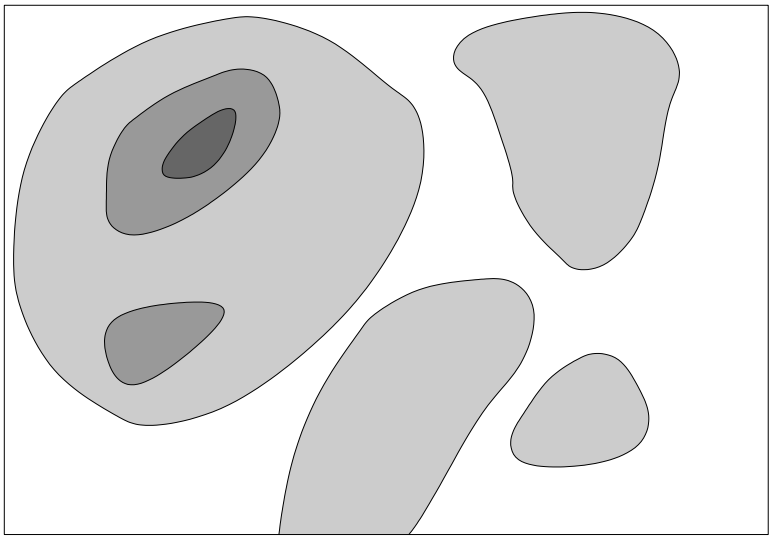


Figure 4.1: An isograph

Through simulations, we evaluate *isoline aggregation* and compare it against no aggregation and our implementation of aggregation using polygons. *Polygon* aggregation is the general approach used in previous approaches, for example e-Scan [24] and *isobars* [23], both of which are discussed in more detail in Section 4.1 below. Our results show that *isoline aggregation* can achieve significant energy savings when compared to no aggregation and *polygon* aggregation, while yielding high data accuracy.

The remainder of this chapter is structured as follows. We discuss related work in Section 4.1 and describe *isoline* aggregation in Section 4.2. Sections 4.3 and 4.4 present our experimental methodology and evaluation results, respectively. In Section 4.5, we present our concluding remarks as well as directions for future work.

4.1 Related Work

There are mainly two other approaches that target spatially-correlated data aggregation for mapping, namely *eScan* [24] and *isobars* [23]. *eScan* focuses on monitoring the sensor network itself, in particular the remaining energy in the nodes. It queries sensing nodes which, in turn, report their remaining energy. This is done via a data collection tree established at query propagation time. When the data is being reported

back to the sink, nodes aggregate the information as it flows. Aggregation is done by grouping readings that meet a certain criteria. In particular, for a set of readings to be aggregated, they need to be geographically adjacent and they need to be in the same value range.

Data is aggregated into polygons of similar value and represented by the corresponding polygon's coordinates. This approach has a few drawbacks. For one, the aggregation is done as the data flows down the collection tree, which is not always the most efficient way. For instance, if two nodes close-by are in the same value range but are in different branches of the tree, their values are not aggregated until they reach a common ancestor, who may be further up the tree. Also, for a node to aggregate the coordinates of a polygon it needs to know the exact location of the nodes. Assuming geographic location encoding requires more bytes than node identifiers, propagating location information results in significant additional overhead.

In contrast, *isoline aggregation* employs localized aggregation by detecting *isolines* with neighboring nodes. Hence, aggregating is done "on the spot" rather than down the collection tree. Location information is only needed at the sink and can be collected once.

The *isobar* mapping approach is part of the the advanced aggregation techniques proposed in TAG [23]. Here nodes are part of a grid. A node's location is based on its position on the grid. Data is collected by aggregating nodes with similar readings into polygons. On a heavily populated grid, aggregation yields good results. If the grid is sparse, or if packets are dropped, or if energy efficiency is favored over accuracy, *bounding boxes* are used for defining the polygons. A bounding box is created around an area to be aggregated. Cuts are then made to the bounding box to approximate the shape of the polygon. The more cuts, the more data that needs to be reported and the better the accuracy. Less cuts means decreased accuracy, but less data to be sent, improving energy efficiency.

Isobar mapping suffers from similar problems as *eScan* since it also performs aggregation as the data flows towards the sink. Node location is represented by the corresponding grid coordinates minimizing the need to transmit real location information.

More recently, the energy-accuracy tradeoff study by Boulis et al. [2] proposes

a data collection mechanism where nodes decide whether to share their own readings based on estimates they get from other nodes. While this works well for operations like reporting the maximum or minimum value, it does not apply to more complex applications like mapping.

Approaching the subject from a mathematical angle, Doherty et al. [27] studied how different mechanisms to collect scattered data perform in dense sensor networks. The focus of their work is on node selection rather than protocol development.

Both *eScan* and *isobars* use polygons to aggregate data of similar value produced by neighboring nodes. We will compare the performance of *isoline aggregation* against (1) no aggregation and (2) *polygon* aggregation, our implementation of the aggregation mechanism underlying both *eScan* and *isobars*. Even though continuous monitoring is not specifically addressed by *eScan* or *isobars*, in our implementation of *polygon* aggregation we employ temporal aggregation so we can conduct a fair performance comparison against *isolines*.

4.2 Isolines

The goal of *isoline aggregation* is to provide energy-efficient data collection by reducing redundant transmissions. One challenge is to achieve this goal using only local information. Another challenge is to maintain high data accuracy. To address these challenges, *isoline aggregation* uses the concept of *isolines*, lines of the same value. Energy efficiency is achieved by having each node only report to the sink if it detects an isoline between itself and its neighbors; otherwise, no report is generated.

4.2.1 Neighbor-to-Neighbor Protocol

Isolines are detected based on neighborhood information gathered through a neighbor-to-neighbor protocol, or NNP. Essentially, NNP broadcasts the local sensed value. Nodes decide when they need to communicate their sensed information to their neighbors. This happens when (1) a node is started and (2) when data changes cause an isoline to appear or disappear. First-time reports allow the network to detect initial values and the presence of new nodes.

An NNP packet is a very simple broadcast packet containing a packet type, a query id, node id and value. The packet type is a common field between all packets and is just used to determine the packet is of type NNP. The query id is used to differentiate between multiple queries (i.e. one for temperature, another for soil acidity, another from a different sink, etc). Node id is just the id of the node sending the NNP. Value just contains the value sensed by that node. The NNP exchanges are basically overhead, but are easily outweighed by the savings achieved.

4.2.2 Isoline Detection and Reporting

Isoline detection is a very simple yet elegant method of collecting information efficiently for contour map generation. It works as follows. First, a node compares its reading with the reading of neighboring nodes (received through NNP reports). If the readings lie in different sides of an isoline, then a report needs to be generated. For example, if the isolines measure multiples of 10, then a node sensing 35 and a neighbor whose sensed value is 42 are able to detect that there is (at least) one isoline of value 40 passing between them.

Once the existence of an isoline has been determined, it needs to be reported to the data collection sink. Reporting an isoline consists of sending to the sink the node's sensed value and the value of the neighboring node across the isoline. Sending just the isoline value saves some bytes at the expense of accuracy.

The detection of the isoline is symmetric, i.e., both the node and its neighbor will detect it. The node who actually reports the isoline is the one closest to the sink (according to hop count). If both nodes are at the same distance, the node with the lowest reading will send the message.

Nodes will only report to the sink when there are new isolines nearby. This could lead to problems since we assume that, on the absence of reports, nothing has changed. For this situation we also implement probabilistic reporting. Nodes broadcast their information periodically even when they do not need to do so. This also helps improving the accuracy of the maps generated and can be fine-tuned appropriately.

An isoline report message contains a packet type, the node id of the sending node and it's value as well as the node id and value of the neighbor. If a node detects

isolines with multiple neighbors it would be possible to include an array of neighbor-value pairs; our current implementation does not do that.

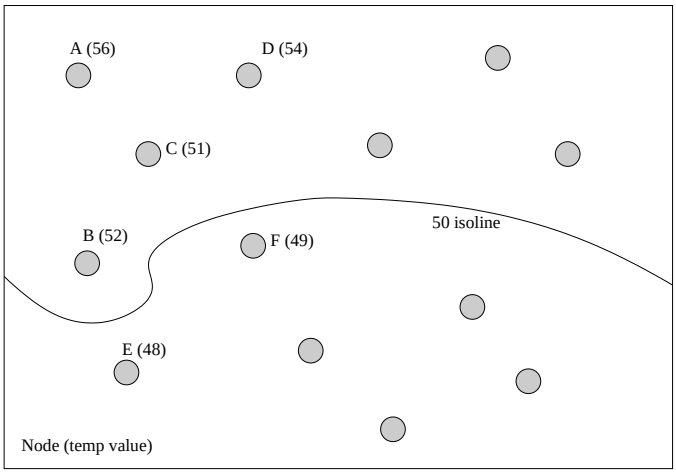


Figure 4.2: A temperature isoline

Nodes that are inside the same isoline are said to belong to the same *isocluster*. Figure 4.2 depicts a temperature isoline where temperatures within 10-degree ranges belong to the same isocluster. Nodes A and C are part of the same isocluster and will know, upon exchanging sensed values, that an isoline does not exist between them. On the other hand, nodes B whose value is 52 and E who is measuring a temperature of 48 will detect the 50-degree isoline when they compare values.

There is a clear trade-off between the range of the isolines, which determines the accuracy of the aggregation algorithm, and energy efficiency. The smaller range an isoline covers, the more accurate the overall map is. However, denser contour maps (in terms of number of isolines per area), generate more data and thus are less energy efficient.

4.2.3 Continuous Monitoring

In applications that are continuously monitoring a specific condition, the sensor network needs to continuously sample and report the current state to the sink. To accomplish this task in an energy-efficient manner, *isoline* aggregation takes advantage of temporal data correlation and reports only when changes are significant (ie. isolines

```
while(1){
  get_reading_from_sensor()
  if(reading_has_changed_range)
    broadcast_reading_to_neighbors()
  while_monitor_events
    switch(event){
      case receive_reading_from_neighbor:
        if(reading_ranges_differ())
          //determine who sends the readings
          //to the sink, me or the neighbor
          if(i_am_closer_to_sink ||
             (we_are_at_same_distance && my_reading_is_lower))
            add_readings_to_report()
          //else neighbor will send readings
        //else we are in the same range so we don't report
        break;
      case period_over:
        exit_event_monitoring()
    }
  if(report_contains_readings){
    send_report_to_sink()
    reset_report()
  }
}
```

Figure 4.3: Pseudo-code for continuous monitoring main loop

move, appear or disappear).

This capability uses the normal isoline detection mechanism. More specifically, a node starts by sampling its sensor and using the NNP to communicate its reading to its neighbors. Based on the information it gets from its neighbors, a node will report to the sink if an isoline is detected. Nodes will then sample the sensor periodically, but will broadcast their reading only when they change from one range to another. For example, if sensors are monitoring ranges of 10s (0-9, 10-19, 20-29, ...) and a node's reading goes from 17 to 23, the node sends out a NNP broadcast. This basically means that an isoline moved, appeared or disappeared, and the node needs to check if it needs to report.

Pseudo-code for the continuous monitoring algorithm's main loop can be seen in Figure 4.3. NNP is implemented by `broadcast_reading_to_neighbors()`. This NNP message contains a node's reading, the node id and sometimes its distance (hop count) to the sink. For clarity, the pseudo-code leaves out some error detection features.

Isoline aggregation uses cascading *timers* [36] to schedule node transmissions as a function of the node's position in the data collection tree. The outcome is that a cascading effect is achieved from the leaf nodes to the data sink; in other words, the farthest away node schedules its transmission first; the next hop (toward the sink), schedules its transmission next allowing enough time to receive data from its children, etc. Not only does this reduce delay, but it fits very well with schemes (e.g. at the MAC layer) that save energy by turning off the radio when the node is idle.

4.2.4 Dense Deployments

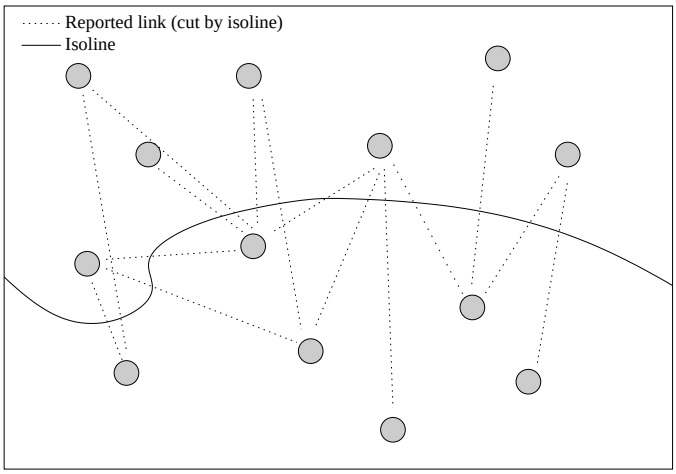


Figure 4.4: Dense isoline reports

Care must be taken when sensor nodes are deployed in a dense manner. Isolines need to account for such situations so that they avoid excessive redundant reporting. Figure 4.4 shows an example of a dense scenario. Nodes detect isolines by comparing their neighbors' reported values to their own. This is equivalent to checking whether an isoline "cuts through" a link between neighbors. Figure 4.4 also depicts all links that are cut by the isoline. In dense deployments, the number of links cut is usually large,

which leads to high overhead.

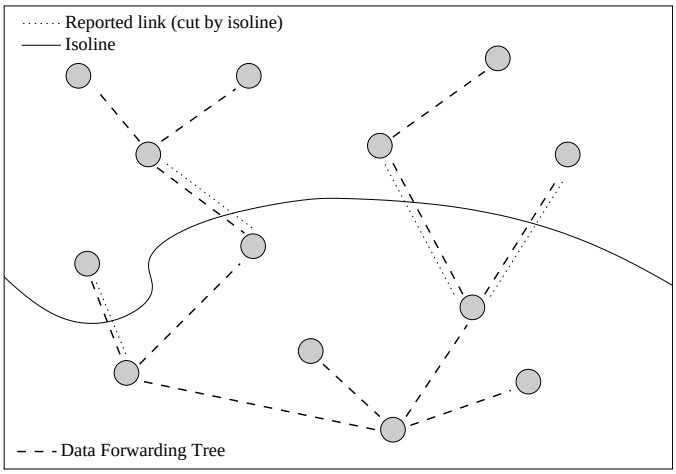


Figure 4.5: Optimized dense isoline reports

In order to minimize overhead incurred in dense deployments, isoline aggregation can be optimized by reporting only a subset of the links that are cut. Figure 4.5 shows this optimization. The dotted lines represent the links to be reported, the dashed lines represent the links that belong to the data collection tree. Basically a link being cut by an isoline will only be reported if that link is part of the data collection tree. In other words, nodes will only report isolines that they detect between themselves and their children or parent according to the collection tree.

We have evaluated this optimization in section 4.4.3, where we report performance results comparing *no aggregation*, and *isolines* with and without this density optimization.

4.3 Experimental Methodology

We use simulations to evaluate the performance of *isoline aggregation* and compare it against no aggregation and polygon aggregation. The remainder of this section presents our experimental methodology in detail.

4.3.1 Other Aggregation Algorithms

Polygon aggregation is used to represent the existing algorithms of *eScan* and *isobars*. Our implementation of is as follows: when a node receives data from its children, it aggregates it into polygons and sends only the polygons' vertices. All nodes in the polygon are assigned the average value of the polygon's range; for example, in the case of a [40-50) temperature range, nodes take the value of 45. Similarly to *isobar* aggregation [23], we use the sensors' grid coordinates as their locations. Reports from nodes may contain multiple polygons if multiple value ranges have been aggregated. *Polygon aggregation* uses the PolyBoolean library [28] for handling polygons and aggregating them. For both *polygon-* and *isoline aggregation*, data is grouped in ranges of 10, that is, from [0-10),[10-20), etc.

Similarly to isolines, we use *cascading timers* [36] as the timing model since it allows nodes to wait for their children to report without increasing the data collection delay. This is important because we are doing continuous monitoring and need to deliver information in a timely fashion.

The original *eScan* and *isobars* algorithms did not provide continuous monitoring. For fairness reasons, we incorporate temporal aggregation when extending *polygon aggregation* to perform continuous monitoring. For instance, if the temperature of a node has not changed ranges from the last time this node reported, a report will not be generated. This implies that leaf nodes will not report if there has been no range change since the last report. Inner tree nodes will have to report if leaf nodes report since they are part of the aggregation tree. Polygon aggregation needs them to include their information to perform aggregation.

No aggregation is our baseline algorithm. Nodes simply send their readings down the aggregation tree to the sink. This is very simple and effective. Basically the sink will get a reading from every node and will be able to generate the most accurate map using all possible sensed values. However, there are various problems with this approach. Having all nodes report means that considerable traffic will be flowing through the network.

No Aggregation Optimized builds on *No aggregation* by using temporal data correlation to reduce the number of reports generated. In this optimized version, nodes

report their readings directly to the sink only when their temperature has changed from one range to another.

4.3.2 Simulation Setup

As the experimental platform, we employ the `ns-2` [44] network simulator. For medium access control, nodes use CSMA at 115Kbps. Their transmission range is set to 40m. FLIP [35] was used as the network protocol. Node identifiers and location information are 2 bytes long. Temperature information is also 2 bytes.

We chose temperature as the condition being monitored. The algorithms and protocols can measure any other sensed variable. The temperature reality is simulated by a matrix of $80x40$ points. This will represent the area we are monitoring. When a node located at the center of our area samples its sensor it will read the center value of the matrix. These values are determined by the scenario and time in the simulation. In the case of the static scenarios the matrix will have one single value through out the simulation. In the case of the dynamic scenarios the values of this reality matrix will change.

4.3.3 Performance Metrics

The performance of the aggregation algorithms is evaluated according to two metrics: their energy efficiency and their accuracy. Of course, our goal is to minimize energy consumption without sacrificing data accuracy. We use average number of bytes transmitted by each aggregation protocol to measure energy consumption. This includes all data transmitted by all nodes, intermediate nodes included. While this might seem to not take receive power into consideration we should note that the timing scheme used, cascading timeouts, is well suited to MAC protocols that switch idle nodes off to low-power radio mode since communication is not taking place. This is true for both polygon aggregation and isoline aggregation. No aggregation doesn't do scheduling of transmissions and cannot use such a scheme without additional modifications.

Accuracy is measured by how **similar** the resulting map is to reality. Since our goal is to draw contour maps, it seems appropriate to calculate similarity as the percentage of points that are actually in the correct value range when compared to

reality. That is, for every point, we compare the reality value to the value interpolated from the data collected in a specific round. We calculate the percentage of points in the correct value range. In the case of continuous monitoring we average it over all simulation rounds (seconds). Simulations were run 10 times. This metric is called *contour similarity*.

4.4 Results

This section reports simulation results obtained from having the sensor field report a "snapshot" of the sensed data (Section 4.4.1 as well as experiments where continuous monitoring (Section 4.4.2) is employed. It also presents results showing the performance of the optimization techniques discussed in Section 4.2.4 (Section 4.4.3).

4.4.1 Taking Snapshots

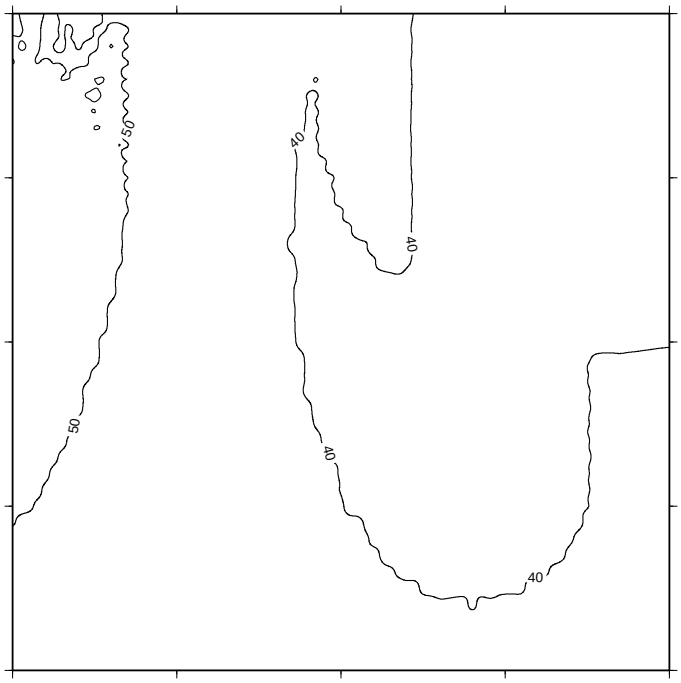


Figure 4.6: Real map

In these experiments, we use two different sensor network deployments, namely:

(1) a 400X400m field monitored by a 16X16 sensor grid evenly spaced at 25m intervals, and (2) a 800X800 field using 32X32 grid.

The sink node, which is placed at the center of the map, starts by broadcasting a query for the map at time 1s. From time 3s to 4s, nodes report their temperature readings. The simulation is stopped at time 5s. The data points used to compute the tabulated results are obtained by averaging over 10 runs.

Even though Figure 4.6 shows the real map, we will use Figure 4.7 as our idealized map to get performance bounds. The map in Figure 4.7 is generated when no aggregation is used, i.e., all the nodes are reporting their readings. This means that the sink has all the information the network can provide; the only way to be more accurate is by deploying more nodes increasing sensor density.

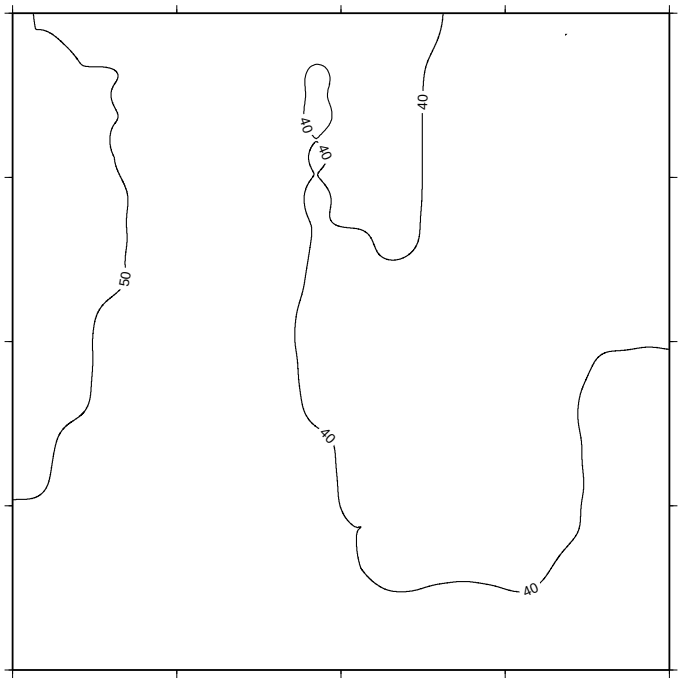


Figure 4.7: A map using all sensor readings

Our main goal is to achieve accuracy similar to the map obtained when no aggregation is employed, while making the collection process energy-efficient. Figures 4.8 and 4.9 present the maps generated using *isoline* aggregation, while Figures 4.10 and 4.11 were obtained using *polygon* aggregation. Figures 4.9 and 4.11 present the same

maps as Figures 4.8 and 4.10, respectively, superimposed atop points representing the readings actually received at the sink.

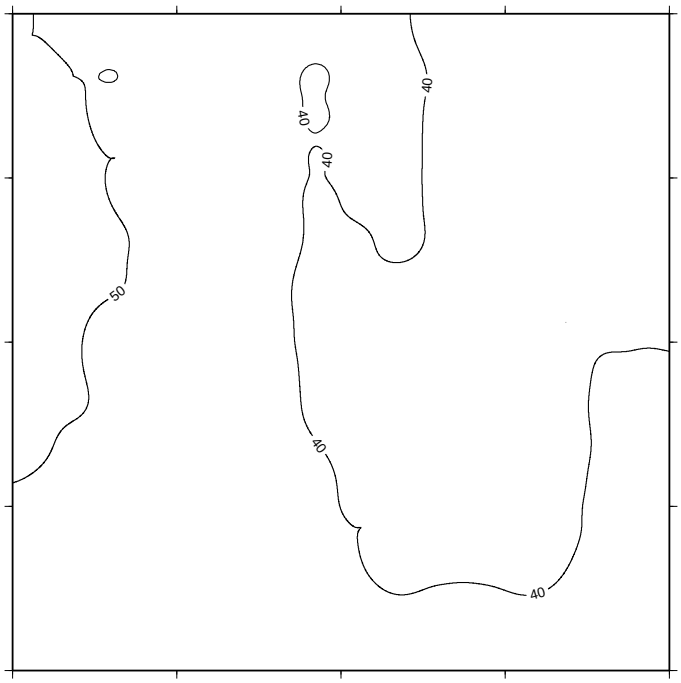


Figure 4.8: Map generated with *isoline* aggregation

Graphing the results obtained from the simulations helps us visualize how the aggregation algorithms perform in terms of accuracy. We should point out that the graphing tools we use, which interpolate the data points to generate the map, have not been optimized for plotting data points provided by aggregation mechanisms that exploit spatial data correlation. For example, in the case of *isolines*, if there are no readings received from an area, then it is an indication that an isoline does not exist. In the case of *polygon* aggregation, reported areas should be graphed as polygons. Moreover, some of the anomalies in the graphs are caused by lost packets. Even though the simulator is not inserting random drops, packet losses can still occur, for example, in the case of collisions.

In order to quantify how similar the maps generated by the different aggregation approaches are, we compute the average distance between corresponding points obtained from no aggregation, *isoline*, and *polygon* aggregation. We used the `ngmath`

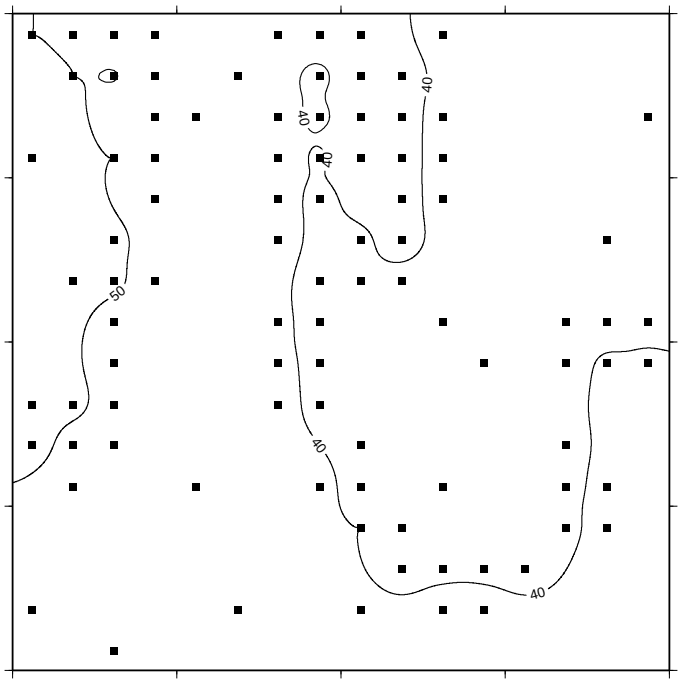


Figure 4.9: Map generated with *isoline* aggregation plus reporting sensors

	Reality (degrees)	No agg. (%)
No Agg.	1.21 (sd=0.01)	0
Isolines	1.59 (sd=0.12)	31.4%
Polygons	2.98 (sd=0.18)	146.3%

Table 4.1: Contour similarity for the 16X16 sensor field

part of the NCAR Graphics Library[31] to interpolate the 80X40 reality data points. Tables 4.1 and 4.2 summarize these results.

The first column shows the average difference and corresponding standard deviation between each aggregation algorithm and reality. For no aggregation, the map obtained when all nodes report differs on average by 1.21 (with a standard deviation of 0.01) in the 16X16 sensor deployment. Similarly, the average difference between *isolines* versus reality and *polygons* versus reality is 1.59 and 2.98 (in degrees as we are mapping temperature), respectively.

Recall that the upper bound in accuracy is obtained when all nodes report their readings and no aggregation is performed. Column 2 thus shows how the accuracy

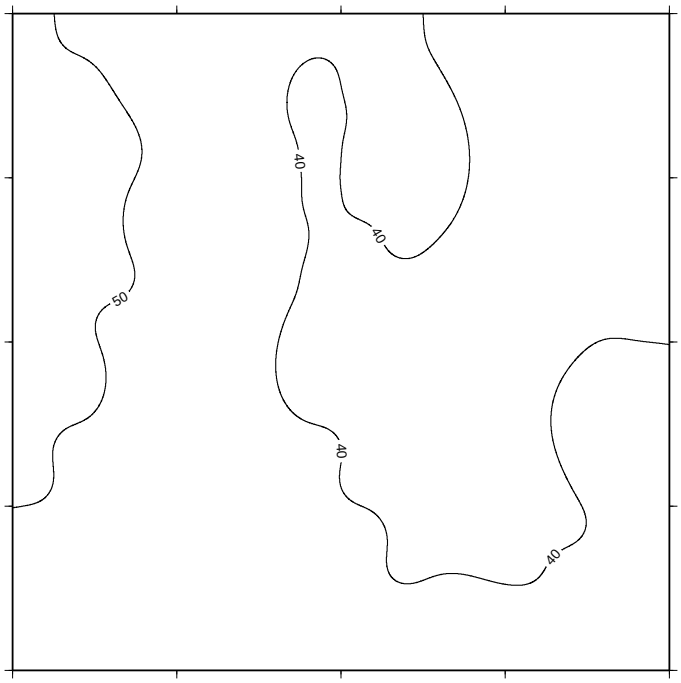


Figure 4.10: Map generated with *polygon* aggregation

	Reality (degrees)	No agg. (%)
No Agg.	1.24 (sd=0.08)	0
Isolines	1.83 (sd=0.24)	47.6%
Polygons	3.48 (sd=0.36)	180.6%

Table 4.2: Contour similarity for the 32X32 sensor field

of *isolines* and *polygons* compare to to that of using *no aggregation*. In the 16X16 map, isoline aggregation yields a difference of only 31%, while the map generated by *polygon* aggregation differs by 146%, about 5 times more than *isolines*. For the 32X32 map, with four times the sensors, we observe 48% and 181% difference, respectively.

On a first analysis, these differences may seem too high. However, when translating them into actual sensed data, they fall into perspective. For example, a point that has the value of 43 degrees might get mapped to 44.6 with *isolines* and 46 with *polygon* aggregation. Note that both of these algorithms are not trying to map reality on a point-to-point basis. Instead, they try to aggregate data by doing this 'lossy compression' into groups of values. To quantify this, we use the *group similarity* metric

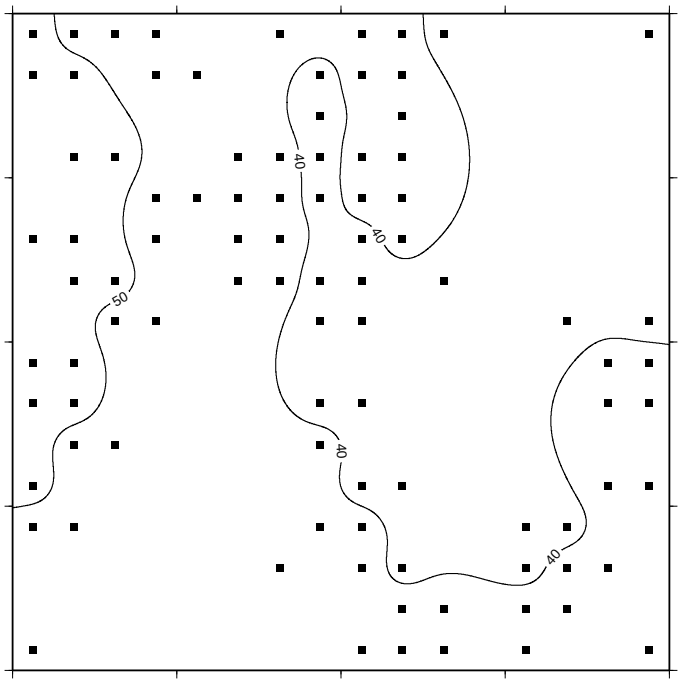


Figure 4.11: Map generated with *polygon* aggregation plus reporting sensors

Reality	16X16	32X32
No Agg.	95.0% (sd=0.1)	93.3% (sd=1.1)
Isolines	93.3% (sd=0.6)	91.8% (sd=1.7)
Polygons	92.5% (sd=3.9)	85.8% (sd=4.9)

Table 4.3: Group similarity

which calculates how many of the points are mapped into the correct value group. It basically measures if the contours look the same. Table 4.3 presents these results for both 16X16 and 32X32 sensor fields.

We observe that both algorithms perform reasonably well according to this metric, with *isolines* exhibiting better performance than *polygons*, especially in the 32X32 sensor field scenario. For example, Figures 4.6 (reality) and 4.7 (no aggregation) are 95% similar. These results also show that larger fields (with the same sensor density) are harder to map. This is particularly true for *polygon* aggregation, whose performance degrades as the network size grows.

Recall that the main goal of data aggregation is to achieve energy efficiency

Reality	Small	Large
No agg	13826 (sd=340)	80635 (sd=1221)
Isolines	7897 (sd=384)	32549 (sd=1095)
Polygons	8311 (sd=263)	34200 (sd=918)

Table 4.4: Bytes sent as energy efficiency

by transmitting less information. Table 4.4 shows the number of bytes sent by all three approaches. We observe that no aggregation transmits 75%- and 148% more data than *isolines* in the 16X16 and 32X32 sensor field scenarios, respectively. With no aggregation, every node needs to transmit its information to the sink which may results in redundant data traveling multiple hops, wasting precious resources along the way. Both *isoline* and *polygon* aggregation try to reduce the number of transmissions, minimizing data redundancy by aggregating spatially correlated information.

We should also point out that *isoline* aggregation uses temporal aggregation as well. That is, over time, nodes broadcast information (NNP) only when local readings change (i.e., they only need local knowledge). Section 4.4.2 evaluates the aggregation protocols under a continuous monitoring scenario where temporal aggregation is used.

Another observation is that *no aggregation* does not scale. As the number of nodes increases, forwarding packets from every node to the sink becomes prohibitively expensive, especially in power-constrained environments. This is especially true when larger areas report very similar values. Excessive redundant reports also result in increased collisions and thus data loss.

To increase accuracy isolines can cover smaller ranges (e.g. have isolines every 5 degrees). We ran some preliminary experiments varying the spacing between isolines. Using isolines every 5 degrees we send almost the same amount of data as no aggregation (13KB) and get a difference of about 1.38 degrees, that is about 13.2% more than *no aggregation*. This is a large decrease over the 31.4% we had before when using a 10 degree spacing between isolines. If we increase the spacing to 20 degrees our average difference increases to 97.5% (still lower than *polygon aggregation*) and we reduce the data sent to about 5.6KB (down from 7.9KB), less than half that of *no aggregation*.

The main disadvantage of using *polygon*-based aggregation lies in the fact that a node cannot make a localized decision of whether or not its information is redundant.

Aggregation is limited to nodes on the same branch of a tree and only happens when a shared ancestor exists. If this ancestor node is far away, redundant information will be propagated closer to the sink.

4.4.2 Continuous Mapping

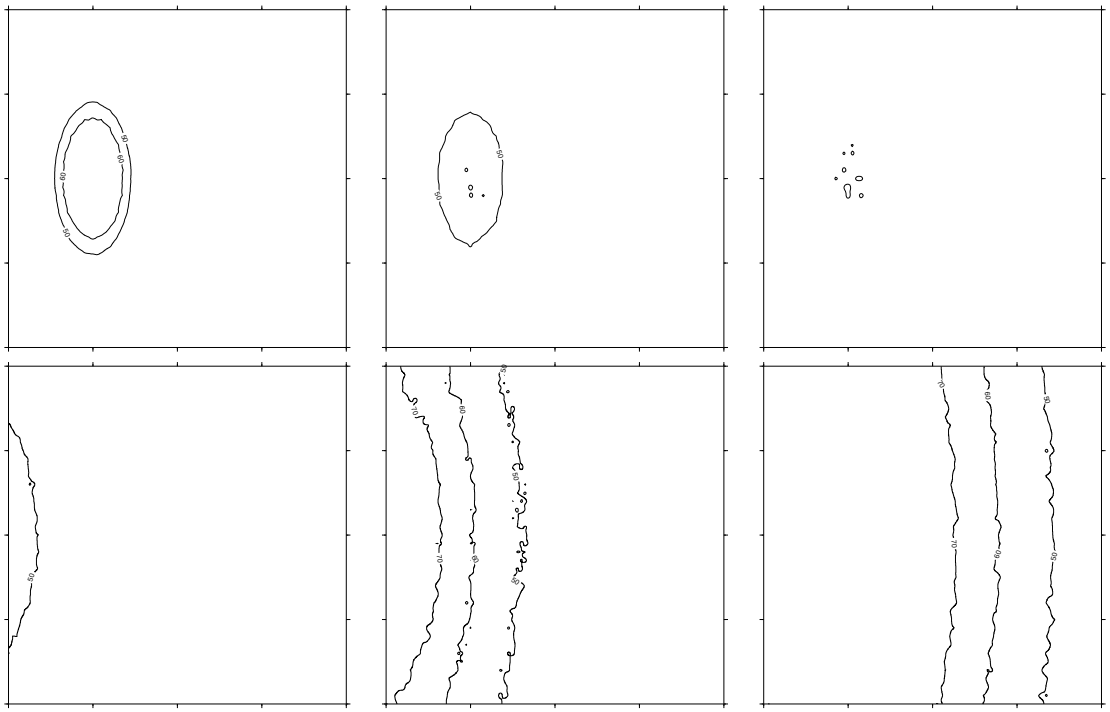


Figure 4.12: Hotspot and front scenarios

We use a sensor network consisting of $16x16$ nodes arranged in a evenly spaced grid monitoring temperature in a $400m^2 area$. Nodes sample the environment in rounds every 1 second.

In our continuous monitoring experiments, we simulate two types of phenomena: hotspots and moving fronts. In the first case, we generate a hotspot in our map with a temperature increase of 25 degrees. Temperature increase decays at power of 4 from the center of the hotspot. The region will then slowly go back to the original

temperature, as the hotspot vanishes. In our second scenario, we have a front moving from the left to right. Temperature increases from the forties to the seventies in about 150 meters. The front moves to the right in 13 seconds.

The experiment collects local temperature information for a period of 15 seconds. Figure 4.12 shows the real maps for the two scenarios at different points in time.

The starting value of all points is centered at 45 degrees with a small randomization factor of +/- 2 degrees. The sensor network will be basically subsampling the 80X40 matrix with a maximum of 16X16 samples if all nodes are reporting.

The sink node, which is placed at the center of the map, starts by broadcasting a query requesting the map at time 1s. From time 3s to 14s, nodes report their temperature readings. The simulation is stopped at time 15s. The data points used to compute results in Tables 4.5 and 4.6 are obtained by averaging over 10 runs.

Similarly to the results in Section 4.4.1, we also employ the `ngmath` part of the NCAR Graphics Library [31] to interpolate missing points (as received by the sink) of the $80x40$ reality map. As previously pointed out, the assumptions made by the interpolation algorithm may not be adequate to the data collection scenarios under consideration. For example, if we are capturing a gradient, like the one in the moving front scenario, some of the collection algorithms will only send the points corresponding to the nodes that changed value. If we only provide these points to the interpolation algorithm, it will assume that the gradient continues on both sides of the front. This will create very disparate graphs. The solution to this problem is to use the last data reported by a node as input to the interpolation. This problem was very pronounced in the "optimized" version of no aggregation as explained in the sequel.

In the case of *isoline aggregation* we have, by definition, nodes reporting only when they detect an isoline. Hence, we can infer that if no node reported, the values of that area have not changed over to the "next" isoline. So, if the highest report is 46, for example, we know that if the interpolation algorithm generates a value greater than 50, that value should not be considered.

Tables 4.5 and 4.6 present results for the hotspot and moving front scenarios, respectively. The first column shows contour similarity and corresponding standard

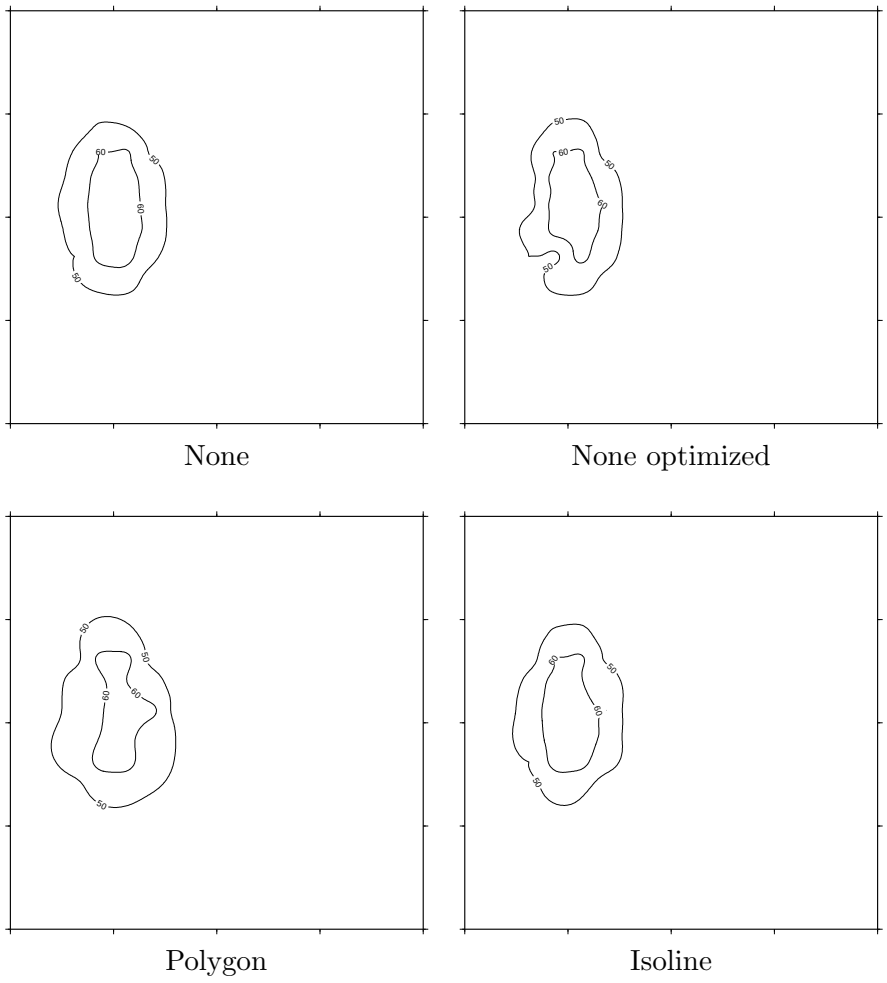


Figure 4.13: Hotspot scenario, T=4

deviation (in parenthesis). The second column shows the average number of kilobytes sent per round. This measures potential energy savings.

For the hotspot scenario we observe that all algorithms exhibit very good accuracy. This is to be expected as there are not many nodes changing values. The greatest differences can be seen in the amount of data sent. The unoptimized version of no aggregation sent a total of 180KB. That is more than 11 times the amount of data sent by *isoline aggregation*.

No aggregation optimized sends a total of 21KB. That is 38% more than *isoline aggregation*. Yet, isolines yields similar accuracy. This difference is mostly due to the

	Similarity	KBytes sent
No Agg.	98.7 (sd 0.09)	180.0 (sd 5.4)
No Agg opt.	98.9 (sd 0.09)	21.1 (sd 0.4)
Polygons	98.1 (sd 0.49)	62.9 (sd 4.6)
Isolines	97.0 (sd 0.36)	15.3 (sd 1.2)

Table 4.5: Hotspot scenario: contour similarity and number of bytes sent

	Similarity	KBytes sent
No Agg.	93.2 (sd 1.72)	177.1 (sd 5.9)
No Agg opt.	89.3 (sd 0.70)	62.1 (sd 2.3)
Polygons	82.4 (sd 2.93)	77.0 (sd 3.6)
Isolines	96.7 (sd 0.50)	55.8 (sd 3.1)

Table 4.6: Front scenario: contour similarity and number of bytes sent

fact that the initial collection without using aggregation is very expensive. When data changes in groups, *isoline aggregation* exhibits lower overhead since nodes with neighbors in the same range will not need to report. Therefore, if larger areas change values (e.g., if the hot spot affects larger regions), we expect that the energy savings achieved by *isoline aggregation* will be even more pronounced.

In the case of *polygon aggregation* more data is sent because aggregation happens only down the collection tree, and this implies that nodes which would otherwise not have reported will report. It also means that aggregation does not take place as soon as possible.

Figure 4.13 shows the maps generated from the data collected at time $T = 4$ for the hotspot scenario. This can be compared to the real map in Figure 4.12. Note that the maps in Figure 4.13 are a snapshot at time $T = 4$ of a single run; therefore, nuances and quirks are to be expected. We can see that the maps from *Isoline* and *None* (i.e., no aggregation without optimization) are the ones that most resemble the reality map.

Results for the moving front scenario are presented in Table 4.6. In this scenario, all nodes will eventually change value. The amount of data that needs to be

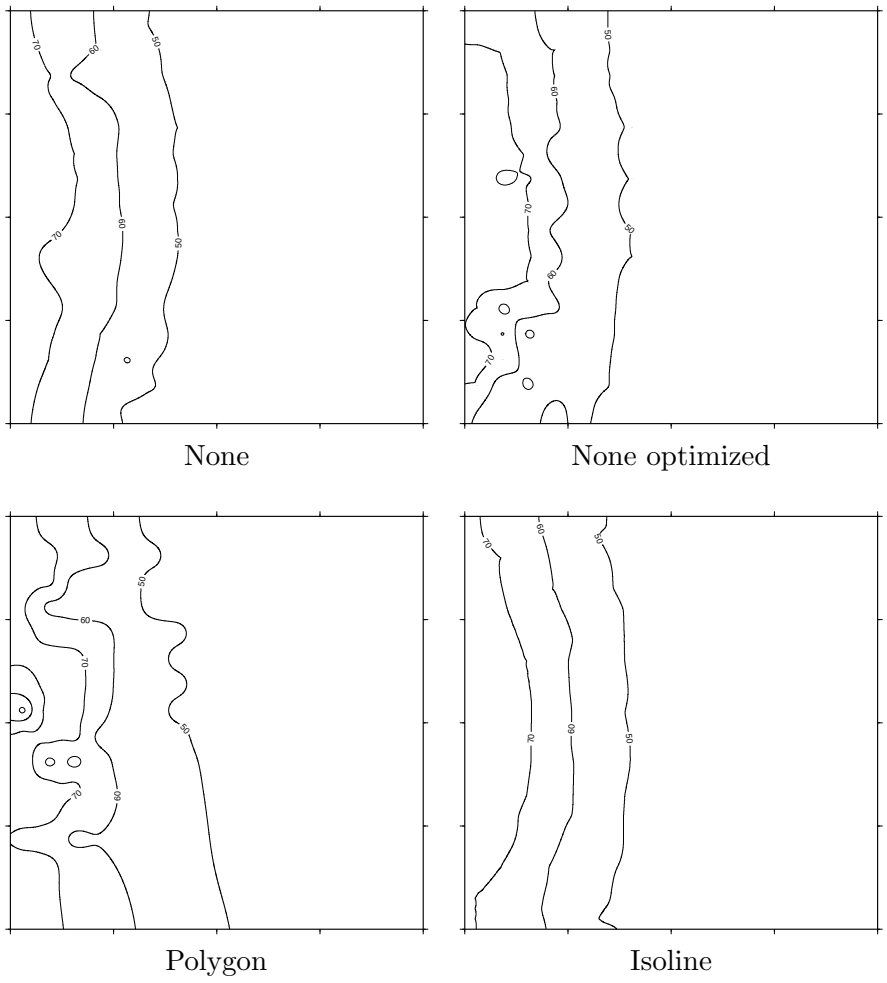


Figure 4.14: Front scenario, T=7.

reported is thus larger.

In terms of accuracy, *isoline aggregation* comes on top, with 96.7% similarity. Sending less data causes less contention at the MAC layer and therefore is able to achieve better accuracy. Note that this is true even if an underlying reliable MAC protocol were used. In fact, depending on the the reliability mechanism used by the MAC layer, higher contention and higher delays can be incurred.

The other optimized methods, i.e., "optimized" no aggregation and *polygons*, both have accuracy in the 80%s while still transmitting more data than *isolines*. The unoptimized no aggregation method exhibits 3.5% less similarity than *isolines* yet re-

porting over 3 times more data.

Figure 4.14 shows maps generated by the algorithms for the moving front at time $T = 7$. In the case of *isolines* and "unoptimized" no aggregation, the resulting graphs are relatively accurate. There are still some minor mistakes which happen from reports that got lost along the way. In the case of "optimized" no aggregation, the errors are different: they are "carry overs". Since nodes only report when data changes, and the map is constructed using old readings, a node whose update is lost might stay with an old value and create a semi-permanent error in the graph.

In the case of *polygon aggregation*, when a packet is lost, we lose an aggregated report, and hence we see areas where there are problems. *Isoline aggregation* reports quite accurately with some minor distortion near the edges due to the lack of nodes to sample reality and the interpolation algorithm.

It is important to note that the algorithms presented here might not be able to get 100% similarity under the contour similarity metric. This is because a perfect score implies that all of the reality points are placed in the correct ranges. This might not be possible since we are only subsampling the space. At some point it is necessary to increase sensor density to obtain higher accuracy. We expect that higher sensor density will help *isoline aggregation*'s accuracy. As pointed out previously, accuracy increases at the expense of energy savings. However, *isoline aggregation* minimizes this trade-off.

4.4.3 Dense random node placement

When isoline aggregation is used, a node will coordinate with nearby neighbors to determine if an isoline is present between them. If the isoline is found it is reported to the sink. Under "even" node deployments this by itself is a very efficient way to reduce the number of reports while maintaining high accuracy. However, in denser deployments, a node might have too many neighbors, which in turn might mean that there is an isoline between itself and a large number of nodes. This will create too much redundant reports and network traffic. To address this we have developed heuristics to suppress excessive reporting in dense deployments as discussed in Section 4.2.4.

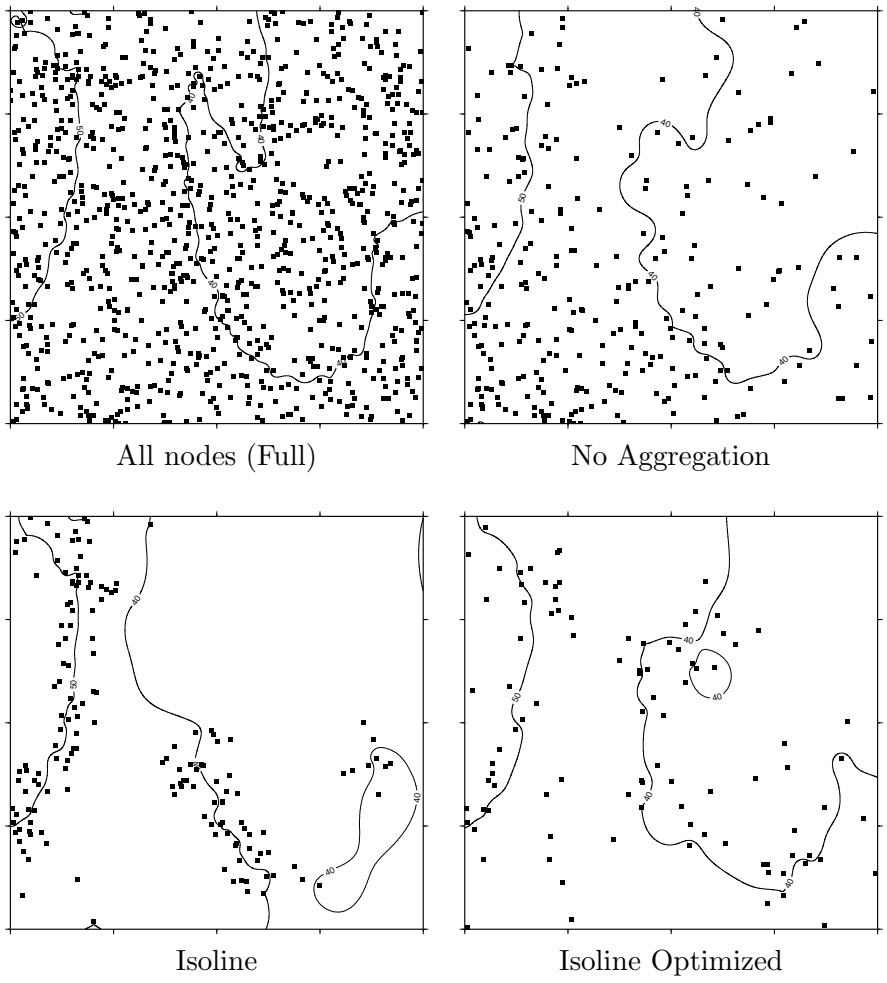


Figure 4.15: Mapping dense deployments with isolines

We repeat the same experiment performed in Section 4.4.1 and modified some parameters to test the performance of our optimizations for dense random node placement. For these experiments, we deploy 1024 nodes randomly over a 400x400m field. We then take a snapshot of the map (Figure 4.6) using no aggregation, isoline-, and optimized isoline aggregation.

The first map (top left) in Figure 4.15 shows the best possible scenario, i.e., the case where all the nodes report and their information is received at the sink. We call this the *Full* map which shows the 1024 nodes deployed in a random fashion. The second (top right) map shows the result from using *no-aggregation*. The dots show

	Similarity	KBytes sent	Nodes reporting
Full	96.6%		1024
No Agg.	89.5% (sd 1.63)	53.1K (sd 5.6K)	259
Isolines	83.8% (sd 5.91)	70.8K (sd 6.8K)	133
Opt. Isolines	87.2% (sd 3.69)	19.1K (sd 2.0K)	87

Table 4.7: Dense scenario: contour similarity, bytes sent and average nodes reporting

which readings are received at the sink. It is evident that it is drastically less than the total number of possible readings, in this case 272 out of 1024.

The following two figures show the maps obtained when we use *isolines*. The bottom left map depicts the results from plain *isoline aggregation* showing that, while in some high density areas we get a high number of redundant reports, in other areas no reports are received because of collisions and packet losses. The bottom right map shows the results of using *optimized isolines*. Note that in this case the reporting nodes are more evenly spread along the isolines in the map.

Table 4.7 summarizes results of the dense deployment experiments. The first thing to note is the fact that the similarity values are not as high as when grid placement is used. This is due to the fact that, in random placement, coverage is not as even as in grid placement and thus high density areas will be more prone to collisions and packet losses.

From the first row in Table 4.7, we observe that a maximum of 96.6% similarity can be achieved which is obtained when readings from all the sensor nodes are considered. Using *no aggregation* we can obtain 89.5% similarity, followed by *optimized isolines* with 87.2%. Plain *isolines* perform slightly worse due to increased packet drops.

The *isoline* algorithm incurs some overhead when compared to *no aggregation* because the nodes need to exchange neighbor information. This overhead is more evident in denser networks, both in packet drops, as seen by the lower similarity, and in bytes sent, as seen by an increase of almost 33%.

Optimized isolines addresses both these issues. A node will remember its parents and children from data collection tree establishment, then use only these nodes to detect isolines. This lowers the number of bytes sent to only 36% of what is required by *no aggregation*. At the same time the similarity is increased because of reduced number

of collisions in the high density areas. It is important to note that *optimized isolines* uses 35% less readings that *isolines* and achieves better similarity.

We should point out that choosing a subset of a node's neighbors based on the data collection tree incurs no extra overhead (in terms of extra information that needs to be exchanged) given that this information is already available to nodes.

4.5 Conclusions and Future Work

In this chapter, we introduced a novel data aggregation algorithm that targets spatially correlated data. *Isoline* aggregation uses local information from neighbors to group nodes that report similar readings. Energy efficiency is achieved by having only a subset of the nodes, i.e., the ones next to the isolines, report to the sink.

Simulation results show that *isoline* aggregation can lead to significant energy savings (some scenarios reported that no aggregation can send close to 150% more bytes than *isoline* aggregation) with adequate data accuracy. We also compared *isolines* against *polygon* aggregation, our implementation of an approach representing existing spatially-correlated data aggregation mechanisms (e.g., *eScan* and *isobars*). Our results report that *isolines* exhibit higher accuracy with a slight advantage in energy efficiency.

Continuous monitoring is one key application of sensor networks. We evaluated *isoline aggregation* as a collection technique targeting continuous mapping in sensor networks. It is a simple, elegant, and efficient algorithm for generating accurate maps of continuously changing conditions.

Two different continuous temperature monitoring scenarios were simulated, namely hotspot and moving front. In the hotspot scenario, a spot of activity appears and disappears; in the front scenario, temperature increase wave moves through the area. Both of these scenarios have real life analogies (e.g., fires, animal migration).

In the hotspot scenario, *isolines* delivered comparable accuracy while sending 38% less data than our nearest competitor, an optimized form of no aggregation, where nodes report only when the sensed value changes. The non-optimized alternative required more than 11 times the amount of data transmitted. For the front scenario, *isoline aggregation* delivered higher accuracy than all other protocols we studied (including *polygon aggregation*, which represented schemes like *eScan* and *isobars*, while

still yielding the highest energy efficiency.

We have optimized isolines so that it can handle dense random placement scenarios efficiently. We can achieve a similarity close to using no aggregation while using only 35% of the bytes sent.

One of our objectives is to test isoline aggregation under real scenarios. We plan to run multiple simulations with non synthetic data and study how well it performs. For this we will require a very accurate map of reality. Some of our future work also involves testing with a simple deployed scenario and comparing the performance to the data obtained with using no aggregation, that is all the data we can possible capture.

Our work on isolines has had two conference publications [38][37].

Chapter 5

Conclusions

Sensor networks have brought a new research field to the engineering community. The basic protocols that we've come to rely on for the stability and prosperity of the Internet have not been designed to deal with the requirements of these unique network scenarios. What we once considered very flexible is today very constricting. One of the biggest hurdles to overcome is in the power dimension, which was never addressed because of the original environments in which they were design to operate.

Typical sensor network deployments consist of numerous small nodes monitoring an area. These nodes normally feature a low power wireless radio, an embedded processor, some sensors (temperature, light, acceleration, etc) and a small battery. It has been normally assumed that placement will be uniformly random over the area to be monitored, though structured deployment is not that uncommon.

This thesis has dealt with various aspects involved in providing efficient protocols for these fringe networks. The first problem we tackled was that of a base protocol to build upon. We could have chosen to develop a custom protocol for all the scenarios encountered, however, this idea was soon ruled out. It would have involved having a set of protocols, all at layer 3, for all the possible exchanges. The second option was to create a generic protocol to deal with all the cases. This would have meant that all the functionality the network was going to provide would have to be build in from the very beginning. If we wanted the network to be able to do point-to-point communications we needed to add header fields for those exchanges, fields that would just be overhead in some data gathering scenarios. Since the network was going to be in this mode most

of the time, the overhead from these generic fields was unacceptable.

To provide the network with an efficient protocol, yet a fully functional one that would be able to tackle different scenarios we designed FLIP. FLIP is a flexible protocol whose headers can be customized to the functionality required. Via multiple simulations and scenarios we have showed that this functionality comes at a very minimal overhead and is very beneficial to these fringe network. It allows them to provide full functionality when required while permitting them to keep a very compact header for the day to day data gathering process. Based on the same ideology of FLIP we also started the design and development of GTP, a flexible protocol for the transport layer.

Having a solid ground to work on, we then proceeded to study various data gathering protocols, and found that most of them suffered from very high delay. The time it took for the data to get from the edge of the network to the collection point was very high. This was due mostly to the fact that when the original protocols were designed (i.e. Directed Diffusion [5]), delay was not a characteristic considered. The paramount goal was to save energy and other aspects were not closely considered.

We developed "Cascading Timers", a timing mechanism to make sure data gets to the sink with minimal delay when doing aggregation. The basic idea is that nodes will wait just the right amount of time for their children to transmit and then send the aggregated data to their parent. In our simulation scenarios by using aggregation we can reduce the data transmitted dramatically, up to 5 times, and by using "cascading timers" we can reduce the delay an order of magnitude. When doing aggregation however, we make ourselves vulnerable to very costly packet losses. When a packet gets lost near the sink it could mean half the information collected from the network has vanished. To address this we took a look at using multiple transmissions based on the weight of the packet (number of readings contained).

Aggregation is widely known to be a very effective way to save energy. Nodes will process data as it's flowing through the network. When dealing with simple data this is very simple, for example calculating the maximum temperature in the field. For this we take the values sent by all the children in the data gathering tree, compare it to our own reading, and send the maximum, effectively reducing the amount of data packets that are sent to the sink. Other data gathering scenarios are not as simple, this

is the case of mapping.

We investigated ways to optimize data gathering for map generation and how best to use aggregation. Previous approaches (i.e. isobars [23]) have addressed the problem by sending polygon data down the collection tree. This has various drawbacks, like requiring the polygon areas to be in the same data collection subtree in order to be aggregated, or the fact that location information has to be included in the packets or distributed to all the nodes. We approached the problem by gathering contour information using "Isoline aggregation".

When doing "Isoline Aggregation" nodes detect the presence of isolines (contours) by broadcasting their value to their neighbors. When an isoline is detected it's reported to the sink. This means that data is aggregated because we just send the required points to draw the contour and not every node in the middle of the region. This is basically done with only local information, reducing the required communications. We extended our work to do mapping for continuous environments, effectively providing a live monitoring protocol for sensor networks. Isolines can also deal with high node densities and random node placement while keeping the accuracy of the collected data high.

5.1 Future Directions

Over the course of this research there have been multiple paths that we haven't had time to explore. Some of the future directions include:

- Develop a closer interaction between FLIP, at the network layer, with the protocols at the MAC layer. Reducing redundant information where possible could give large gains on some scenarios. For example, if only local unique addresses are required then the MAC addressing could be used for network layer as well. Controlling how these fields could be handled by the higher level applications is something to be investigated.

- Continue the work on GTP and provide a fully functional transport layer API. Allow applications to select in a simple standard way what features they need their flows to have: connection/connection-less, ordered delivery, reliable deliver,

timely delivery, etc. A new socket family and some setsockopt functions could provide this sort of feature.

- The timing studied while developing Cascading Timers is very dependant on the lower layers providing very fast response. A contention MAC was assumed for the simulations, however, the trend in this area is to use scheduled MACs. This could have a disruptive effect on higher layer scheduling. Cooperation between the application, transport and network layer with the MAC is essential.

- On all our simulations we noticed a sharp effect when using different node placement. Looking closer at node placement schemes, specifically for mapping is something that could prove very fruitful.

- When studying mapping we ran out of time to test our protocols under real situations. Getting "reality" data to subsample and use for evaluating our protocol proved to be something challenging. Most data out there is already subsampled or at a scale that renders it ineffective for our uses. It might turn out that the only way to gather valuable data is to sample reality ourselves at a very high density, thus allowing us to run multiple tests on it.

Bibliography

[1] The bluetooth project. http://www.bluetooth.com/.

[2] A. Boulis, S. Ganeriwal, and M.B. Srivastava. Aggregation in sensor networks: An energy-accuracy trade-off. In *Proceedings of the First IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003.

[3] M.D. Addlesee, A.H. Jones, F. Livesey, and F.S. Samaria. The ORL active floor. *IEEE Personal Communications*, 4(5):35–41, October 1997.

[4] C. Intanagonwiwat, D. Estrin, R. Govindan and J. Heidemann. Impact of Network Density on Data Aggregation in Wireless Sensor Networks. In *Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS'02)*, July 2002.

[5] C. Intanagonwiwat, R. Govindan and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking (MobiCom)*. ACM, August 2000.

[6] DARPA. http://www.darpa.mil/ito/solicitations/glomo/ glomobrief.html, February 1995.

[7] R. Katz et al. The daedalus project. http://daedalus.cs.berkeley.edu/, January 1996.

[8] F. Benner, D. Clarke, J. Evans, A. Hopper, A. Jones and D. Leask. Piconet: Embedded mobile networking. *IEEE Personal Communications*, 4(5):8–15, October 1997.

[9] C. Florens and R. McEliece. Packet distribution algorithms for sensor networks. In *Proceedings of IEEE INFOCOM 2003*, April 2003.

[10] P. Osborn G. Girling, J. Li Kam Wa and R. Stefanova. The pen low power protocol stack. *9th IEEE International Conference on Computer Communications and Networks*, October 2000. ftp.uk.research.att.com:/pub/docs/att/tr.2000.12.pdf.

[11] G. Pottie and W. Kaiser. Wireless integrated network sensors. *Communications of the ACM*, 43, Issue 5, May 2000.

[12] The VINT Group. VINT:virtual internet testbed. http://netweb.usc.edu/vint, 1996.

[13] R. Frederick H. Schulzrinne, S. Casner and V. Jacobson. Rfc1889, RTP: A transport protocol for real-time applications, January 1996.

[14] A. Harter and F. Bennett. Low bandwidth infra-red networks and protocols for mobile communicating devices. Olivetti Research Laboratory Technical Report.

[15] I. Solis, J. Marcos and K. Obraczka. FLIP: a flexible protocol for efficient communication between heterogeneous devices. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, July 2001.

[16] IETF. http://www.ietf.org/html.charters/manet-charter.html, July 2000.

[17] J. B. Postel. User Datagram Protocol. Technical Report RFC-768, IETF, August 1980.

[18] J. B. Postel. Internet Protocol, September 1981. IETF RFC-791.

[19] J. B. Postel. Transmission Control Protocol. Technical Report RFC-793, IETF, September 1981.

[20] J. Elson and D. Estrin. An address-free architecture for dynamic sensor networks. Technical Report 00-724, Computer Science Department USC, January 2000.

[21] J. Elson, L. Girod and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, December 2002.

[22] J. Lilley, J. Yang, H. Balakrishnan and S. Seshan. A unified header compression framework for low-bandwidth links. In *International Conference on Mobile Computing and Networking (MobiCom)*. ACM, August 2000.

[23] J. M. Hellerstein, W. Hong, S. Madden and K. Stanek. Beyond average: Towards sophisticated sensing with queries. In *Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks (IPSN '03)*, March 2003.

[24] J. Zhao, R. Govindan and D. Estrin. Residual energy scans for monitoring wireless sensor networks. In *Proceedings of the IEEE Wilress Communications and Networking Conference (WCNC'02)*, pages 17–21, March 2002.

[25] K. Almeroth, K. Obraczka and D. De Lucia. A lightweigth protocol for interconnecting heterogeneous devices in dynamic environments. In *International Conference on Multimedia Computing and Systems (ICMCS)*. IEEE, June 1999.

[26] B. Kantor and P. Lapsley. Network News Transfer Protocol. IETF RFC-977, February 1986.

[27] L. Doherty and K. S. J. Pister . Scattered data selection for dense sensor networks. In *Proceedings of the Third Symposium on Information Processing in Sensor Networks*, April 2004.

[28] M. Leonov. PolyBoolean Library, February 2004. http://www.complex-a5.ru/polyboolean/.

[29] M. Degermark, B. Nordgren, S. Pink. IP header compression, RFC-2507, February 1999.

[30] M. Degermark, M. Engan, B. Norgreen and S. Pink. Low-loss TCP/IP header compression for wireless networks. In *International Conference on Mobile Computing and Networking (MobiCom)*. ACM, November 1996.

[31] National Center for Atmospheric Research. NCAR Graphics Library, February 2004. http://ngwww.ucar.edu/ng4.3/.

[32] R.E. Blahut. *Theory and Practice of Error Control Codes*. Addison Wesley, 1984.

[33] S. E. Deering and R. Hinden. Internet Protocol, version 6 (IPv6) specification. IETF RFC-1883, December 1995.

[34] I. Solis and K. Obraczka. The case for a flexible-header protocol (FLIP) in power constrained networks. In *Proceedings of the IEEE Wireless Communication and Networking Conference (WCNC)*, March 2003.

[35] I. Solis and K. Obraczka. FLIP: A flexible interconnection protocol for heterogeneous internetworking. *ACM/Kluwer Mobile Networking and Applications (MONET) Special on Integration of Heterogeneous Wireless Technologies*, August 2004.

[36] I. Solis and K. Obraczka. The impact of timing in data aggregation for sensor networks. *Proceedings of the IEEE International Conference on Communications (ICC)*, June 2004.

[37] I. Solis and K. Obraczka. Efficient continuous mapping in sensor networks using isolines. In *Proceedings of the International Conference on Mobile and Ubiquitous Systems (Mobiquitous)*, July 2005.

[38] I. Solis and K. Obraczka. Isolines: Energy efficient mapping in sensor networks. In *Proceedings of the IEEE Symposium on Computers and Communications (ISCC)*, June 2005.

[39] S.R. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the Symposium on Operating Systems Design and Implementation, OSDI*, December 2002.

[40] USC/ISI. SCADDS: Scalable Coordination Architectures for Deeply Distributed Systems. http://www.isi.edu/scadds/, November 2001.

[41] USC/ISI, UCLA, Virginia Tech. Dynamic sensor networks (DSN). http://www.east.isi.edu/DIV10/dsn/, August 2000.

[42] L. Schwiebert V. Annamalai, S.K.S Gupta. On tree-based convergecasting in wireless sensor networks. In *Proceedings of the IEEE Wilress Communications and Networking Conference (WCNC'03)*, March 2003.

[43] Packet radio topics in professional journals, http://www.tapr.org/tapr/html/biblio.html, August 2000.

[44] VINT. The Network Simulator NS-2. http://www.isi.edu/nsnam, November 2001.

[45] W. Heinzelman, J. Kulik and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedins of the International Conference on Mobile Computing and Networking (MobiCom)*, August 1999.

[46] R. Want, A. Hopper, V. Falcao, and J. Gibbons. The active badge location system. *ACM Transactions on Information Systems*, 10(1):91–102, January 1992.

[47] Mark Weiser. Ubiquitous computing. http://www.ubiq.com/hypertext/weiser/weiser.html.

[48] Mark Weiser. Research reports of the infrastructure for ubiquitous computing project. http://www.ubiq.com/weiser/researchreports.htm, March 1996.

[49] H. Zimmermann. OSI reference model - the ISO model of architecture for open systems interconnection. *IEEE Transactions on communications*, April 1980.