FLIP: A Flexible Interconnection Protocol for Heterogeneous Internetworking

Ignacio Solis and Katia Obraczka
{isolis,katia}@cse.ucsc.edu
Computer Engineering Department
University of California, Santa Cruz

September 9, 2003

Abstract

This paper describes the Flexible Interconnection Protocol, or *FLIP*, whose main goal is to allow interconnection of heterogeneous devices with varying power, processing, and communication capabilities, ranging from simple sensors to more powerful computing devices such as laptops and desktops. The vision is that FLIP will be used to interconnect such devices forming *clouds* in the farthest branches/leaves of the Internet, while still providing connectivity with the existing IP-based Internet infrastructure. Through its flexible, customizable headers FLIP integrates just the functions required by a given application and that can be handled by the underlying device. Simple devices like sensors will benefit from incurring close to optimal overhead saving not only bandwidth, but, more importantly, energy. More sophisticated devices in the cloud can be responsible for implementing more complex functions like reliable/ordered data delivery, communication with other device clouds and with the IP infrastructure.

FLIP is designed to provide a basic substrate on which to build network- and transport-level functionality. In heterogeneous environments, FLIP allows devices with varying capabilities to coexist and interoperate under the same network infrastructure. We present the basic design of FLIP and describe its implementation under Linux. We also report on FLIP's performance when providing IPv4 and IPv6 as well as transport-layer functionality a la TCP and UDP. We show FLIP's energy efficiency in different sensor network scenarios. For example, we use FLIP to implement the directed diffusion communication paradigm and obtain an improvement of 50% in energy savings over an existing directed diffusion implementation. Finally, we showcase FLIP's flexibility by demonstrating its ability to incorporate new protocol functions seamlessly. In particular, we add data aggregation functionality onto FLIP and show that it significantly increases the system's energy efficiency.

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. More recently, protocols specifically tailored for sensor networks have been implemented. Because they are so specialized, these pro-

tocols will not be able to accommodate more sophisticated and powerful devices.

In this paper, 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.

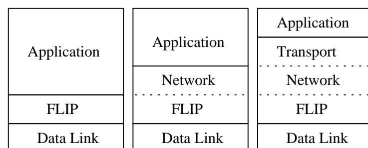
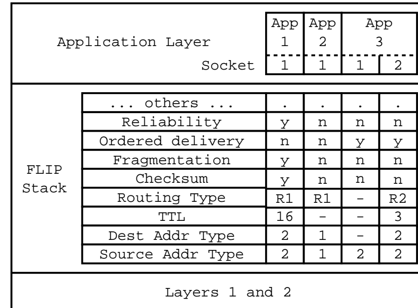


Figure 1: FLIP in the protocol stack.

Figure 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 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 3, we implemented a BSD socket interface that provides the application layer with access to FLIP.



Application Layer	App 1	App 2	App 3	
Socket	1	1	1	2
FLIP Stack				
... others ...	.	.	.	.
Reliability	y	n	n	n
Ordered delivery	n	n	y	y
Fragmentation	y	n	n	n
Checksum	y	n	n	n
Routing Type	R1	R1	-	R2
TTL	16	-	-	3
Dest Addr Type	2	1	-	2
Source Addr Type	2	1	2	2
Layers 1 and 2				

Figure 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 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 [14] and IPv6 [22]), 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

[1] Section 2.1 provides a more detailed description of FLIP's fields

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 4, FLIP is used to provide IPv4 and IPv6 functionality. Section 5 evaluates how well FLIP matches the needs of *directed diffusion* [2], while still being power-efficient. *Directed diffusion* (described in more detail in Section 5.1) 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 6 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 7, 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 and 3 describe FLIP's basic design principles and a reference implementation under Linux, respectively [2]. Related work is discussed in Section 8 and Section 9 presents our concluding remarks.

2 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 [11]

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 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 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.

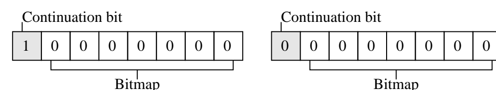


Figure 3: FLIP meta-headers

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 4 depicts both ESP cases.

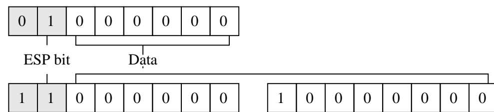


Figure 4: Extra Simple Packet (ESP)

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 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.

- **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.

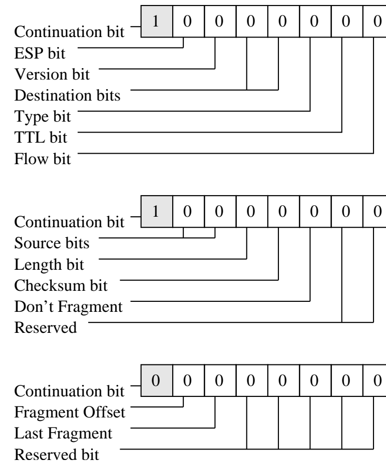


Figure 5: The FLIP meta-header bitmap

- **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 [14] addresses correspond to 4-byte FLIP addresses and IPv6 [22] 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 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

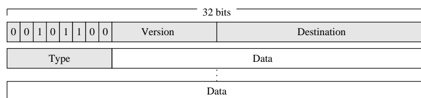


Figure 6: FLIP sample packet

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.

2.2 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.

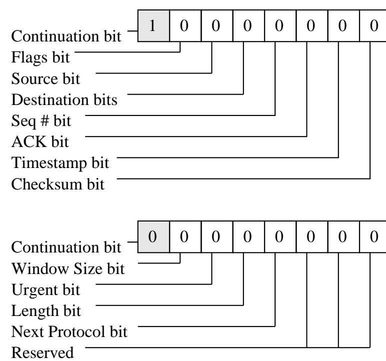


Figure 7: GTP meta-header

Figure 7 depicts GTP's meta-header and its fields. GTP's header includes fields used by existing transport protocols, specifically TCP [15], UDP [13], and RTP [9]. 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 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.

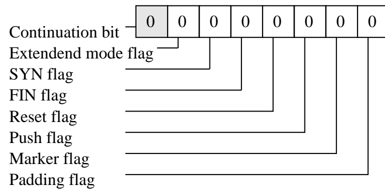


Figure 8: GTP Flags field

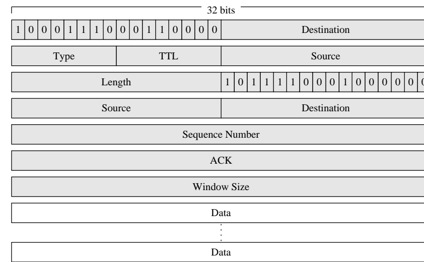


Figure 9: FLIP/GTP packet

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 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 4, 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.

2.3 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.

3 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 10: Sample application code

Figure 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 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.

4 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.

4.1 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

Table 1: FLIP-IP comparison in terms of packet size

Data	none			1 byte			1000 bytes		
	IPv4	IPv6	FLIP	IPv4	IPv6	FLIP	IPv4	IPv6	FLIP
Full IPv4 functions	20	N/A	24	21 (2000%)	N/A	24 (2300%)	1020 (2%)	N/A	1024 (2.4%)
Typical IPv4 functions	20	N/A	17	21 (2000%)	N/A	18 (1700%)	1020 (2%)	N/A	1017 (1.7%)
IPv6 functions	N/A	40	44	N/A	41 (4000%)	45 (4400%)	N/A	1040 (4%)	1044 (4.4%)
Dest. & Source	20	40	10	21 (2000%)	41 (4000%)	11 (1000%)	1020 (2%)	1040 (4%)	1010 (1%)
Dest. only	20	40	3	21 (2000%)	41 (4000%)	4 (300%)	1020 (2%)	1040 (4%)	1003 (0.3%)

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 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 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 no payload), 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 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 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.

Table 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)

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 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.

4.2 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.

5 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 [16] 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 [2]. 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.

5.1 Directed Diffusion and FLIP

Directed diffusion [2] 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.

5.2 Diffusion over FLIP

We use the diffusion packet definition from their 3.0 beta release [23]. Figure 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.

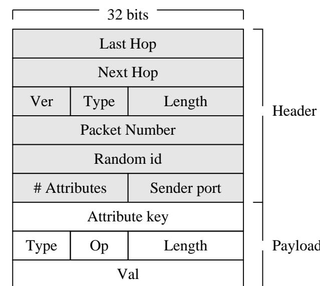


Figure 11: Diffusion packet for an `int` attribute

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.

5.3 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 12 shows the resulting optimized diffusion packets. Shaded and unshaded areas denote header and payload, respectively.

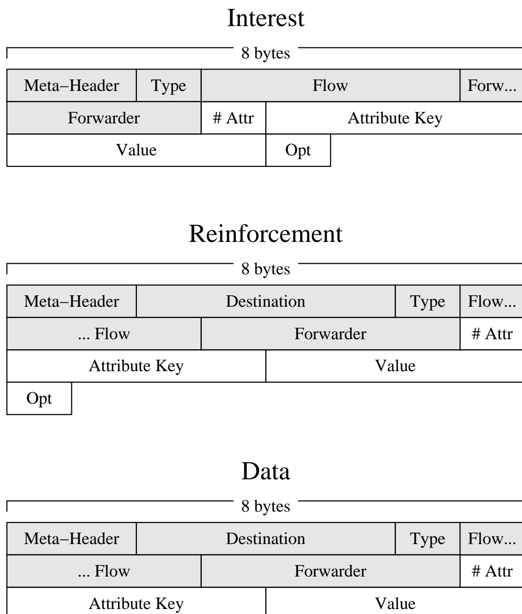


Figure 12: FLIP-optimized diffusion headers

5.4 Simulation Results

To evaluate these FLIP-based diffusion variants, we modified the original diffusion code in the `ns-2` network simulator [8]. 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 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 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

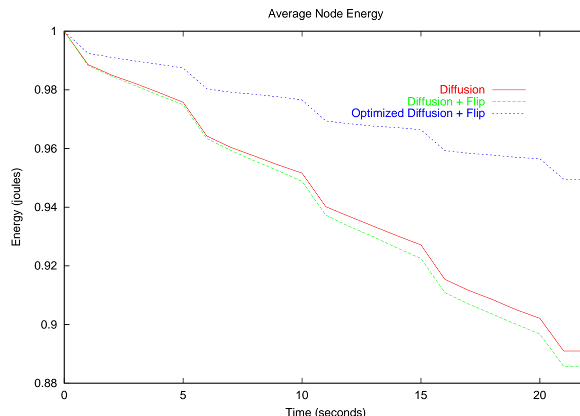


Figure 13: Energy levels over time for different diffusion variants.

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.

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 3 summarizes energy consumption results for the different diffusion variants.

Table 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)

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

6 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 7. We describe the different packet formats below.

6.1 Header Models

We compare FLIP's flexible headers with two static header models: *minimal* and *full* headers. Figure 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.

In this particular application scenario, since flows in different directions need to carry different information,

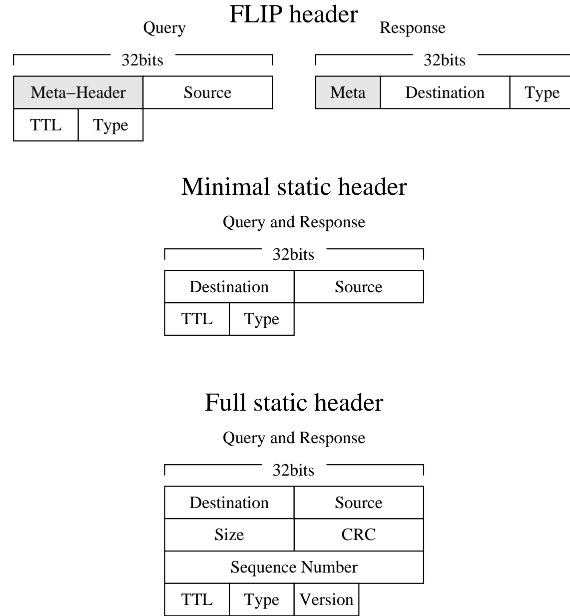


Figure 14: Header models

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.

6.2 Simulation Results

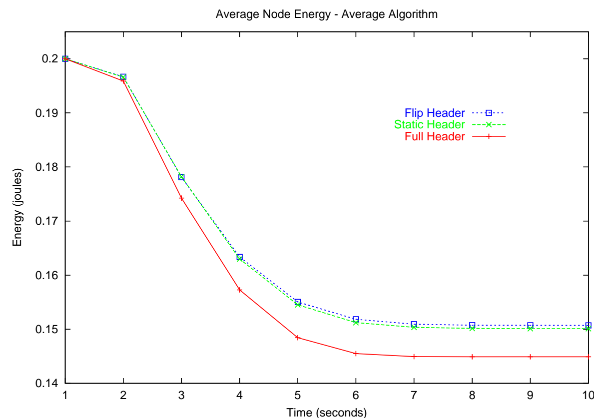


Figure 15: Data gathering (temperature averaging) energy consumption

The graph in Figure 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 [2]. 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.

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 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

Table 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.

7 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 6 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 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.

Data aggregation at ring nodes compensates for energy consumption at nodes within the requester's importance

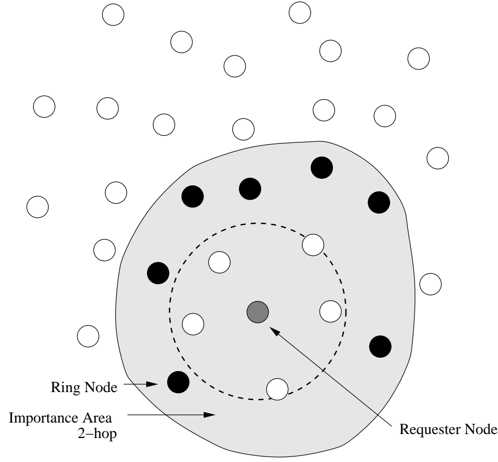


Figure 16: Sample ring aggregation scenario

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 5.1 and 6. 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.

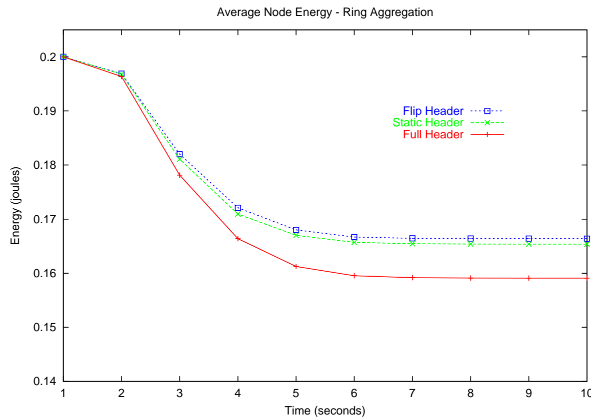


Figure 17: Data gathering (temperature averaging) energy consumption with data aggregation.

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

Table 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 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.

Table 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%

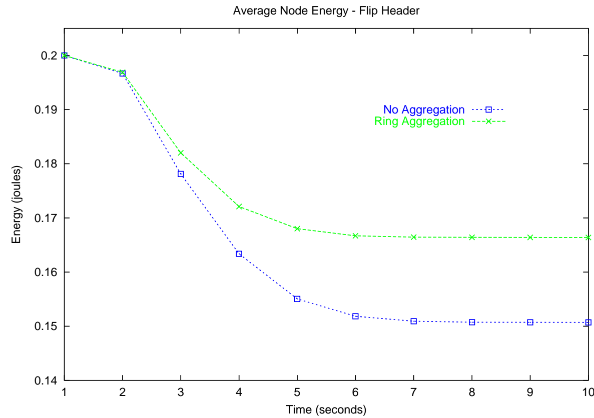


Figure 18: Effect of data aggregation on energy consumption

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

ity 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.

8 Related Work

Communication protocols for wireless networks have been an active area or research and include efforts such as Packet Radio [25], GloMo [3] and the IETF's Mobile Adhoc Networks (manet) working group [12]. In the early 90's, several efforts focused on the concept of "ubiquitous computing" [27]. Some examples include a number of projects at Xerox PARC [28] and the Daedalus/BARWAN project [4] 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. [18] introduced the main concepts behind FLIP.

AT&T Laboratories Cambridge (former Olivetti Research Labs) has lead several related initiatives including: Piconet [5] and its low-range radio network, an infra-red (IR) network [10] connecting active badges and IR-base sensors. Their efforts to develop a low-power protocol stack [6] 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 [26], and the Active Floor [1].

In the Scalable Coordination Architectures for Deeply Distributed Systems (SCADDS) project [23], nodes loose their individuality and the focus lies in the data generated by the whole system. In this context, the *directed diffusion* [2] 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 5.1, 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 [24] 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 [7], describes a basic sensor network environment and presents a solution based on layered processing.

Research such as the Address Free Architecture [16] 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 [21] 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 [17] aims at creating a standard way in which protocols in general can define header compression. Previous work [19][20] 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.

9 Conclusions

This paper 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 compared GTP to TCP/UDP and showed that it yields increased efficiency when providing transport level func-

tionality 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 redesigned 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.

References

[1] 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.

[2] 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.

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

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

[5] 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.

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

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

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

[9] H. Schulzrinne, S. Casner, R. Frederick, V. Jacobson. RTP: A transport protocol for real-time applications. Technical Report RFC-1889, IETF, January 1996.

[10] A. Harter and F. Bennett. Low bandwidth infra-red networks and protocols for mobile communicating devices. Olivetti Research Laboratory Technical Report, May 1993.

[11] 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.

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

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

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

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

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

[17] 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.

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

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

[20] 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.

[21] Bluetooth Project. Bluetooth specification. http://www.bluetooth.com/dev/specifications.asp, 2003.

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

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

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

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

[26] 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.

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

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