

Language Support for Interoperable Messaging in Sensor Networks

Kevin Chang^{*}
Department of Computer Science
University of California,
Los Angeles, CA 90095-1596
kchang@cs.ucla.edu

David Gay
Intel Research Berkeley
2150 Shattuck Ave, Suite 1300
Berkeley, CA 94704
david.e.gay@intel.com

ABSTRACT

Development of network communication in a homogeneous sensor network environment is straightforward as the nodes can share message layouts simply by letting the compiler lay out messages in an arbitrary fashion and using the same executable code on all nodes. However, this simple approach does not usually work in a heterogeneous sensor network setting because different compilers may generate different message layouts, and different processors often have different basic type representations and alignments. The traditional solutions to this problem is to either require programmers to insert network-byte-order and host-byte-order conversions, or to use a compiler that automatically generates marshalling and unmarshalling routines. Unfortunately, these approaches are inadequate for sensor networks because they are either error-prone and/or add significant overheads to already resource-constrained sensor motes. Instead, we propose a language extension — *network types* — which supports heterogeneous networking in a simple and efficient way. We have implemented network types in the nesC, the language of the TinyOS sensor network operating system and its applications. We have used network types to supports heterogeneous networking between micaz and telos motes (which have different alignment restrictions). We also show that our implementation introduces a negligible amount of overhead in runtime and code size. Network types have the additional benefit of requiring few changes to existing TinyOS code.

Categories and Subject Descriptors

D.3.3 [Programming Languages]: Language Constructs and Features—*Data types and structures*; C.2.0 [Communication Networks]: General—*Data communications*

General Terms

Design, Languages

^{*}Partially supported by the NSF ITR award 0427202

Keywords

Data layout, Heterogeneous networks, Sensor networks

1. INTRODUCTION

Early sensor network platforms such as the mica [1] and mica2 use non-standard, platform-specific radios that are not interoperable with one another. Consequently, existing sensor networks have mostly consisted of homogeneous nodes. In May of 2003 the IEEE approved the 802.15.4 radio MAC (medium access control) and physical layer standard, standards which are designed specifically for Low Rate Wireless Personal Area Networks (LR-WPANs). Several new sensor network platforms such as the micaz and telos support this new radio standard, allowing them to communicate with each other. This allows designers to build hierarchical networks from heterogeneous nodes, leading to improved scalability, flexibility and lifetime [2, 3]. In addition, many sensor network designers already require some degree of heterogeneity for development and testing in simulated frameworks such as the TinyOS Simulator [4] and EMStar [5].

It is thus highly desirable for these platforms to interoperate with one another. The 802.15.4 standard addresses packet representation up to the MAC layer; further standardization of packet formats is up to the higher layers of the protocol stack and the application. In the case of TinyOS [6] — the operating system of the mica, mica2, micaz and telos motes — these protocols are currently specified in a platform-dependent manner, precluding interoperation.

This problem is of course not new. One standard solution is to define a platform-independent data layout language such as ASN.1 [7] or XDR [8] to define packet representations, and use automatically-generated *marshalling* and *unmarshalling* routines to convert this representation to/from some host-specific representation. Another approach is to have the programmer insert calls to data format conversion functions (e.g., Unix's `htons`, `ntohs`, etc functions) when accessing packet fields. We argue (Section 4) that the first approach is inappropriate for resource-constrained sensor network nodes (a few kB of RAM, must run off batteries for months), and that the second is hard to make portable, time-consuming and error-prone.

In this paper, we propose a different solution based on extending the nesC programming language [9], used to implement TinyOS and its applications. We add new types and type-constructors (arrays, records) whose representation is platform-independent. We call these types *network types*; a program which only uses network types to access packets will automatically support heterogeneous

environments. Network types have several advantages. First, as they are integrated into the language’s type system, they are very easy to use: a received packet (array of bytes) can simply be cast to a network type, and accessed like a regular data structure. Second, because existing TinyOS code uses a similar model, it is very easy to modify existing TinyOS programs to use network types. Finally, using network types does not require any extra RAM resources, which is important on severely resource-constrained sensor network nodes. One disadvantage of network types is that accesses to packets may incur extra overhead, as the host’s load/store instructions may not be usable because of alignment restrictions. We show (Section 5) that this overhead is very low in practice.

Network types are available in version 1.2 of the nesC programming language, available from SourceForge. TinyOS 2.0, currently under development, will use network types for all its network protocols.

The rest of this paper is structured as follows: Section 2 surveys related work. Section 3 presents background material on TinyOS, nesC, and the hardware platforms they support. Section 4 presents our rationale for network types, their design, and their implementation in the nesC compiler. Section 5 presents our evaluation of network types on a set of applications and platforms. We conclude in Section 6 with a discussion of possible language extensions and future directions.

2. RELATED WORK

Many languages have been proposed for defining platform independent data-types, of which the most popular are ASN.1 (Abstract Syntax Notation One) [7] and XDR (External Data Representation) [8]. ASN.1 separates data type definition and data object encodings, and has multiple encoding standards including Basic [10] and Packed [11] Encoding Rules. Unlike network types, these are typically used with stub generators to convert between the network representation and some corresponding type in a particular programming language. There are many marshalling function generators, supporting various source data-description languages and target programming languages, including USC (Universal Stub Compiler) [12], Flick [13] and MAVROS [14]. In Section 4.1 we discuss why this approach is inappropriate for severely hardware constrained sensor networks.

Distributed object paradigms to solve data heterogeneity include Java RMI/Serialization, CORBA [15], and DCOM [16]. These paradigms use explicit typing and are too heavyweight for memory- and bandwidth-constrained motes.

The definition of records in Ada [17] includes “representation clauses” to specify some aspects of in-memory record representation. However, these are not sufficient to guarantee a platform-independent representation. In particular, these clauses need not be respected by a compiler, and there is no specification of byte-endianness for multi-byte integers (though bit-endianness can be specified). C++’s facility to overload assignment and define implicit conversions can be used to build integer types with a specific layout. Combining this with gcc’s `attribute((packed))` extension gives a facility similar to network types. However, this relies on a compiler-specific extension, cannot be extended to support bitfields and does not allow for network-type-specific compile-time checks and optimizations.

Unsurprisingly, languages designed for network processors include

facilities similar to network types. For instance, in Intel’s NCL language [18, Chapter 16] a programmer can define a `protocol` and specify the names, layouts and representations of its fields. However, NCL is not a general purpose programming language that can be readily ported for use in existing sensor network systems that use TinyOS.

3. BACKGROUND

We briefly cover TinyOS [6], it’s nesC [9] programming language, and the main TinyOS hardware platforms.

3.1 nesC and TinyOS

TinyOS is the most popular operating system for sensor network motes, and is used by hundreds of groups worldwide. TinyOS programs are built by assembling components, which contain application logic, operating system services (e.g., timers), and, in particular, different layers of the network stack. TinyOS does not contain a single network stack, rather application designers build their own stack by selecting amongst compatible components for, e.g., multi-hop routing. TinyOS programs are written in nesC, a component-oriented programming language. nesC’s components are either *configurations*, which connect components together, or *modules*, which contain executable code written in a C variant. The use of C for executable code provides all the low-level features necessary for accessing hardware resources. The nesC compiler generates C, which is then compiled by a platform-specific C compiler. Most current TinyOS network protocols and applications use C structs to define packet layouts. As a result, TinyOS applications built on different hardware architectures do not interoperate.

3.2 Hardware Platforms

TinyOS runs on many different platforms, including CrossBow’s `mica`, `mica2`, `mica2`¹, and Moteiv’s `telos`². Additionally, TinyOS programs can be run in a simulation environment on PC’s. Characteristics of these platforms are shown in Table 1. Alignment is the largest alignment of any type on the given processor, e.g., 16-bit integers must be aligned on a 2-byte boundary on the TI MSP 430. While the x86 does not have alignment requirements, compilers do align data to improve performance. As this table shows, platforms are in constant evolution. In addition, most of the software abstractions are specialized and/or are still in flux [19], making portability and maintenance a necessary task.

Earlier platforms such as the `mica`, and `mica2` use non-standard radios, with different frequencies, modulation, data rates, and protocols, and are thus unable to communicate with one another. The newer `mica2` and the `telos` conform to the IEEE 802.15.4 standard. These last two are the platforms we use in our evaluation of network types. Processing power is relatively plentiful given the nature of sensor networks; sensor data acquisition programs tend to be inactive most of the time and require minimal data rate on the radios. On the other hand, RAM is the most constrained resource.

4. DESIGN

We start by considering existing approaches to supporting heterogeneous networks (Section 4.1) before presenting the design (Section 4.2) and implementation (Section 4.3) of network types.

A few basic principles guide our decisions. First, we want the compiler to automatically convert data for use on the host with minimal

¹<http://www.xbow.com>

²<http://www.moteiv.com>

Mote Type	mica	mica2	micaz	telos	Tmote sky	PC
						
Date	2/02	10/03	10/04	9/04	3/05	8/81

Microcontroller				
Type	Atmega103	Atmega128	MSP 430	x86
Frequency	4MHz	7.2MHz	8MHz	GHz
Size (bit)	8		16	32
Code (KB)	128		60	48
RAM (KB)	4		2	10
Alignment	1		2	4
Endianness	little			

Communication				
Radio	RFM TR1000	CC1000	802.15.4	none
Rate (Kbps)	10	20	250	
Modulation	ASK	FSK	O-QPSK	

Table 1: Popular TinyOS platforms.

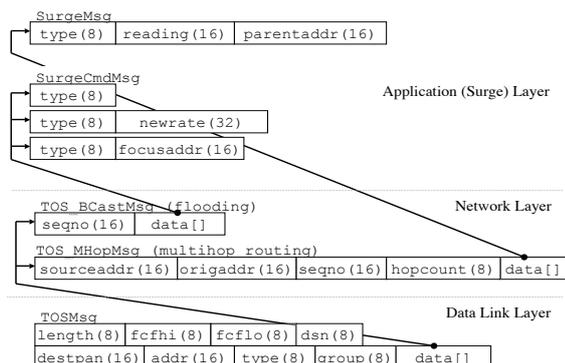


Figure 1: Layers and Packet Layout in Surge

programmer assistance. In addition, we want to minimize the CPU and, especially, RAM overhead of data conversion. Lastly, we want to make it easy for programmers to migrate legacy TinyOS code.

Throughout this section we use the Surge application from TinyOS to illustrate some of the issues and potential solutions in supporting heterogeneous network environments. Surge is an application that performs periodic sensor sampling, uses multi-hop routing to deliver packets, and responds to simple commands. Figure 1 shows the three layers of the Surge protocol stack, and their data layouts. Each line represents a protocol’s layout, each rectangle represents a field in that layout and includes its size in bits. A `data[]` field is a variable-size payload for the next layer. For example, `TOS_BCastMsg`, used for flooded commands, contains just a 16-bit sequence number (`seqno`) and the next layer’s payload (`data`). These layouts are all currently defined using C structs.

4.1 Existing Approaches to Interoperability

We consider three standard solutions for attaining interoperability between network protocols, and discuss why they are inappropriate in TinyOS.

The first approach is to define a type M which represents all possible packets that can be sent and received by the application, and use marshalling and unmarshalling routines to convert values of type M between network (platform-independent) M_n and host M_h representations. Programs can operate on values of type M_h , and the marshalling and unmarshalling routines can be called at the lowest level of the network stack, just before messages are sent and just

after they are received. In the Surge example, M is a type that represents all of Figure 1.

A major problem with this approach is obtaining the definition of M . Defining M manually is a laborious process in the TinyOS model, as the programmer would have to examine all the components used in the network stack and extract their packet layout information. This is repetitive and time consuming, and leads to maintenance problems when, e.g., the programmer decides to use a new, improved routing component with a different packet header. To extract M automatically is also hard: the compiler needs to know which components form the network stack, how they are assembled, and what packet layout they use. The presence of variable-size headers (common in existing protocols such as TCP/IP) would further complicate this process. We thus rejected this approach as requiring excessive language (to declare protocol-related issues) and compiler changes.

The second approach is similar, but defines a type M_p for each protocol, and uses marshalling and unmarshalling routines to convert values of type M_p between network M_{pn} and host M_{ph} representations in each protocol layer. A TinyOS component can call these marshalling and unmarshalling functions just after receiving from, and just before sending packets to, lower layers. In Surge, there would be marshalling functions for the `TOSMsg`, `TOS_MHopMsg`, `TOS_BCastMsg`, `SurgeCmdMsg` and `SurgeMsg` types.

This approach requires a copy of data at each layer of the protocol stack. The CPU overhead of this is not that significant (sensor network messages are small), but each copy also requires an extra RAM buffer: because of alignment restrictions M_{ph} may be larger than M_{pn} so conversion cannot happen in-place.³ Section 5 reports on the costs of this approach on micaz motes. This approach would also require significant changes in existing TinyOS components, to include extra buffers, define the M_p types, include the marshalling functions, and add appropriate calls to them. We thus also rejected this approach.

The third approach is similar to current TinyOS practice: a data structure is defined in C which matches the packet layout, and the programmer manually inserts calls to functions like Unix’s `hton`s (16-bit int host-to-network format conversion) to deal with endianness issues. This programming style is used, e.g., for the Unix

³The network representation could make some pessimistic alignment assumptions but this would waste space, and hence energy, in every radio packet.

TCP/IP socket functions. In the Surge example, this would correspond to adding conversion functions on all accesses to fields such as `reading`, `parentaddr`, etc.

This approach has two major problems. First, as in TinyOS, the code may not be portable as C and most other languages provide few guarantees as to data structure layout. Thus the structure which matches the packet layout on one platform may not on another because of alignment issues. This portability problem can be avoided by using only arrays of characters addressed via constant offsets, but this makes code unreadable and hard to maintain. Second, adding the manual conversion functions is an error-prone process. This approach would represent the smallest change from current TinyOS practice, but would provide the programmer little help in supporting heterogeneous network environments.

4.2 Network Types

Network types, our extension to nesC for platform-independent networking, is closest in spirit to the third approach considered above: the programmer defines a data structure, using syntax very similar to existing C type declarations, which matches the packet layout, and just accesses it like a regular C type. The compiler ensures that this type has the same representation on all platforms and generates any necessary conversion code. For instance, the `SurgeCmdMsg` type definition using network types is:

```
typedef nx_struct {
  nx_uint8_t type;
  nx_union {
    nx_uint32_t newrate;
    nx_uint16_t focusaddr;
  } args;
} SurgeCmdMsg;
...

SurgeCmdMsg *pCmdMsg = (SurgeCmdMsg *)payload;
if (pCmdMsg->type == SURGE_TYPE_SETRATE) {
  timer_rate = pCmdMsg->args.newrate;
  ...
}
```

Unlike the two marshalling-based approaches, no extra buffers are needed. Additionally, the compiler does not need any global knowledge about the network stack to generate the conversion code. The disadvantage is that if the program accesses certain data fields repeatedly, conversion occurs repeatedly as well, causing higher runtime costs than the marshalling/unmarshalling method. However we find that in practice (Section 5) these overheads are low, and dwarfed by the cost of sending or receiving a message.

Formally, nesC includes three kinds of network types. *Network base types* are similar to the fixed size types defined in `inttypes.h`, they include 8, 16, 32 and 64-bit signed and unsigned integers denoted by the types `nx_int8_t`, `nx_uint8_t`, `nx_int16_t`, etc. *Network array types* are any array built from a network type, using the usual C syntax, e.g. `nx_int16_t x[10]`. *Network structures* are declared like C structures and unions, but using the `nx_struct` and `nx_union` keywords (as in the `SurgeCmdMsg` example above). A network structure can only contain network types as elements. All these network types can be used exactly like regular C types, with a few restrictions which will be lifted in future versions:

- Network base types cannot be used in casts, as function arguments and results (the programmer can simply use the corresponding non-network types such as `uint16_t`).

- Declarations of variables of a network base type cannot have an initializer.
- Network structures cannot contain bit-fields.

Network types have no alignment restrictions, network structures contain no padding, and the network base types use a 2's complement, big-endian representation. Thus their representation is platform-independent, and any arbitrary section of memory can be accessed via a network type. The endianness of the base types is big-endian as this is the dominant endianness in existing networking protocols.⁴

Network bit-fields are not currently supported. This limitation is acceptable as they are not widely used in existing TinyOS programs. A careful inspection of the TinyOS repository which includes code contributed by many different research institutions during the month of April 2005 shows that there are 2687 structs, of which 189 contain bit-fields, of which only 15 are used by applications that transmit on the radio. Likewise, UCLA's CENS repository contains 3111 structs, of which 192 contain bit-fields, of which only 25 are used by applications to transmit on the radio.

4.3 Network Types Implementation

The nesC compiler compiles nesC programs to C, which are then passed to a native compilers such as `msp430-gcc` or `avr-gcc`. We considered two code-generation strategies: converting all network types to byte arrays, and preserving the structure of the network types in the generated C code.

Generating byte arrays for network types is done by replacing any use of a network type by a correspondingly-sized byte array. For instance:

```
SurgeCmdMsg x; --> char x[5];
```

and accesses to network types are replaced by appropriate array operations:

```
SurgeCmdMsg *pCmdMsg = (SurgeCmdMsg *)payload;
if (pCmdMsg->type == SURGE_TYPE_SETRATE)
  timer_rate = pCmdMsg->args.newrate;
```

becomes:

```
char *pCmdMsg = (char *)payload;
if (pCmdMsg[0] == SURGE_TYPE_SETRATE)
  timer_rate = NTOH32(&pCmdMsg[1]);
```

where `NTOH32` is a function to decode a 32-bit network int. Generating byte arrays for all network types ensures maximal portability as it does not depend on any assumptions about how the underlying C compiler represents types (see below). However, it requires that the nesC compiler replace all field accesses by offsets, replace uses of `offsetof`, and many other issues. In particular, C arrays behave differently from other C types, so all assignments, casts, pointers involving network types (note, e.g., the declaration of `pCmdMsg` in the example above) require special handling. We thus rejected

⁴For completeness, we also provide little-endian versions of the base types, named `nxle...`

this approach in favor of a type-structure-preserving transformation, described next.

The other approach is to replace (in the generated code) each network type with an “equivalent” C type, by translating `nx_struct` and `nx_union` to `struct` and `union` and defining the network base types as a `struct` containing a byte array. We can then do a much simpler translation of expressions involving network types:⁵

- If e is a read of a network base type t , replace e by `NTOHt(&e)`, where `NTOHt` is the function to decode a sequence of bytes representing network base type t .
- If $e_1 = e_2$ is a write to a network base type t , replace it by `HTONt(&e1, e2)`, where `HTONt` is the function to encode a value into a byte array for a network base type t .

For instance, our `Surge` example becomes:

```
typedef struct { char data[4]; } nx_uint32_t;
static inline unsigned short NTOUH32(void *target) {
    unsigned char *base = target;
    return (unsigned long)base[3] << 24 |
           (unsigned long)base[2] << 16 |
           base[1] << 8 | base[0];
}
...
typedef struct {
    nx_uint8_t type;
    union {
        nx_uint32_t newrate;
        nx_uint16_t focusaddr;
    } args;
} SurgeCmdMsg;
...
SurgeCmdMsg *pCmdMsg = (SurgeCmdMsg *)payload;
if (NTOUH8(&pCmdMsg->type) == SURGE_TYPE_SETRATE) {
    timer_rate = NTOUH32(&pCmdMsg->args.newrate);
}
...
```

For this translation to be correct, a structure containing only characters (such as `nx_uint32_t`) should have no alignment restrictions, and the same must hold for structures containing such structures (e.g., `SurgeCmdMsg`). This is true on many, but not all platforms: on ARM processors, all structures are aligned to 4-byte boundaries, and on Motorola 68K processors they are aligned to 2-byte boundaries. We currently work around this problem by using `gcc`’s non-standard `packed` attribute. A fully portable (to all compilers and platforms) implementation of network types would require implementing the byte-array approach sketched above.

5. EVALUATION

To validate our network types implementation we test interoperability between `micaz` and `telos` motes by modifying TinyOS and some TinyOS implementations to use network types. We report the size of these changes and measure their runtime overhead; furthermore, we compare the network types overhead with marshalling on `micaz` motes. Since the sensor network applications we observed are mostly idle and non time-critical, the changes in running time do not affect their behaviors.

⁵Read/write operators like `++`, `+=`, etc require a slightly more involved translation using a temporary variable.

Application	Mote	Changed Lines	Original	Network
TinyOS		33		
CntToRfm	micaz	3	9.5kB	9.6kB
	telos		12.6kB	12.8kB
RfmToLeds	micaz	3	9.1kB	9.1kB
	telos		12.3kB	12.4kB
OscilloscopeRF	micaz	11	11.3kB	11.4kB
	telos		16.6kB	16.8kB
Surge	micaz	10	15.5kB	16.0kB
	telos		20.6kB	21.2kB

Table 2: Source code changes and compiled code size when using network types (compiled with optimization on).

5.1 Interoperability

We successfully tested the interoperability between `telos` and `micaz` motes on a set of TinyOS applications converted to use network types. These applications are `CntToRfm`, `RfmToLeds`, `OscilloscopeRF` and `Surge`. `CntToRfm` is a straightforward application that sends an incrementing value on the radio, while `RfmToLeds` receives this value and displays it on the mote’s LEDs. `OscilloscopeRF` periodically senses the environment, then sends batches of values on the radio. `Surge` (discussed above) uses multihop routing; its use of TinyOS protocols is very similar to that of realistic sensor network applications such as `TinyDB` [20]. Our network consists of one `micaz` and one `telos` mote for `CntToRfm`, `RfmToLeds` and `OscilloscopeRF`, and of two `micaz`’s (one the root of the multi-hop tree) and one `telos` for `Surge`.

Table 2 summarizes the size of the changes to use network types, and their effect on program size. `Changed Lines` is the number of lines of code changed to use network types, `Original` is the compiled size of the original program and `Network` is the size of the program with network types. The `TinyOS` row presents the sizes of the changes to core TinyOS components, i.e., the basic `TinyOS TOSMsg` type and the `TinyOS MintRoute` multi-hop routing (in `tos/lib/MintRoute`) and flooding (`tos/lib/Broadcast`) libraries (used by `Surge`). Additionally, we eliminated some differences between the `micaz` and `telos` motes by deleting some components modified for the `micaz` platform (`FramerM` and `NoCrPacket`) and by using a common, network-type based definition for the basic `TinyOS TOSMsg` type. Our version of `Surge` is slightly modified from the `TinyOS` one so as to support `telos` motes; we do not include the size of these changes in `Changed Lines`.

The ROM increase for using the network types is small as shown in Table 2. The worse case is `Surge` where it is only 3%. The changed lines represent only changes to type declarations and uses, no logic changes were required in any of these applications.

5.2 Overhead

The runtime penalty for using network types is negligible, as long as we enable optimization when compiling the generated C code: Figure 2 shows the runtime of the four applications of Table 2 on `micaz` motes without using any optimization, and Figure 3 shows the same graph using optimization (`gcc`’s `-Os` optimization flag). The results are the sum of the times, in clock cycles, to prepare, send, receive and process a message⁶. Each application contains three bars, corresponding to the runtime of the original code, the network types code, and the first marshalling method of Section 4.1. The three sub-elements per bar correspond to the amount of time spent in different layers in the network stack — the actual

⁶Excluding lightweight services such as `leds`, `timers`, etc.

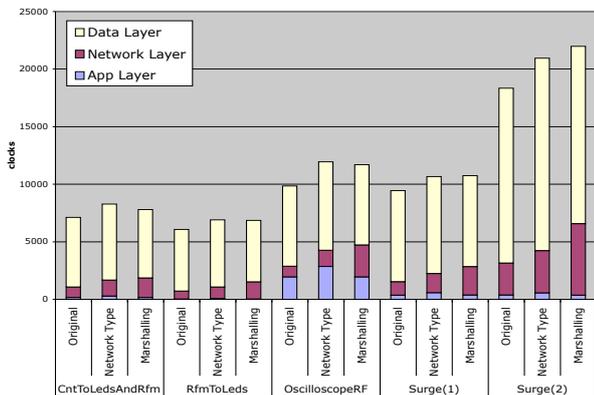


Figure 2: Runtime cost of network types and marshalling without using compiler optimization.

application, network,⁷ and data-link layers. Note that Surge(1) is a standalone Surge application whereas Surge(2) is a pair of Surge applications interacting with one another.

We obtain these results by manually modifying the applications to keep track of the runtime on different network layers on *micaz* motes. Runtime is measured as clock cycles by accessing the system clock and then averaging the results of 100 messages. The cost of accessing this system clock is low (a few cycles), so this instrumentation does not significantly affect the time spent in the network layers; furthermore, this overhead is identical for all measurements. Because Surge(2) has an initial setup time (to build its ad-hoc routing tree), its results are obtained after the system stabilizes.

As expected (Figure 2), all layers show small runtime increases for network types. For marshalling, the increase happens only in the network layer runtime as it contains the network-to-host and host-to-network encoding functions. For example, for unoptimized OscilloscopeRF, the application layer runtime for network types is 2850 cycles, and only 1940 cycles for both the original and marshalling cases. Conversely, the network layer takes 2770 cycles for marshalling, 1400 cycles for network types and 1000 cycles for the original code. Once we turn on optimization (Figure 3), we see that network types has less overhead than marshalling on *micaz* motes, and only a small overhead (the best case is Surge without any overhead, and the worst case is CntToLedsAndRfm, with 67 more cycles or 4.1%) over the original code. The *micaz* mote has only 8-bit loads, thus the differences between original and network type execution time reflect places where the C compiler compiles loads and shifts into less efficient code than multi-byte loads.

Note that in both graphs, the increase in both the application and network layer are negligible compared to the large number of cycles required for actual packet transmission in the data layer (The CC2420-based radio on the *micaz* takes about 1ms, i.e., 7000 cycles to send a 32-byte packet).

5.3 Surveys

We conducted an informal survey in UCLA’s CENS lab and found out that it is routine to use `gcc’s __attribute__((packed))` for all message types. While this has worked for all past platforms,

⁷We use the term loosely, to mean all packet processing between the data-link and application layers.

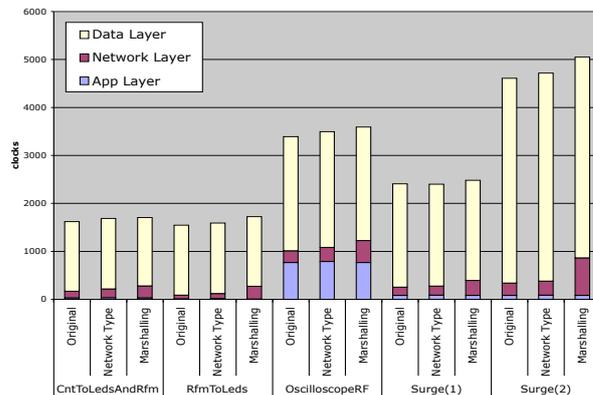


Figure 3: Runtime cost of network types and marshalling using compiler optimization.

compilation for new telos motes (whose msp430 processor requires 2-byte alignment for most types) fails because its native compiler, `msp430-gcc` does not fully support packing. The work around in CENS is to manually insert padding where necessary to guarantee interoperability.

The idea of using network types is thus very appealing because there are numerous legacy programs that would require manual and laborious porting to telos motes. In addition, future platforms may require additional efforts to port, e.g., for big-endian platforms. With network types, developers can “write once, run on any mote.”

6. CONCLUSION AND FUTURE WORK

Our solution to sensor network heterogeneity is simple, efficient, and easy to use. We designed and implemented network types, an extension to the nesC programming language to support types with a platform-independent representation. Unlike traditional approaches based on marshalling, network types let the programmer operate on packets in the network representation just like any other C type. This makes it easy to implement new, and modify existing TinyOS applications and protocols to interoperate on various hardware platforms. We found that network types made it possible to port existing TinyOS applications with very limited source code changes, needing minimal time and effort. Additionally, network types imposed no RAM, and little CPU, overhead.

In the future, we plan to address the current limitations of network types (mostly the lack of bitfield support), and investigate their use in other scenarios than sensor networks. If the CPU overhead of network types becomes an issue, we believe that there are several opportunities for optimizing network type accesses: avoiding unnecessary network/host and host/network conversions if the host representation is not used, avoiding repeated conversions on multiple reads of the same network type, temporarily storing network types in host order.

The upcoming version 2.0 of TinyOS will use network types; we hope that the wider TinyOS community will also adopt network types as a basis for platform-independent networking.

7. ACKNOWLEDGMENTS

We thank Jens Palsberg, Benjamin Greenstein, Phil Levis and Wei Hong for helpful comments on the design of network types.

8. REFERENCES

- [1] Jason L. Hill and David E. Culler, "Mica: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, 2002.
- [2] Mark Yarvis, Nandakishore Kushalnagar, Harkirat Singh, Anand Rangarajan, York Liu, and Suresh Singh, "Exploiting Heterogeneity in Sensor Networks," in *IEEE INFOCOM*, Mar. 2005.
- [3] Y. Ge, L. Lamont, and L. Villaseñor, "Improving Scalability of Heterogeneous Wireless Networks with Hierarchical OLSR," in *The OLSR Interop and Workshop*, Aug. 2004.
- [4] Matt Welsh Philip Levis, Nelson Lee and David Culler, "TOSSIM: Accurate and Scalable Simulation of Entire TinyOS Applications," in *In Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.
- [5] Lewis Girod, Jeremy Elson, Alberto Cerpa, Thanos Stathopoulos, Nithya Ramanathan, and Deborah Estrin, "EmStar: a Software Environment for Developing and Deploying Wireless Sensor Networks," in *Proceedings of the 2004 USENIX Technical Conference*, Boston, MA, 2004, USENIX, To appear.
- [6] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E. Culler, and Kristofer S. J. Pister, "System Architecture Directions for Networked Sensors," in *Architectural Support for Programming Languages and Operating Systems*, 2000, pp. 93–104, TinyOS is available at <http://www.tinyos.net>.
- [7] "Information technology – Abstract Syntax Notation One (ASN.1): Specification of basic notation," ISO/IEC Standard 8824-1:2002.
- [8] Sun Microsystems, Inc., "RFC 1014 - XDR: External Data Representation," 1987.
- [9] David Gay, Phil Levis, Rob von Behren, Matt Welsh, Eric Brewer, and David Culler, "The nesC language: A holistic approach to networked embedded systems," in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.
- [10] "Information technology – ASN.1 encoding rules: Specification of Basic Encoding Rules (BER), Canonical Encoding Rules (CER) and Distinguished Encoding Rules (DER)," ISO/IEC Standard 8825-1:2002.
- [11] "Information technology – ASN.1 encoding rules: Specification of Packed Encoding Rules (PER)," ISO/IEC Standard 8825-2:2002.
- [12] Todd Proebsting Sean O'Malley and Allen Brady Montz, "USC: A Universal Stub Compiler," in *Proceedings of the Conference on Communications Architectures, Protocols and Applications (SIGCOMM)*, Aug. 1994.
- [13] Eric Eide, Kevin Frei, Bryan Ford, Jay Lepreau, and Gary Lindstrom, "Flick: A flexible, optimizing idl compiler," in *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'97)*, June 1997.
- [14] Christian Huitema, "MACROS: Highlights of an ASN.1 Compiler," Tech. Rep. Internal working paper, INRIA Project RODEO, 1991.
- [15] J. Siegel, *CORBA 3: Fundamentals and Programming*, John Wiley and Sons, Inc, 2000, Second edition.
- [16] M. Kirtland, *Designing Component-Based Applications*, Microsoft Press, 1999.
- [17] "Information technology – Programming languages – Ada," ISO/IEC Standard 8652:1995.
- [18] Douglas E. Comer, *Network Systems Design using Network Processors*, Prentice Hall, 2004.
- [19] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler, "The emergence of networking abstractions and techniques in tinyOS," in *First Symposium on networked system design and implementation (NSDI04)*, San Francisco, California, USA, 2004, pp. 1–14.
- [20] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong, "Tinydb: An acquisitional query processing system for sensor networks," *Transactions on Database Systems (TODS)*, Mar. 2005.