Generic Multi–Packet Communication through Object Serialization

Leon Evers
Pervasive Systems Group
University of Twente,
the Netherlands
eversl@cs.utwente.nl

Maria Eva Lijding
Pervasive Systems Group
University of Twente,
the Netherlands
lijding@cs.utwente.nl

Jan Kuper
Embedded Systems Group
University of Twente,
the Netherlands
kuper@cs.utwente.nl

ABSTRACT

Wireless sensor networks communication protocols and abstractions have remained fairly simple until now, dealing only with payloads the size of individual network packets. A method to transparently communicate variably sized data in a platform-agnostic manner may ease building energy-efficient and robust applications.

This paper presents a communication abstraction that enables multi-packet communication, while minimizing memory requirements by using an object serialization mechanism to integrate memory management and communication functionalities. Evaluation shows reduction of communication of up to 3.5 times using our method compared to state of the art, and improvement of application performance through the use of reliable communication.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Performance, Reliability, Experimentation

Keywords

Wireless Sensor Networks; Transmission Protocol; Reliable Communication; Memory Management

1. INTRODUCTION

Wireless Sensor Networks have been around for a number of years now, and the software and tool support for programming them is expanding and maturing quickly. Surprisingly however, the software abstractions for communication have remained fairly basic. All that WSN programmers can use to communicate are individual byte packets to transfer single integers or small, flat structures. Given the importance of communication in WSN applications – it is practically the only method of data in– and output – this is truly a remarkable fact.

In Internet communication, we are used to communicating streams of data, automatically packaged into separate packets, using the TCP/IP protocol. More advanced services like serialization or marshaling directly transfer language-level objects, without needing to explicitly encode and decode those objects. Having a similar kind of communication service for WSN platforms could simplify writing applications, which are more portable across platforms using different packet sizes.

For the SensorScheme WSN platform [4], we have developed a communication mechanism called *ObjectStreams* that transfers collections of linked objects in a sequence of packets. Using ObjectStreams makes it straightforward to communicate payloads of any size, which is then transmitted in as many packets as needed.

In section 2 we describe the object model, the memory-efficient serialization mechanism, and multi-packet ordering and recombination technique used. ObjectStreams is a mechanism that can be used in conjunction with any of the wide variety of existing MAC, routing and transport protocols for WSNs. It is independent of the underlying protocols and packet payload size, and optionally supports reliable transport through the use of acknowledgments and retransmissions.

We evaluate the benefit for such applications in section 3 by taking a closer look at two example applications: neighborhood gossiping, and tree-routed data collection as used by for example TinyDB [8]). We analyze the communication efficiency and memory requirements for different implementations of the same application, one of which uses ObjectStreams.

We conclude this paper with a review of the state of the art in section 4, and state our conclusions in section 5.

2. OBJECTSTREAMS

ObjectStreams is a communication library that merges multi-packet streaming communication with object serialization in single communication protocol stack layer. Serialization is a well-known method used to transport language-level objects across networks, available in languages such as Java for use with RMI [10] , or known as marshalling in CORBA [13]. ObjectStreams was developed as a communication library for the SensorScheme [4] platform, but – as it is built on TinyOS [6]– its benefits are also applicable for use with TinyOS applications.

Unlike internet-scale serialization mechanisms that make

MidSens'08, December 1-5, 2008, Leuven, Belgium

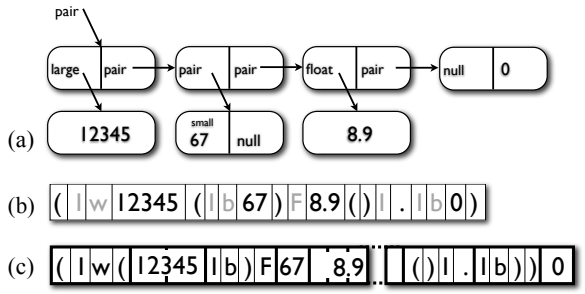

Figure 1: Encoding of an example message: (a) The in-memory layout; (b) tokens and bytes emitted during encoding; (c) message packed into two bytes.

use of the TCP/IP protocol for transmission and fragmentation, ObjectStreams is responsible for the serialization of payload data, fragmentation of messages, and recombination on the receiving side.

The ObjectStreams protocol layer positions itself between the network or routing layer and the application in a protocol stack.

Different routing protocols can serve as the lower layer to ObjectStreams, provided that they present a standard interface for sending and receiving of individual packets, such as the TinyOS `Send` and `Receive` interfaces. This allows application developers to use ObjectStreams with the transport protocol of their choosing or send ObjectStreams messages directly over the broadcast medium (using Active Messages on TinyOS) to direct neighbor nodes. If desired, it is possible to reliably transfer the sequence of packets comprising an ObjectStreams message, with the use of retransmission mechanisms that a transport protocol provides.

2.1 Object model

ObjectStreams uses a linked list data structure designed to minimize memory footprint. Objects of different kinds can be made to form compound data structures containing integer and floating point numbers. SObjects are a generic data type that can can contain four different kinds of data:

Integer a signed integer between -2^{31} and $2^{31}-1$;

Floating point a 32-bits floating point number;

Pair an object containing two references – *left* and *right* – to other objects;

Null a special object (with its own type) used to signify the end of a list.

While the first two kinds just contain a single number, the *pair* SObject kind is the constructive element for complex data structures, such as lists, associative lists or dictionaries and binary trees. Arrays and *struct*s (as they are called in the C programming language) – particularly relevant in the context of WSNs, because of their frequent use – can be represented by forming a linked list of *pairs* where the *left* references point to the elements inside the array or struct, and the *right* reference points to the rest of the list, or to the *null* object in the case of the last element of the list.

2.1.1 Implementation

To achieve acceptable memory use, minimizing the memory footprint of SObjects is crucial. Figure 1 (a) shows the in-memory layout of an example ObjectStreams message.

All SObjects are allocated from an allocation pool consisting of 4 byte allocation cells. Application developers using ObjectStreams may themselves choose the size of the allocation pool, but for maximum performance it is considered good practice to allocate as large a pool as possible, that is, all memory not allocated by other parts of the application should be available for allocation of SObject cells.

References to SObjects, which are the contents of *pair*s, occupy 16 bits. References, shown as arrows in figure 1 (a), contain a type tag (shown next to the arrow) of 2 bits in size, to distinguish the different kinds of SObjects from each other. The remaining 14 bits of the SObject reference contain the address of the SObject carrying the data, in the case of large integers, floats or pairs, stored in an allocation cell. References using the fourth type tag value are considered short integers: the address bits of the reference contain the numerical value, between -2^{13} and $2^{13}-1$. The majority of integer numbers in a program are within this range, and the use of short integers eliminates the need for a separate cell to store the number. Finally, the special-purpose *null* SObject does not need a storage location. Only tests for equality are performed on the null SObject, so its value is a single address that is outside of the range of available allocation cells.

The ObjectStreams library provides the necessary operations to allocate, access and deallocate SObjects, and provides a number of conversion routines, to make it easy and straightforward to use SObjects in WSN application code witen in NesC.

2.2 Serialization

During communication, ObjectStreams serializes SObjects as a sequence of bytes, that – stored into one or more packets – travel to a receiver, which reconstructs the message as a collection of SObjects in memory. Packets are filled to their maximum payload size, or with as many bytes as are left at the end of the sequence.

The method of encoding SObjects, shown in figure 1 (b) is similar to the Lisp [9] notation of linked list structures (in figure 1 (b), the black tokens): a list is surrounded by opening and closing tokens (parentheses), with inbetween the elements of the list, which may themselves be lists surrounded by a pair of opening and closing tokens, or integer or floating point numbers. Instead of characters separated by whitespace, ObjectStreams uses a compressing encoding, emitting tokens and single bytes according to the algorithm in figure 2.

The algorithm in figure 2 uses pattern matching on the type of the SObject to define the operations performed in the main encoding routine – `encode` – and the auxilary procedures `enc-int` and `enc-list`. Tokens encode which one of four possible types of data is encoded next, after which the actual data follows. At any moment during encoding, only four possible tokens are permitted, from either the `ObjToken` or `IntToken` data types defined in figure 2. Figure 1 (a) shows the result of applying this algorithm to the example message in figure 1 (a). Tokens are encoded using 2 bits, and four subsequent tokens are put together in a single byte. Figure 1 (c) shows the result of serializing the example message

```
data SObject  = Pair SObject SObject | Int num
              | Float float | Null;
data ObjToken = TStart | TEnd | TInt
              | TFloat;              // '(', ')', 'I', 'F' in fig 1
data IntToken = TByte | TWord | TLong
              | TDot;                // 'b, 'w', 'l', '.' in fig 1

encode (Null)     = emit (TStart), emit (TEnd);
encode (Pair l r) = emit (TStart), encode (l), enc-list (r);
encode (Int n)    = emit (TInt), enc-int (n);
encode (Float f)  = emit (TFloat), emit-bytes (f, 4);

enc-list (Null)     = emit (TEnd);
enc-list (Pair l r) = encode (l), enc-list (r);
enc-list (s)        = emit (Int), emit (TDot), encode (s);

enc-int (-2^7 ≤ n < 2^7)   = emit (TByte), emit-bytes (n, 1);
enc-int (-2^15 ≤ n < 2^15) = emit (TWord), emit-bytes (n, 2);
enc-int (-2^31 ≤ n < 2^31) = emit (TLong), emit-bytes (n, 4);
```

Figure 2: Algorithm for encoding of SObjects into a token sequence.

into a sequence of bytes stored into two packets.

Encoding of SObject structures and filling packets proceeds in a streaming fashion: The SObject is traversed and tokens emitted until the packet's payload section is filled, or the entire SObject structure is encoded. When a packet is filled before the entire structure is traversed, the current state of encoding is stored in a sequence of linked SObjects, to be used again when encoding can resume. Subsequently, when the next packet will be filled, the encoding state is retrieved, and the encoding process continues, storing emitted tokens in the new packet.

Similarly, receiving nodes decode the contents of a packet sequence on a per-packet basis. During decoding, tokens are read from the packet payload, and the encoded SObject structure is created, allocating SObjects as needed. When a message consists of multiple packets, at the end of each packet (except for the last packet of the sequence), the current decoding state is stored in a number of allocation cells, which are stored away to be retrieved when the next packet in the sequence arrives. When the last packet of the sequence is received, ObjectStreams returns to the application a reference to the SObject structure contained in the packets. In case not all packets in the sequence are received, the SObjects received are deallocated, and not returned to the application.

2.3 Packet Sequencing

ObjectStreams carries its payload in a sequence of packets. Receivers of this packet sequence must be able to recognize all packets belonging to the same sequence, and process them one by one, in order, without missing ones or duplicates. Additionally, nodes may receive multiple streams simultaneously, possibly from the same sender. The ObjectStreams protocol defines a packet header format designed to take care of all of this, while being compact enough to induce only marginal overhead. The encoded SObject structure can have arbitrary complexity and size, and its size can be calculated only by traversing the entire structure. The encoded message therefore does not contain information regarding the total number of packets it consists of, but rather is a stream of packets, marking only its start and end.

ObjectStreams payload is carried in a stream of packets, each consisting of a small header, and a payload field to carry the stream content. ObjectStreams defines three header fields: a start flag a, a stop flag z and a sequence number k – implemented as a fixed-bit-width number of b bits. The start flag marks the start of a stream, and is set only on the first packet of the stream. Similarly, the end flag is set on the last packet of a stream. The sequence number is used to track the order of packets in a stream. Every subsequent packet in the stream contains a sequence number $k_{i+1} = k_i + 1 \bmod 2^b$, until the last packet, which is marked with the stop flag.

The tuple $\langle$ sequence number, source address $\rangle$, called a *packet ID*, uniquely defines every individual packet. Therefore, no two packets with the same packet ID should be alive in the network at any moment. When a packet is received and processed at its destination it is no longer active and a packet with the same packet ID is permitted again. Packets can remain active for a longer duration because of buffering or caching at either the source or destination node, or on any intermediate node in the case of multi-hop link models.

Multiple streams between the same sender and receiver can be active at any one time, alternating packet transmissions. Streams are numbered S^0, S^1 ... S^n, each of which consists of packets p_0, p_1 ... p_m. To minimize the probability of duplicate packet IDs active in the network, the sequence number of a stream's first packet S^{n+1}_{p0} is initialized to the sequence number of the most recently sent packet from the stream last created at the same node – S^n_{pm} – minus a number $k(\bmod 2^b)$, where k is greater than the number of packets the network can buffer: $\text{seqno}(S^{n+1}_{p0} = \text{seqno}(S^n_{pm})k \bmod 2^b)$.

3. EVALUATION

In this section we investigate the consequences of replacing well-known existing WSN application protocols with an implementation based on ObjectStreams: a 2-hop neighborhood gossip protocol, and a tree-routed data collection application. Our main objective is to expose the differences in communication and memory needs between the alteratives evaluated. Besides, as the compared implementations display somewhat different behavior, we look at the impact of alternative implementations on application performance.

3.1 Simulation and communication modeling

We evaluate our protocols using a simulated network of nodes that run the algorithms described below. We model the network as a fully connected graph, with directional edges, labeled with P_{ij}, the probability of reception between any two nodes, that ranges from 100 % for nearby nodes to around zero for the most separated nodes. The reception probability P_{ij} between each pair of nodes are drawn from a Gaussian random distribution, the mean and variance of which are a function of the distance between the nodes. The distance-related distribution parameters are obtained from an empirical study performed by Ganesan *et al.* [5], also used in the TOSSIM simulation framework.

The network consists of a large number of nodes, placed randomly on a square area of varying sizes. The square area 'wraps around' at the top and bottom and the sides, making nodes at opposite edges each others' neighbors. When the

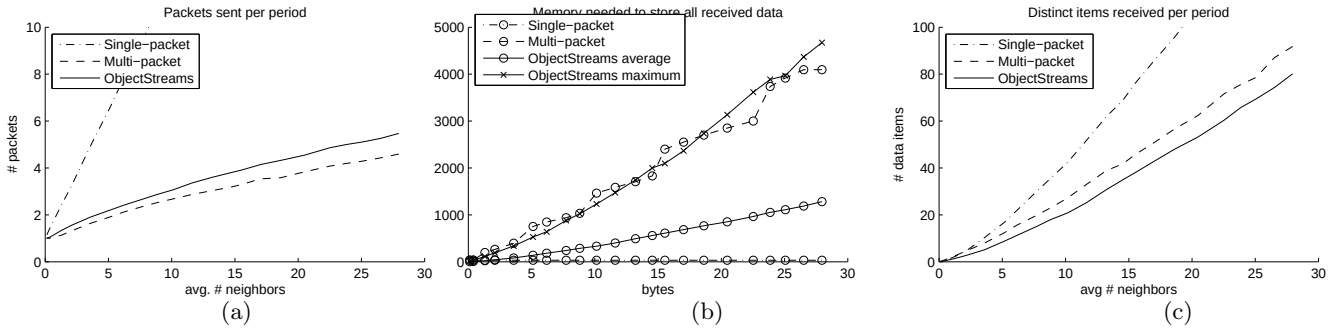

Figure 3: Performance parameters of broadcast protocol evaluation.

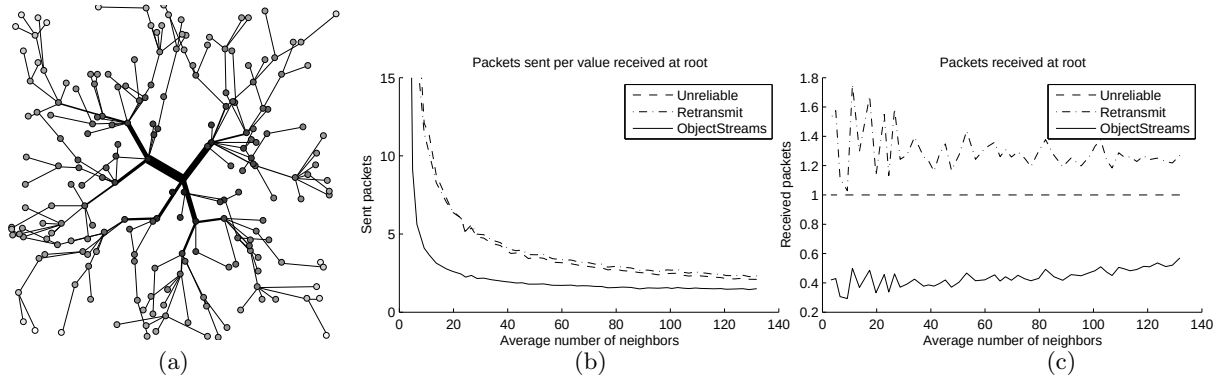

Figure 4: Performance parameters of the tree routing protocol evaluation.

transmission range is much smaller than the area dimensions, the 'wrapped' network mimics a network of infinite size. We use this method to eliminate the effects of nodes at the edges with reduced connectivity, and make the performance characteristics independent of network size. In all of the studies described we have experimented with networks of different numbers of nodes, and found the results to be independent of network size. We consistently have used large networks of 700 nodes, to reduce the influence of random effects, but the results are representative of smaller networks as well.

The parameter we vary in the studies is the area size in which nodes are placed. As the transmission range of the nodes remains equal, this has the effect of increasing connectivity as the area size decreases. We measure the *connectivity* as the number of other nodes that each node is le to receive from with probability of 50 % or greater. The connectivity, averaged over all nodes, is displayed on the horizontal axes of the graphs in figures 3 and 4.

Furthermore we measure only packet transmission and reception occurrences. Communication is assumed instantaneous and packet collisions do not occur. These assumptions accurately mimic actual implementations with very infrequent transmissions, short packet transmission times and a randomly chosen moment of transmission within each period. The goal of avoiding packet collisions coincides with our goal of reducing communication: reduction of communication results in a proportional reduction of packet loss, and increased communication results in an increase in packet loss due to collisions.

3.2 Two-hop gossip protocol

As our first example we take a gossip protocol where every node gathers sensor data from its two–hop neighborhood (that is, the node's neighbors and its neighbors' neighbors).

The protocol operates in rounds of fixed duration. Every round, nodes broadcast their sensor data, and received data from its neighbors. The next round, nodes broadcast all data received from its neighbors alongside their own new sensor data, and receive all second–hop–neighbor data sent by its neighbors. During each round, nodes make their calculation using all (first and second hop) data received. Note that multiple instances of second hop neighbor data might be received, while only a single instance is needed in the calculation.

The data items that nodes communicate are small – a sensor reading, a node ID, and two coordinates in the case of an object tracking application – and usually smaller than the payload of sensor network platforms (28 bytes for the Mica 2 motes using TinyOS [6]). During every round, nodes transmit multiple data items, both from themselves and forwarded data items from their neighbors.

We compare the performance of ObjectStreams to two alternative implementations, both of which use existing WSN communication protocols and abstractions, modified to execute this protocol more efficiently. The first (called *single-packet*) is an implementation of the protocol that transports a single measurement per packet, as is done in the abstract regions [15] implementation.

The second (named multi-packet) represents the lower bound on communication requirements. It operates similar to the ObjectStreams implementation, sending a single multi-packet message each timer interval, but uses fixed-sized statically allocated message buffers to store data re-

ceived from first-hop neighbors. Both the size of these memory buffers, and the number of buffers needed depends on the number of neighbors. The need to statically allocate a large number of these memory buffers results in large memory requirements for this implementation. All three implementations send data items containing three 16-bit sensor values and a 16 bit network address, and use packets with 28 bytes of payload.

Figure 3 shows the evaluation results of these three protocol implementations for varying connectivity rates. Figure 3 (a) shows the number of packets sent per period. For the more dense networks, the single-packet implementation quickly becomes very costly in terms of communication needs. ObjectStreams can contain about 2 $\frac{2}{3}$ items in a single packet, which reduces communication considerably. The multi-packet implementation is even more efficient, packing 3.5 data items per packet.

Figure 3 (b) shows the memory requirements of the three implementations. *Single-packet* uses no additional packet buffers to hold received data, so its memory use is a small, fixed amount. The multi-packet version uses multiple packet buffers to receive and store received neighbor data items. The amount of memory that needs to be allocated for packet buffers, shown in the graph, is the minimum number of bytes needed to receive messages from its neighbors for 95 % of the nodes. The graph shows two data sets for the ObjectStreams version: a maximum and an average amount. The maximum is the amount needed to store into SObject cells the equivalent of all multi-packet buffers. This is a theoretical maximum amount of memory, that is never fully used in any of the nodes participating in the evaluation . The (averaged) memory allocated by nodes during the simulation is shown as the 'average' result set.

The ObjectStreams and multi-packet implementations receive fewer – about 75 % to 50 % – second-hop data items, as shown in figure 3 (c), which translates into reduced application performance. This reduced performance occurs especially in higher-densty networks, when the loss of some data is less problematic.

From these results we can conclude that the three implementations behave quite differently: While the single-packet implementation has the lowest memory footprint and achieves the best application performance (in terms of the number of distinct data items received), the multi-packet implementation is the most energy-efficient as far as communication is considered, at the cost of considerable memory requirements. Inbetween these two, ObjectStreams operates energy-efficiently, while keeping memory consumption low.

3.3 Tree–routed data collection

The second application we evaluate concerns the collection of sensor data, and routing it down a tree rooted at a gateway node. TinyDB [8] is the prime example of using this method. When the complete data set is required, as opposed to only a summary, such as the average vale, intermediate nodes need to forward all individual data values received from nodes higher up in the tree. In current implementations such as TinyDB, every individual sensor data item is transmitted in a separate packet.

We we evaluate ObjectStreams by comparing it to two alternative implementations. Both alternative implementations use a single packet for every sampled data item, similar to TinyDB. The first alternative – called *unreliable* – transports the sensor data in a best-effort manner, without the use of acknowledgements and retransmissions. The other implementation, called *retransmit* uses acknowledgements and retransmissions as does the ObjectStreams implementation to achieve a near 100 % delivery ratio of measurements to the root: Parent nodes will send an acknowledgement upon reception of packets containing sensor data items. When the sender does not receive the acknowledgement, it retransmits the packet up to 5 times, and reselects a parent if the fifth retransmission was not successful either, after which transmission restarts. All three protocols use the TinyOS 2 Collection protocol [12] to create and maintain the routing tree. The communication involved in tree construction and maintenance is not included in the measurements reported here.

Our performance measurements count only the data items successfully delivered at the collection root, rather than the number of measurements sent from the leaves. In the case of the *unreliable* implementation, each sensor data packet is sent to the node's parent node in a best–effort fashion, without retransmissions in the case of packet loss. The result is that only a fraction of the sensor data is actually delivered to the routing tree, and the probability of delivery is not equal for all nodes, but depends on the position in the routing tree. For nodes high up in the routing tree, delivery of its data takes many individual transmissions, each of which may fail. To graphically indicate this, figure 4 (a) shows a randomly generated routing tree configuration for a sensor network of 200 nodes. The circles show the locations of the nodes, and the lines between them indicate a parent link for each node. The tree root is a node in the center of the figure (hidden behind the thick lines). The shading of the circles indicate the end-to-end delivery probability of each node's sensor data, ranging from as low as 11 % (in white) in the outer corners of the network to 100 % for nodes near the root (in black). The line thickness of the links connecting nodes to their parents are proportional to the amount of data traveling across it. The connections near the base of the network carry a heavy load, up to as much as 73 sensor data records every interval.

Figure 4 (b) displays the number of packets sent per value arriving at the collection root. Interestingly, *unreliable* and *retransmit* perform practically identical. ObjectStreams requires considerably less communication because it is able to transport about 3.5 sensor measurements in each packet. For sparser networks, where nodes are at greater distances from each other, the tree construction algorithm creates deep trees, consising of connections to a parent node that have a high probability of failing. Denser networks need less communication, as a result of lower packet loss rates and reduced tree depth.

Figure 4 (c) displays the number of packet transmissions to the root per received data value. For the *unreliable* implementation, this metric is one by definition, since one packet per data item is used, and no retransmission occur. The impact of retransmissions for a *retransmit* implementation is a higher communication load for the root node, since possible retransmissions take place. In the case of ObjectStreams, in the last hop to the root all sensor values generated in the subtree below the sender are put into a multipacket message to the root, thus significantly reducing communication.

4. RELATED WORK

This work aims to offer some of the advantages of TCP/IP to WSN platforms. As mentioned earlier, μIP [3] is an IP stack for WSN platforms. The TCP protocol is, however, not very suitable for the bulk of WSN communication.

Communication of data structures through serialization is a well-known and often-used technique, and lay at the basis of technologies such as Java RMI [10] and CORBA [13]. These serialization mechanisms are not designed for the small devices used in wireless sensor networks, and cannot be used. Besides ObjectStreams, serialization mechanisms targeted specifically at WSNs have not been developed until now.

One aspect or our work, to chain together a number of packets and deliver them reliably and in order has been addressed by other published protocols. The Rudolph stack layers, part of the Rime [1] protocol stack for the Contiki [2] WSN operating system, can be used to communicate payloads larger than a single physical packet. Its use is restricted, however, to sending or receiving a single multipacket message at a time, since only a single message buffer is available in memory. This limitation reduces its usefulness, and reveals the memory-related difficulties for implementing practical multi-packet transmission protocols. In the Rime protocol stack, multiple large message buffers would occupy too much memory.

In different ways, attention has been given to reliable transmission of multi-packet messages. The S-MAC protocol [16], for example includes provisions for reliably sending a burst of packets using acknowledges and retransmissions. Such functionality is more often part of transport protocols, of which Wang *et al.* present a survey [14]. Many of these protocols can serve as a transport protocol for ObjectStreams, while some handle reliable transport of sequences of packets itself, as does the Flush protocol [7].

5. CONCLUSION

ObjectStreams uses object serialization to enable multipacket communication by combining the communication protocol with a memory management strategy. Together it is possible to efficiently transmit messages consisting of multiple packets, and receive multiple such messages concurrently.

We evaluated the performance of two applications using ObjectStreams compared to alternative implementations. The two-hop gossip application evaluation shows that ObjectStreams provides a new trade-off between memory use, communication cost and application performance. ObjectStreams makes it possible to reduce communication to near the theoretical minimum, with only a small reduction in application performance and while keeping memory use low. Evaluation of the the tree collection protocol reveals a significant reduction in communication of over 3 times, while the use of a reliable transmission protocol can ensure lossless delivery of all sensor data to the collection root.

6. REFERENCES

[1] A. Dunkels. Rime — a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, Jan. 2007.

[2] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networkedsensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, Nov. 2004.

[3] A. Dunkels, T. Voigt, J. Alonso, H. Ritter, and J. Schiller. Connecting Wireless Sensornets with TCP/IP Networks. In *Proceedings of the Second International Conference on Wired/Wireless Internet Communications (WWIC2004)*, Frankfurt (Oder), Germany, Feb. 2004. (C) Copyright 2004 Springer Verlag. http://www.springer.de/comp/lncs/index.html.

[4] L. Evers, P. J. M. Havinga, J. Kuper, M. E. M. Lijding, and N. Meratnia. Sensorscheme: Supply chain management automation using wireless sensor networks. In *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2007, Patras, Greece*, pages 448–455, Los Alamitos, September 2007. IEEE Computer Society Press.

[5] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report RB-TR-02-003, Intel Research, march 2002.

[6] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E.Culler, and K. S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[7] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. E. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: a reliable bulk transport protocol for multihop wireless networks. In *SenSys*, pages 351–365, 2007.

[8] S. Madden, M. J. Franklin, J. M. Hellerstein, and WeiHong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conferenceon Management of data*, pages 491–502, New York, NY, USA, 2003. ACM Press.

[9] J. McCarthy. *LISP 1.5 Programmer's Manual*. The MIT Press, 1962.

[10] S. Microsystems and Inc. *Java Remote Method Invocation Specification*, 1997.

[11] Network Working Group. Tep 123: Collection tree protocol (ctp), 2007.

[12] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, (14):46–55, 1997.

[13] C. Wang, M. Daneshmand, B. Li, and K. Sohraby. A survey of transport protocols for wireless sensor networks. *IEEE Network Magazine Special Issue on Wireless Sensor Networking*, 20(Issue 3):34 – 40, May-June 2006.

[14] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation(NSDI '04)*, Mar. 2004.

[15] W. Ye, J. Heidemann, and D. Estrin. An energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the IEEE Infocom*, pages 1567–1576, New York, NY, USA, June 2002. USC/Information Sciences Institute, IEEE.