SPLIMER

Splitter & Merger for TinyOS

Semester project by Marco De Santis
for
DCG (Distributed Computing Group) at ETH Zurich

Patron: Prof. R. Wattenhofer
Assistants: Pascal von Rickenbach, Yves Weber

Contents

1 Motivation and Goal of the Project

TinyOS is a widespread operating system for sensor nodes that are able to form an ad-hoc network. The existent communication layer commonly used for sending data from one node to another node restricts the payload size per data packet to a fixed number of bytes. If an application running on a node wants to transmit data that exceeds the payload size limitation, the application itself must be equipped with a split/merge feature to be able to split the data, transmit it as several standard sized packets and reassemble the data on the receiver's side. This functionality cannot be regarded as a task that should be implemented by every application that needs to transmit larger amounts of data.
Hence the split/merge task should be fulfilled by an additional layer that is inserted between the user application and the communication layer. This additional layer is not only capable of splitting/merging large payloads but it also stores small payload data and places it in a normal sized message until the no more small messages can be fitted in it. The layer then sends the packed packet to the receiver where all the small payload parts are extracted from the incoming message. This pack/unpack feature enhances the network traffic and improves the ratio between payload size and message header.

The goal of this semester project is to implement the described layer for TinyOS extending it's capabilities to support sending of data packets of arbitrary size.

2 Current State & Planned Work

Before knowing what exactly has to be implemented and how it can be embedded in the existing set of modules, we need to assess the current state of TinyOS. By examining the system carefully we can a priori rule out some ideas of how to achieve the given task.

2.1 TinyOS' Current State

The standard communication stack of TinyOS is called AM as an abbreviation for active message. The active message header file (AM.h), contained in the TinyOS installation package, declares a variable(TOSH_DATA_LENGTH) defining the maximum payload that may be used with active messages (see Figure 1). The default maximum message payload is 29 bytes[1].

```
#ifndef TOSH_DATA_LENGTH
#define TOSH_DATA_LENGTH 29
#endif
```

Figure 1: how TOSH_DATA_LENGTH is defined

Although it might seem as if TOSH_DATA_LENGTH was chosen arbitrarily, it is not. The active message format itself would support much larger packets. This value is accepted as the best tradeoff between faulty transmissions due to bit errors and having a high data overhead.[2]
The longer the message is, the higher is the probability of at least one bit getting flipped during transmission, hence the received faulty message will be discarded on the receiving side. Another problem arises when changing a fundamental constant of TinyOS: A node having assigned a higher value to TOSH_DATA_LENGTH and therefore sending larger messages, is doomed to crash any other node (having the default value as TOSH_DATA_LENGTH) receiving and trying to process the messages.
The shorter the payload is chosen, the bigger the message header overhead gets. Extreme cases can manifest a message containing a header of 7 bytes and a payload of 1 byte which corresponds to an data overhead of 87.5%.

2.2 Planned Improvements

We conclude from chapter 2.1 that although altering the TOSH_DATA_LENGTH variable is possible, it directly leads to negative impacts. Therefore an alternative solution has to be found. Addressing the data overhead problem arising when sending small amounts of data: the solution to this problem is to pack several small messages into one standard sized message.
In order not to cause any other node to crash when sending large messages, we split large amounts of data to fit into several normal sized TOS messages.
Of course should the same split/pack layer be able - upon receipt of split/packed messages - to automatically reconstruct the original data and hand it to the user application.

[1] 29 bytes in TinyOS 1.x, 28 bytes in versions 2.x
[2] according to Michael Schippling
[http://mail.millennium.berkeley.edu/pipermail/tinyos-help/2006-October/020136.html]

3 Implementing Splimer

The implementation of Splimer may be subdivided in a sequence of work steps that lead up to the final product. The location of the Splimer layer has to be defined first. What follows is the definition of the used message formats and finally the coding of the Splimer layer.

3.1 Basic Concept of the Splimer Module

The Splimer module should be wired between the user application and the GenericComm module. GenericComm is a module that organizes transmissions (sending and receiving) of packets via radio and UART.

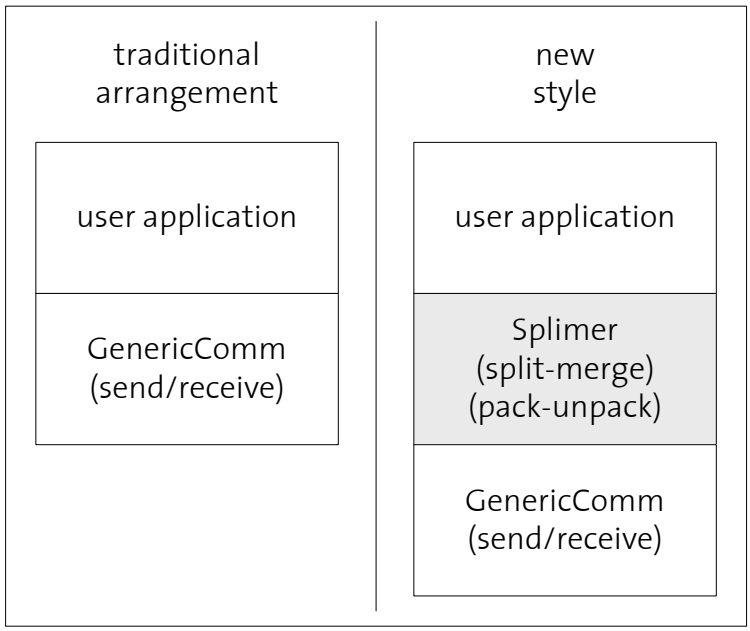


Figure 2: basic wiring concept

Figure 2 illustrates how the Splimer module should be arranged in respect of the user application and the GenericComm module. Figure 3 shows the transparency of the Splimer module. The user does not know (and cannot choose) whether a message is being split, packed or sent as a single message.
Furthermore we aim to provide and use a similar interface a user application has to use/provide when working directly with the GenericComm module. Doing so facilitates the use of Splimer.

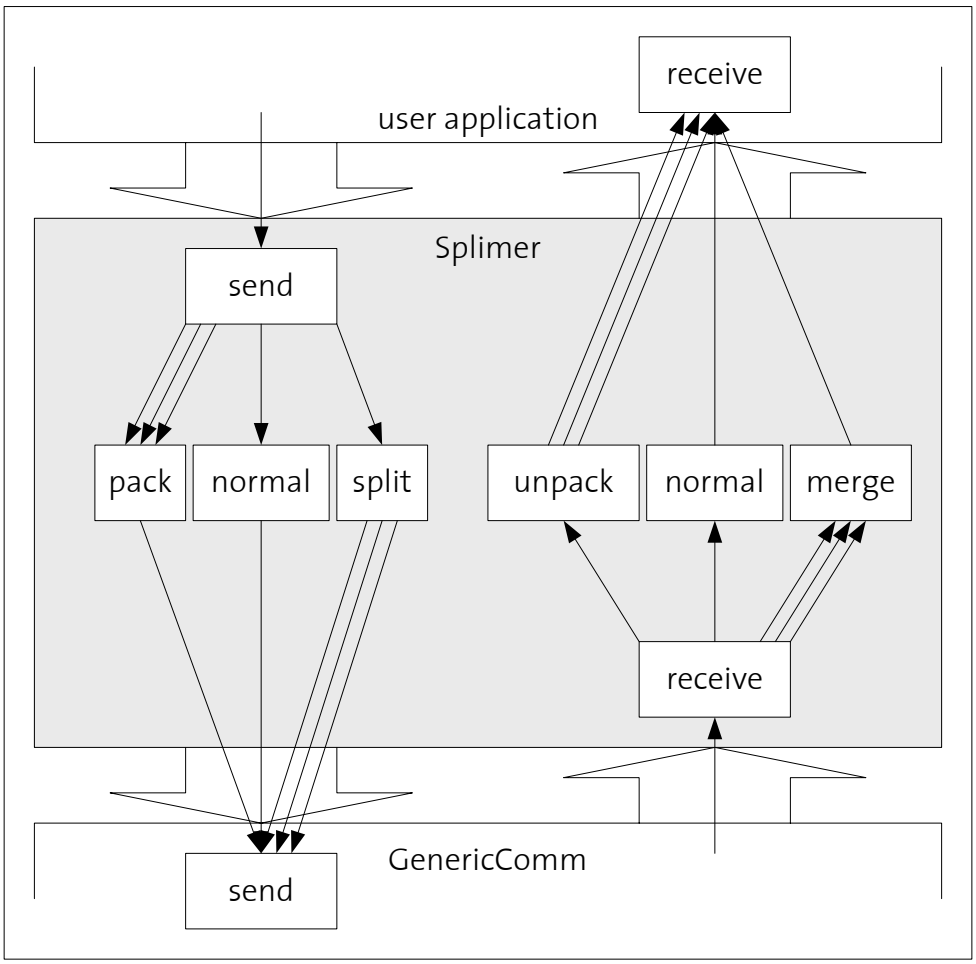
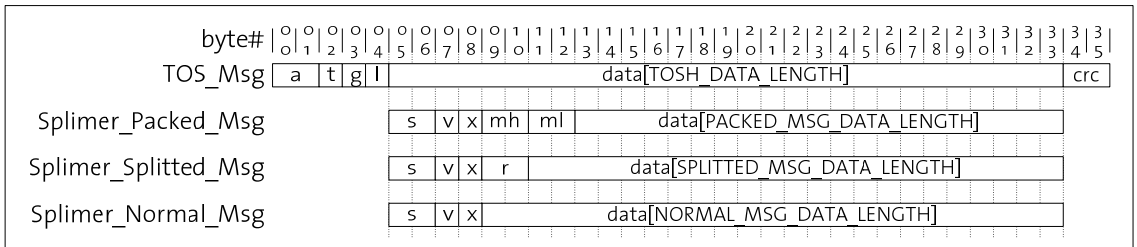


Figure 3: transparency of Splimer

3.2 Message Formats

One of the first steps towards the realization of the Splimer module is to determine what message types will be needed and what the contents (fields) of these messages will be.

During the development of Splimer it turned out that the initially proposed message formats lacked some information, therefore the formats had to be adapted. The final message formats are shown in Figure 4.



symbol	field name	description
a	addr	receiver's address
t	type	denoting the type of the packet
g	group	denoting the owner group (automatically set)
l	length	size of data wrapped by this header
crc	crc	cyclic redundancy code (automatically calculated and set)
s	sender	sender's address
v	datasize	size of data wrapped by this header
x	transactionID	which transaction is this packet a part of
r	dataLeft	data size of this data that is still waiting to be sent (splitted packets only)
mh, ml	splitMask	bit mask indicating how to split the msg into its original parts (packed packets only); mh contains the most significant bits of the mask, ml contains the least significant bits

Figure 4: message formats

TOS_Msg is the standard AM data format used to transmit a message. As described earlier it can hold the maximal number of bytes defined by TOSH_DATA_LENGTH. The custom message format to be used in the Splimer module therefore has to fit into the data section of the TOS_Msg data structure.

Trying to keep the number of different message formats used in Splimer to a minimum and the actual payload size as large as possible, three different message types were defined: one for splitted data messages, one format for packed data messages, and one for messages that neither need to be split nor merged.

3.2.1 Packed Message Format (Splimer_Packed_Msg)

If the data the user application wants to send is smaller than PACKED_MSG_DATA_LENGTH (see Figure 4 and Figure 5) Splimer attempts to insert it into a packed message. Figure 5 shows the layout of this message type.

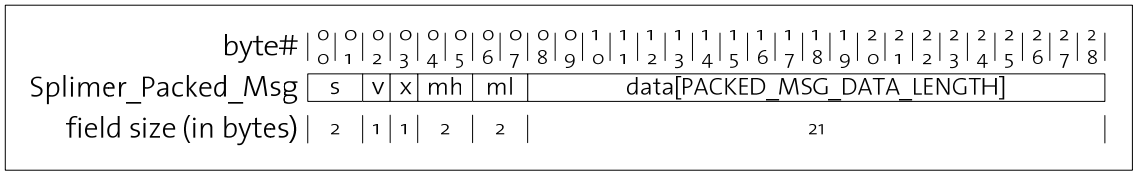


Figure 5: message format for packed data

Since the TOS message header format, which is the wrapper for the custom message formats, does not contain any clue about the sender of the message, this information (namely the sender's address) is the first field (denoted by 's') of all custom message formats. The second field ('v') contains the size of the payload. The third field ('x') denotes the transaction ID (see 3.2.2 for more details). What follows is the most important information for this type of message, the split mask (composed of the fields 'mh' and 'ml') which tells us how the payload has to be split in order to get the original data packets. This mask is a simple bitmask where

a bit set to 1 denotes the start of a new data packet in the payload section. For the sake of simplicity the last data packet is also terminated by a 1 in the split mask. The mask's size is 4 byte (32 bits)[3] which limits the payload section to a maximum of 31 bytes (remember the terminal 1 at the end of the last data packet), which in turn limits the TOS_DATA_LENGTH variable (see 2.1 on page 4) to 39 bytes (31 bytes payload + header size).

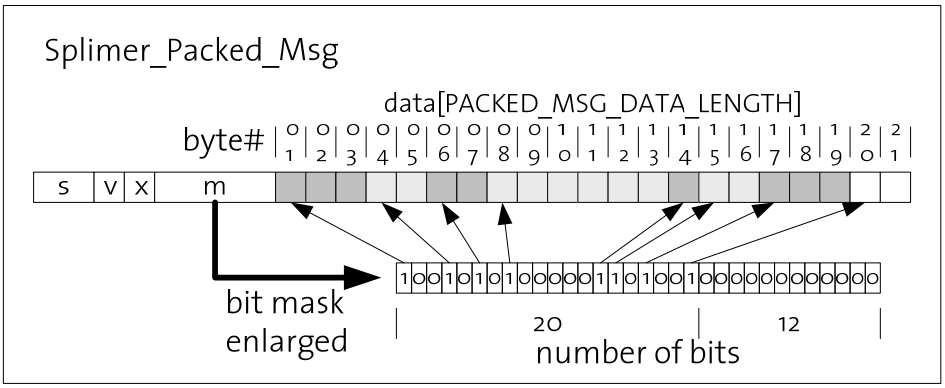


Figure 6: how the bit mask works

Figure 6 shows an example of a packed message. Its payload consists of 7 small parts. Although the packet can hold 21 bytes as payload, only 19 bytes are used. When examining the content of the bit mask, we see what was described above:
Each bit in the mask corresponds to one byte of the payload. A bit set to one means that the corresponding byte in the payload is the start of a new data part. The final byte of the last data part in the packet is indicated by a bit set to 1 in the bit of the bit mask representing the byte following the final byte of the last data part.

3.2.2 Split Message Format (Splimer_Splitted_Msg)

If the length of the data to be sent exceeds the TOSH_DATA_LENGTH, it is split and sent as several packets to the recipient who upon receipt reconstructs the data. Figure 7 shows of what fields this message type consists.

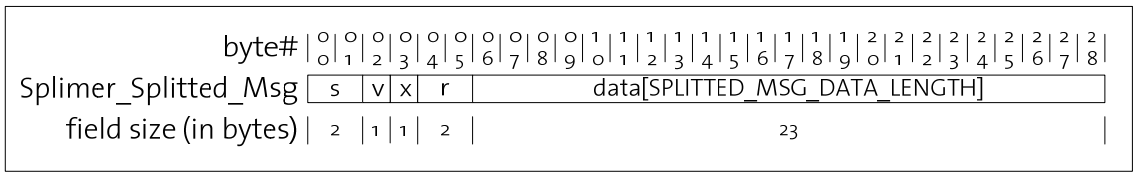


Figure 7: message format for split data

The first three fields(s, v and x) correspond to the packed message layout.
Our focus here is on the transaction ID (denoted by 'x', initiated as random integer) and the 'r' field declaring the number of bytes that are still waiting to be sent. The transaction ID (in combination with the sender's address) allows to identify the incoming message as part of the message that is being assembled at

[3] Due to problems experienced when bit shift operations are performed on 32bit unsigned integers on Tinynode, the originally 32bit field was split into two 16bit fields. For clarification: when referring to 'the mask', the underlying structure (two 16 bit unsigned integers) is ignored and regarded as one 32bit unsigned integer.

the receiver. The field ('r') containing the amount of data that is still on its way allows detection of missing/lost data fragments and it also flags the last packet of the split data (namely if this field is zero). After thoroughly thinking about the appropriate data type for this field, it was set to an unsigned integer represented by 16bit (i.e. a uint16_t type), allowing pending data sizes up to 2^{16}-1= 65535 bytes which correspond to approximately 64 kBytes.
A problem is how to know if the first message received really is the first part of the data (if the first part was not received). The current message layout does not provide any means for flagging or recognizing it.

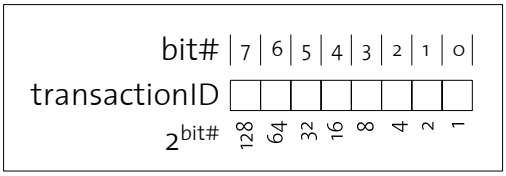


Figure 8: the transactionID field

To resolve the issue about recognizing the first data part, the decision was made to use one bit of the existing transactionID to flag the first data part. The most significant bit (bit number 7, see Figure 8) is used as flag. The consequences hereof are that the transactionID itself is reduced to a range of 0 to 127 which - what the next paragraph will show - is still acceptable.

On the receipient's side a data fragment is only accepted if it's sent by the same sender, with the same transactionID and the dataLeft fields correspond to what's expected to arrive next. Consider a setup where a message is split into *X* parts, the transactionID can take *T* different values and the probability that one message is lost is *p*. Every *X*T* packets the conditions mentioned above are met and a packet may be falsely accepted as missing part on the receivers side. Therefore *X*T* consecutive packets have to get lost. The probability of that to happen is $p_{falseAccept} = p^{X*T} * (1-p)$. It can be seen (Figure 9 & Figure 10) that even if there are only 2 parts sent (worst case), having a transactionID of cardinality 128, the maximal $p_{falseAccept}$ is very small (0.14%). To maximize $p_{falseAccept}$, the message loss probability has to very high (~99%).

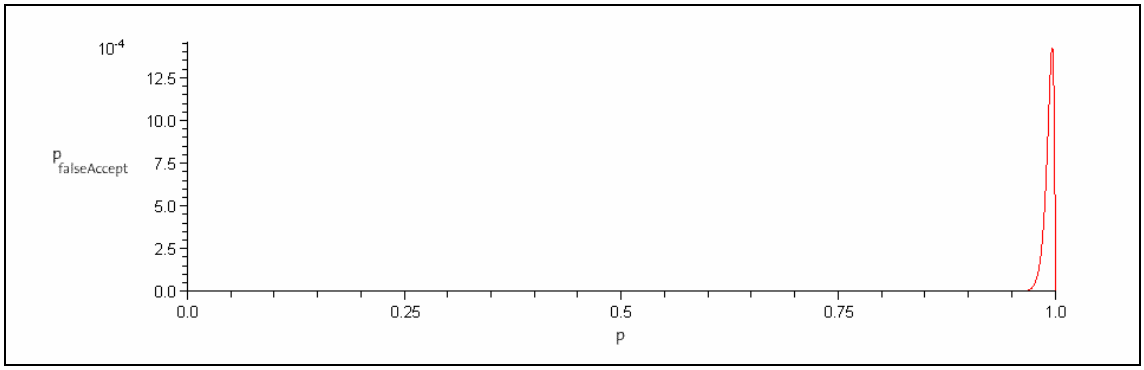


Figure 9: probability distribution of $p_{falseAccept}$ with X=2 and T=128

X	T	maximal $p_{falseAccept}$ p	maximal $p_{falseAccept}$ $p_{falseAccept}$
2	1	0.6667	0.1481
2	128	0.9961	0.0014
2	256	0.9981	0.0007

Figure 10: different X,T pairs and their maximal false accept coordinates

Figure 10 demonstrates that by limiting the transactionID to 128 values instead of 256 values (by introducing the described flag bit) does not significantly deteriorate the probability of false acceptance.

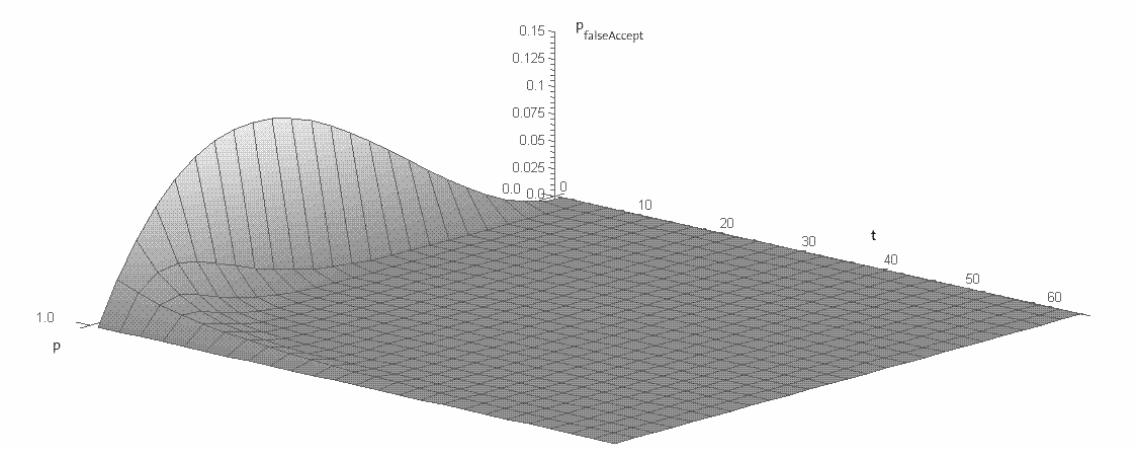


Figure 11: behavior of $p_{falseAccept}$ in dependence of p and t (t = variable transactionID cardinality)

In Figure 11 we see that with increasing cardinality of the transactionID the maximal value of $p_{falseAccept}$ decreases rapidly. In the graph the transactionID cardinality are assumed between 1 and 64. Also we notice a shift of the peak of $p_{falseAccept}$ when the transactionID cardinality is increased.
The "first data fragment"-flag is set by the sending and checked by the receiving part of Splimer (see Figure 12 and Figure 13).

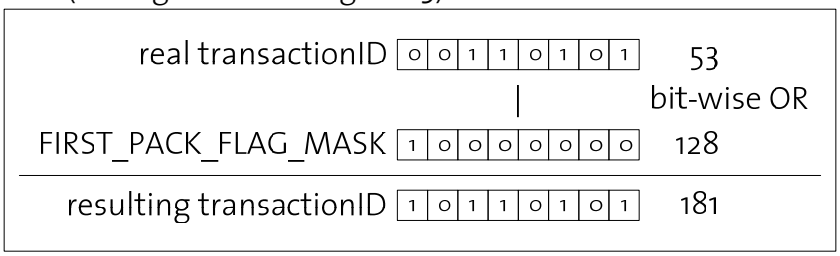


Figure 12: how to set the "first part" flag

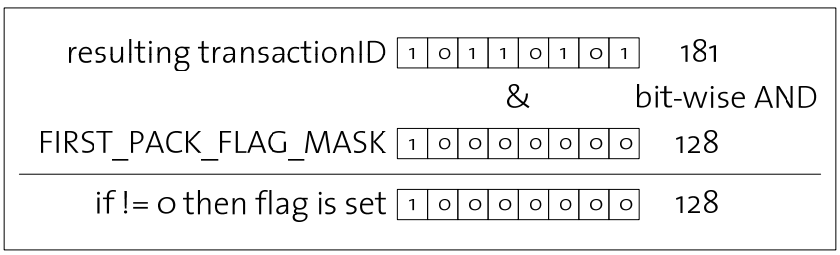


Figure 13: how to check for the "first part" flag

3.2.3 Normal Message Format (extended TOS_Msg)

message type	max payload size		
TOS_Msg	TOSH_DATA_LENGTH	= 29	bytes
Splimer_Packed_Msg	TOSH_DATA_LENGTH - PACKED_MSG_HEADER_SIZE	= 21	bytes
Splimer_Splitted_Msg	TOSH_DATA_LENGTH - SPLITTED_MSG_HEADER_SIZE	= 23	bytes
Splimer_Normal_Msg	TOSH_DATA_LENGTH - NORMAL_MSG_HEADER_SIZE	= 25	bytes

Figure 14: message types and their respective maximum payload size

When reducing the newly introduced fields in the previously defined message formats to the absolute minimum we are left with only three fields (in addition to the data segment): the sender's address the payload size information and the transaction ID (see Figure 15). These three fields occupy only 4 bytes of the TOS message's data section. This "normal message" type allows us sending up to 25 bytes in one packet.

Figure 15 shows the data layout for normal messages.

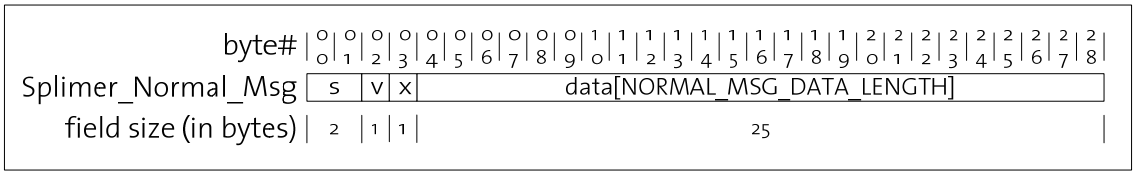


Figure 15: format for normal sized data

3.2.4 Determination of Sending Mode

Splimer packs data if it's smaller than PACKED_MSG_DATA_LENGTH. In case the data is between and including PACKED_MSG_DATA_LENGTH and NORMAL_MSG_DATA_LENGTH in size, the data is sent as normal message. If the data size is bigger than NORMAL_MSG_DATA_LENGTH it is split and sent as several packets. See Figure 16 as illustration.

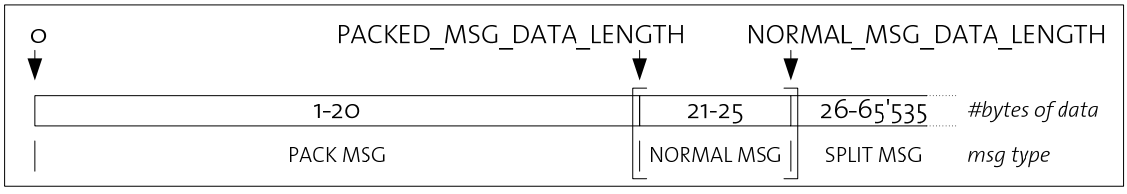


Figure 16: what data size qualifies for what message/transmission type

3.3 The Splimer Configuration

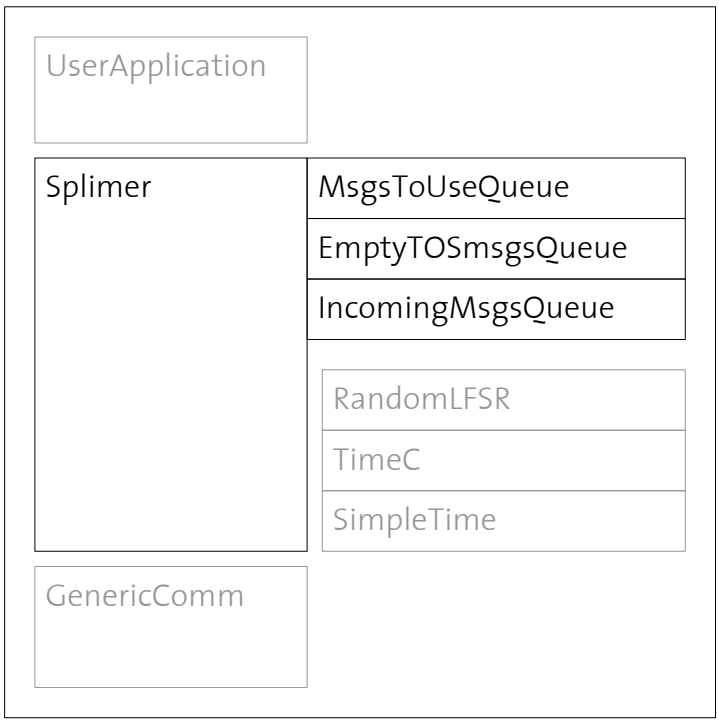


Figure 17: Splimer components and wiring

3.3.1 User Application Requirements

The requirements as for what commands may be called and what events have to implemented in a user application using Splimer are quite similar to what's needed to use GenericComm. However some changes have to be applied.

```
interface SplimerSendMsg
{
  command result_t send(      uint16_t address,
                              uint16_t length,
                              uint8_t *msg);

  event result_t sendDone(    uint8_t *msg,
                              result_t success);

  command result_t flush();
}
```

Figure 18: interface provided by Splimer

```
interface SplimerReceiveMsg
{
  event uint8_t *receive(     uint8_t *msg,
                              uint16_t length);
}
```

Figure 19: interface that must be provided by the user application

In Figure 18 can be seen that there are still three parameters needed in order to send data. The first parameter is the receiver's address. Compared to the 'send' command in GenericComm the parameter 'length' is an 16bit unsigned integer instead of a unsigned 8bit integer. This allows the user to send data upto 65535 bytes (approx. 64kBytes).
When using GenericComm, the user has to provide a pointer to a TOS message as third parameter. When using Splimer, the third parameter is a pointer to the data that the user wishes to transmit. The benefit of this proceeding is that the user does not have to think about whether the data fits into a given packet format. The only limit to the size of the data is the type of the length parameter.

If the user application reaches a state where it must be assured that all packets handed to 'send' were actually sent, the command 'flush' can be called. Flush sends the yet unsent data currently stored in the send buffer. See chapter 3.3.3.2 (on page 16) for more information on flushing.

The arguments of the sendDone event slightly changed compared to the one of GenericComm. The event sendDone is signaled not with a TOS message pointer as first parameter but with a pointer pointing directly to data. The sendDone event returns the pointer previously handed to Splimer via the send command call.

The user application has to implement the receive event (Figure 19). As the parameter's types and some of their meaning were changed in 'send', the parameters of this event have to be adapted accordingly. The pointer to the message that is received points directly to the data (not-as in GenericComm-to the TOS message). Finally, a length parameter is introduced as a 16bit unsigned integer, in order to support the data size that may be sent.
In order for Splimer to return received data it requires the user application to return a pointer to the allocated incoming message buffer when Splimer signals a receive event with a null pointer as message pointer parameter. This requirement demands the user application to allocate the memoy for incoming data. This is necessary because only the user application knows the maximally expected incoming data size. The null pointer receive is signaled only once, prior to returning the first received data.

3.3.1.1 **Behavior of the Splimer Interface** (unreliable communication)

In Figure 20 we see that the behavior of both send and sendDone are not different from the analogous function in GenericComm.

	SUCCESS	FAIL
command **result_t** send(addr, len, data)	data accepted (not sent yet)	wrong args or send in progress
event result_t sendDone(data, **succ**)	successful handover	sending failed on sender side
command **result_t** flush()	succ. flushed or no data to flush	failed to flush

Figure 20: behavior of the Splimer interface (unreliable communication)

The flush command returns FAIL only if flushing of a packet failed, meaning if a sending is still in progress.

3.3.2 The Three Queues

Figure 17 shows the configuration of Splimer. As for now, Splimer uses three different queues. MsgsToUseQueue holds pointers to TOS messages that are ready to be used; meaning that these messages are not used by any other part of the program. This queue is used by the sending part of Splimer. It acquires an unused pointer to wrap around the data to be sent. Upon signaling of Splimer's sendDone event, the message pointer handed to sendDone is enqueued at the end of the MsgsToUseQueue.
EmptyTOSmsgsQueue and IncomingMsgsQueue are used by the receiving part of Splimer. As it is crucial for the receive event to return an unused TOS message pointer instantly, a queue of unused message pointers is needed in order to be able to keep the incoming pointer for further processing. The following happens upon message receipt: the incoming pointer is enqueued in the IncomingMsgsQueue, an unused message pointer is fetched from the EmptyTOSmsgsQueue and returned.
MsgsToUseQueue & EmptyTOSmsgsQueue are initialized holding the maximal number of TOS messages. IncomingMsgsQueue is empty at the start of the program. The three queues' implementations do not differ much one from another. In fact, MsgsToUseQueue & EmptyTOSmsgsQueue are identical. Only their initialization differs from IncomingMsgsQueue.

```
interface MsgsToUseQueue {
     command result_t init();
     command result_t insert(TOS_Msg *msg);
     command TOS_Msg* getPtr2Msg();
}

interface EmptyTOSmsgsQueue {
     command result_t init();
     command result_t insert(TOS_Msg *msg);
     command TOS_Msg* getPtr2Msg();
}

interface IncomingMsgsQueue {
     command result_t init();
     command result_t insert(TOS_Msg *msg);
     command TOS_Msg* getPtr2Msg();
}
```

Figure 21: the same interface for all three queues

As shown in Figure 22 the queue sizes may easily be changed. Increasing the queue size affects the size of the compiled code.

```
enum {
     // queue settings for Splimer
     NEW_MSGS_QUEUESIZE = 2,
     INCOMING_MSGS_QUEUESIZE = 2,
     EMPTY_TOS_MSGS_QUEUESIZE = 2
};
```

Figure 22: queue size settings (file Queue_Settings.h)

3.3.3 The Splimer Module

The Splimer module is divided into two major parts, namely the sending and the receiving part. These two parts themselves are subdivided into three more components that handle the different message types.

3.3.3.1 Sending

Figure 23 illustrates Splimer's sending process. It shows the connection between the commands, tasks and events invoked upon a send request.

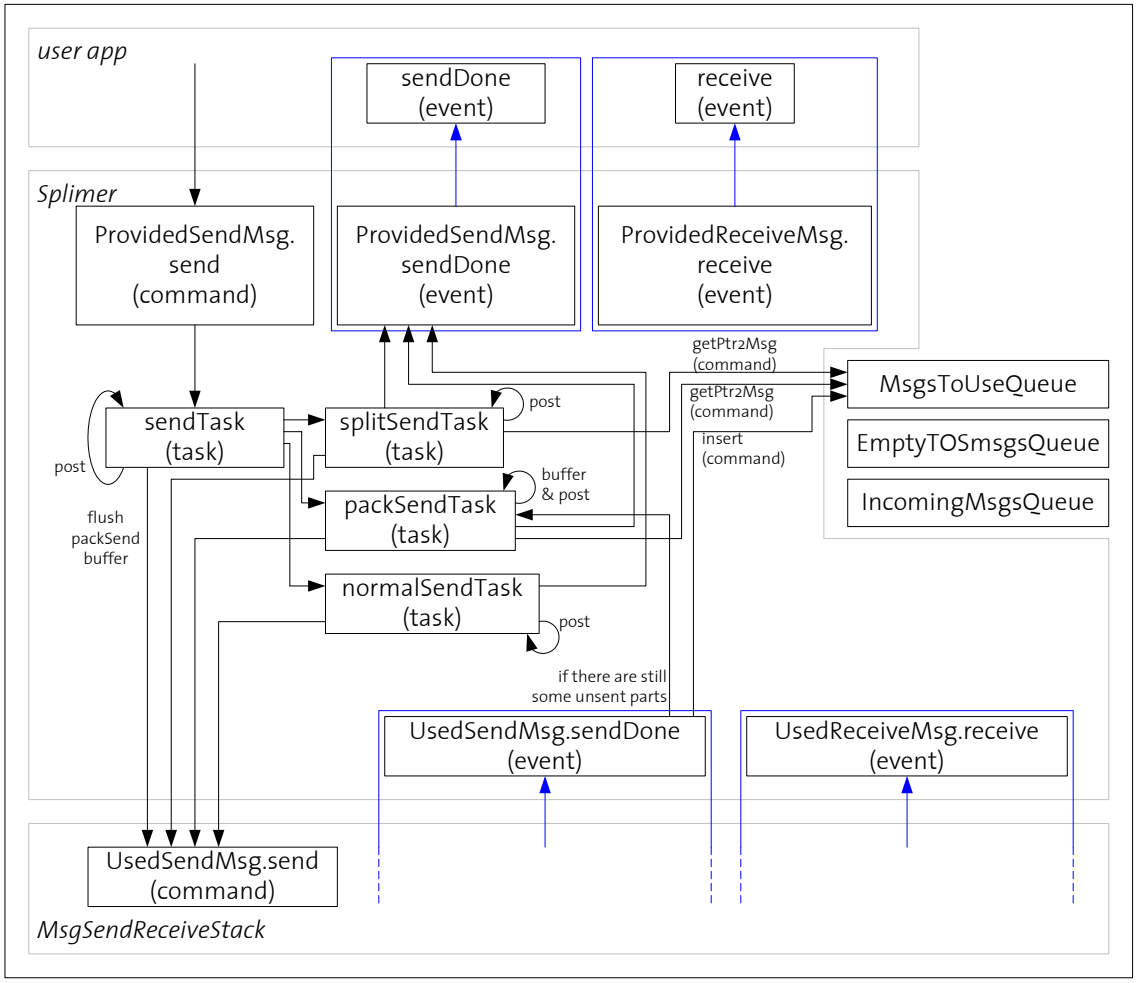


Figure 23: how Splimer processes a send request

The 'send' command is called with the proper parameters by the user application. If there's already a transmission in progress, 'send' returns 'FAIL' because the current version does not support pipelined transmission of data. If there's no send in progress, the send mode of the data is determined and a global variable is set to memorize the mode. 'send' then posts the 'sendTask' task which is responsible for the flushing of partially used buffers if necessary and posting the correct task for processing the data. Figure 24 shows all cases that demand flushing of the buffer. After having successfully flushed the buffer 'sendTask' posts itself to start the correct task for handling the data to be sent.

'sendTask' may post one of three possible tasks to handle the sending process of the data, namely 'splitSendTask', 'packSendTask' or normalSendTask'. All of these tasks check if there's a TOS message available; if not, they try to obtain one (see the following chapter 3.3.2 'The Three Queues' for details).
-*splitSendTask* splits the data into parts of the size defined in chapter 3.2.2 'Split Message Format (Splimer_Splitted_Msg)' and sends one packet and returns. The sendDone event detects if there are more parts left to be sent and reposts the splitSendTask as needed, until the last part is sent.
-*packSendTask* fits the data into the Packed Message Format (Splimer_Packed_Msg) described in chapter 3.2.1. If there's not enough space available in the current TOS message (hosting Splimer_Packed_Msg formatted data), the packed packet is flushed and the data that didn't fit into the now flushed message is put into a new TOS message wrapping a Splimer_Packed_Msg.
-*normalSendTask* sends the data in a slightly enhanced TOS message format (see chapter 3.2.3).

3.3.3.2 Flushing

When sending packets of different sizes it may sometimes be inevitable to flush a packed message before it actually is full. E.g. we were sending small data packets which were aggregated in a packed message(but not actually sent yet) and next we want to send a huge data packet which cannot be fitted in a packed message. Figure 24 shows a table explaining when a partially filled packed message is flushed automatically.

flush needed?

requested mode \ actual mode	split	normal	pack
split	no	no	yes
normal	no	no	yes
pack	no	no	if newAddr != curAddr

Figure 24: when to flush the message buffer

Please note that small data exceeding the current space left in a packed message also results in automatic flushing.

3.3.3.2.1 Explicit Flushing

Splimer provides a command called 'flush' taking no argument. This command, as described in chapter 3.3.1, enables the user application to flush the current message buffer to the underlying communication layer. This should be done by the user application when it wants to be sure that all data has actually been sent.

3.3.3.2.2 Implicit Flushing

It might be of interest to a user application not to use the flush command but the actual send command in order to flush the sending buffer. This is possible and is easily done by calling the send command with a null pointer instead of a pointer to data as parameter. Explicit and implicit flushing are semantically identical.

3.3.3.3 **Message Order Characteristic** (message sending/receiving sequence)

Splimer does not influence the sequence in which data packets arrive. The sequence in which the packets are send corresponds to the sequence the packets are received (sending order = receiving order). This behavior is called FIFO.
The exception to this rule are packets that were not successfully transmitted.

3.3.3.4 Receiving

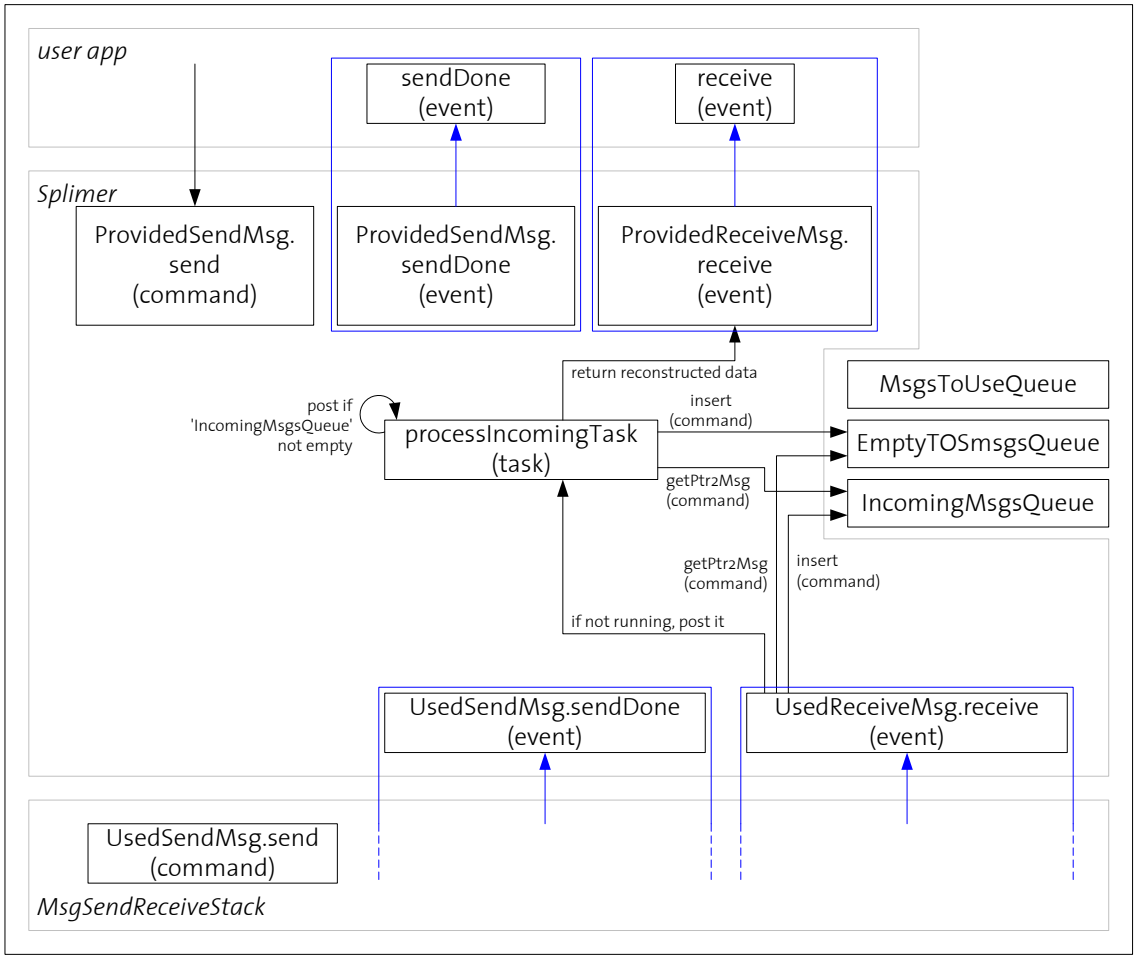


Figure 25: how Splimer processes a receive event

Once the receive event is signaled, it is checked whether the 'processIncomingTask' task is running; if it's not active, it is posted by the receive event. The received message is inserted into the 'IncomingMsgsQueue'. If the enqueuing fails, the message is not further processed and returned. If the insertion succeeded a free TOS message from the 'EmptyTOSmsgsQueue' is acquired and returned.
The 'processIncomingTask' task gets messages (pointers to TOS messages) from the 'IncomingMsgsQueue' and handles it accordingly. It tries to merge or unpack the messages where needed and returns the reconstructed original data via receive event to the user application.

3.3.3.4.1 Reconstructing the Original Data

As described in chapter 3.3.1 the user application is responsible for allocating and providing the memory for incoming data. The first time Splimer signals a receive event to the user application it is only for obtaining the pointer to the location where it can write the incoming data. This call can be identified (apart from being the first signaling of receive) by the message pointer parameter which is null.

Receiving and processing of normal messages is straight forward. The data is extracted from the TOS message and returned to the user application via receive event. The case of packed data is treated similarly. The Whole payload is extracted, split(according to the split mask), and each data packet is then returned to the user application via multiple receive signals.
Incoming split data is reassembled successively in the memory location provided by the user program. When the last part of the data is in place, the pointer to the memory location is handed to the user application using a receive event signal.

3.4 Timers in Splimer

In addition to the described functionalities and behavior, to ensure that no sending or receiving process can block the application and for enhancing the comfort of working with Splimer, different timers are used.

3.4.1 Timed Resend

If handing the message to the lower level fails, Splimer retries, after waiting a certain amount of time, sending the message. The number of retries is limited.

```
MAX_RETRIES = 3,       // how many retries before returning FAIL
WAIT_4_RETRY = 100     // [ms] time to wait before retry
```

Figure 26: variables defining number of retries and waiting time between tries (SplimerSettings.h)

After failing MAX_RETRIES times a sendDone is signaled containing FAIL as success parameter.

3.4.2 Receive Timeout

An important question to find an answer to was what happens if a sender starts sending a split message the last part is never received by the recipient?
This problem was solved by implementing a timer. The timer starts whenever a split packet (except the final packet) is received and it stops when its successor's arrival is detected.
On a timeout the whole message is rejected.

```
RECEIVE_TIMEOUT = 1000    // [ms] max time period of not
                          // receiving an expected followup
                          // packet of a split message
```

Figure 27: variable defining reception timeout (located in SplimerSettings.h)

4 The ReliableLayer Module

The Splimer module itself does not provide reliable communication. All that is guaranteed on a successful sendDone event is that the message was successfully handed over to the communication layer and that the message was received by the recipient. The latter can be found out when inspecting the ack field of the TOS message when the message is returned as a calling parameter of the sendDone event.
However even if the message is received, it is not guaranteed that it was successfully inserted into the recipient's queue which corresponds to a guaranty of being processed at the receiver's.
Hence what we want is a successful sendDone signaling iff the message is enqueued at the recipient's.

4.1 How ReliableLayer is Embedded in the Splimer Environment

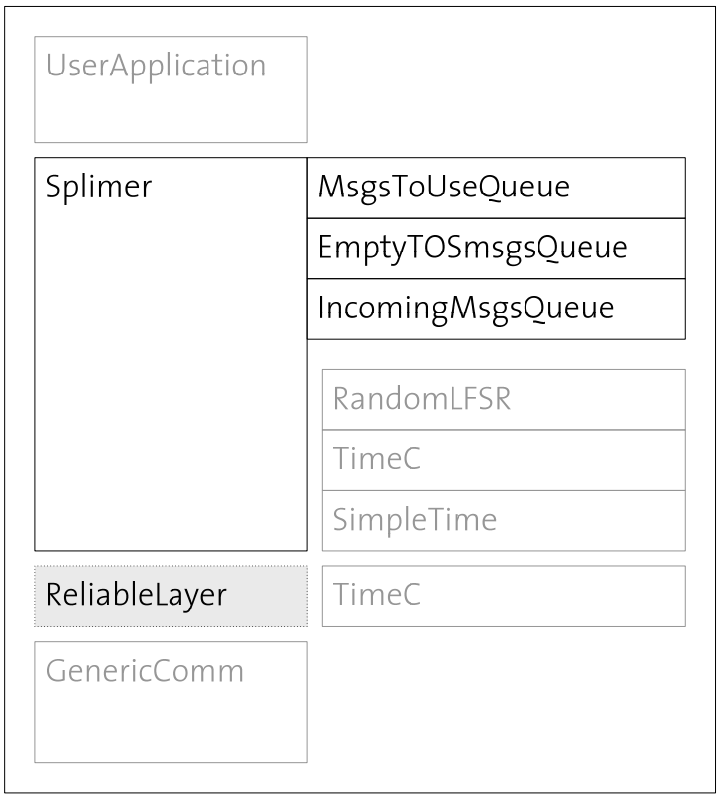


Figure 28: integration of ReliableLayer

As can be seen in Figure 28, the reliable layer is plugged between the Splimer and the communication layer (usually GenericComm).

4.2 Technical Details

4.2.1 Message Formats Introduced

```
typedef struct Ack_Msg{
     uint16_t sender;              // address of sender
     uint8_t acknack;              // ACK or NACK value
     uint8_t transactionID;        // transaction ID
} Ack_Msg;
```

Figure 29: Ack_Msg message format

The Ack_Msg message format is kept to three fields: the sender's address, ACK or NACK information and the transaction ID of the packet this packet is ack-ing.

4.2.2 Editable Variables

```
enum {
     ACK_WAIT_TIMEOUT = 1000, // [in millisecs]
     ACK_MSG_SIZE = 3,        // [bytes] length of ack pack
     ACK = 255,
     NACK = 0,
     ID_ACKNACK    = 68
};
```

Figure 30: non-editable & editable(bold) variables (in file AckPack.h)

There are two variables that may be edited by the user. The first is the ACK_WAIT_TIMEOUT variable which holds the maximal time the ReliableLayer at the sender's waits for an ACK message of the recipient before it automatically acts as if a NACK message was received. This value can be altered for performance fine tuning.
The second variable that may be edited is ID_ACKNACK. This value is simply the id parameter that is used when sending/receiving ack messages. However, altering this value is not recommended.

4.2.2.1 Quick_SendDone

The QUICK_SENDDONE constant can be found in SplimerSettings.h . Its default value is 1. The value of this variable affects the behavior of Splimer when sending data that is packed. This constant does however not affect the behaviour of Splimer when packet types other than packed packets are sent.
Usually when sending data the sendDone event is signaled short after the send command is called and the given pointer to data may be reused to send the next data. When using reliable communication however, a sendDone is signaled when the transmission of the packet was successful but the packet is not transmitted every time the user hands a pointer to Splimer. This leads to the consequence that the user application has to provide 22 pointers in the worst case.
The worst case is when the user application wants to send data of 1 byte in size. Then 21 bytes are used to fill the packet and the 22nd byte produces an automatic flushing of the packet.

Splimer features the QUICK_SENDDONE constant to enable quick sendDone signaling even if ReliableLayer is wired and the packet has not been transmitted yet.
This has the benefit that the user application does not have to manage a large number of pointers. The drawback is that the user application cannot assume successful transmission of a packet upon receiving a sendDone signal. More detailed information can be found in the SplimerSettings.h file.

4.2.3 Behavior of ReliableLayer

A send request is only accepted if ReliableLayer is not waiting for a sendDone event and if there's no incoming ack message pending, otherwise the call returns FAIL. If the call is accepted, the communication layer's send is invoked. When the call to the communication layer returns, ReliableLayer waits for the sendDone event to be signaled.
The sendDone event differentiates between ack messages and data messages. If it's an ack message, nothing is done but if it's a data message a timer is started giving the maximal time we wait for an ack message to arrive for the just sent data message.
When a message is received, it is checked if it's a data message or an ack message. This is done by comparing the incoming id to the ID_ACKNACK, the sender to the expected sender, by checking the content of the acknack field and its accord with ACK (see Figure 30) and by comparing the transactionID of the ack packet to the expected transactionID. In case of a valid ack message the timer started by sendDone is stopped and sendDone is signaled to Splimer. If a data message is received, receive is signaled at Splimer.
If Splimer's receive returns the same pointer as was provided to it, the enqueueing failed and therefore a NACK ack message has to be sent back to the sender of the message. If the returned pointer is different, an ACK ack message is composed and sent. Figure 31 illustrates the behavior of ReliableLayer.
ReliableLayer does not influence the sequence in which data packets arrive (sending order = receiving order).

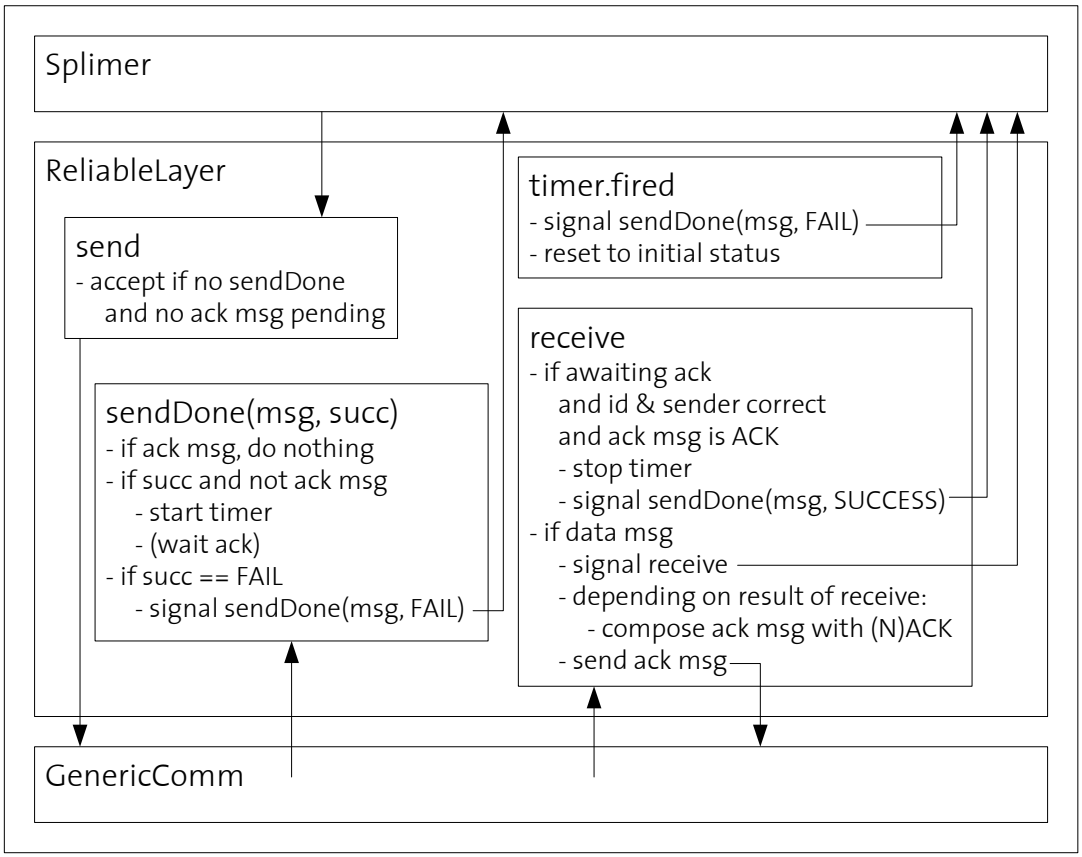


Figure 31: behavior of ReliableLayer

4.3 Timer in ReliableLayer

There is only one timer in use in ReliableLayer. As the timers in the Splimer module it is responsible to guaranty a continuous operational availability.

4.3.1 Ack Wait Timeout

Timer is used for preventing ReliableLayer from blocking upon non-reception or potentially extremely delayed reception of an expected acknowledgement packet.

```
ACK_WAIT_TIMEOUT = 1000  // [ms] how long to wait before we
                         //   treat not receiving ACK
                         //   message as NACK
```

Figure 32: variable defining timeout (located in AckPack.h)

Once the timeout is fired sendDone is signaled with FAIL as one of its parameters.

4.3.2 Timer Dependency

Some restrictions to the choice of ACK_WAIT_TIMEOUT's value apply. For the Splimer (with wired ReliableLayer) to work correctly the following equation has to hold:

ACK_WAIT_TIMEOUT ≤ RECEIVE_TIMEOUT - (WAIT_4_RETRY x MAX_RETRIES)

If this requirement is not met, the receiver may terminate the processing of incoming split data due to the firing of the RECEIVE_TIMEOUT timer. Figure 33 graphically shows the dependency between the different timers.

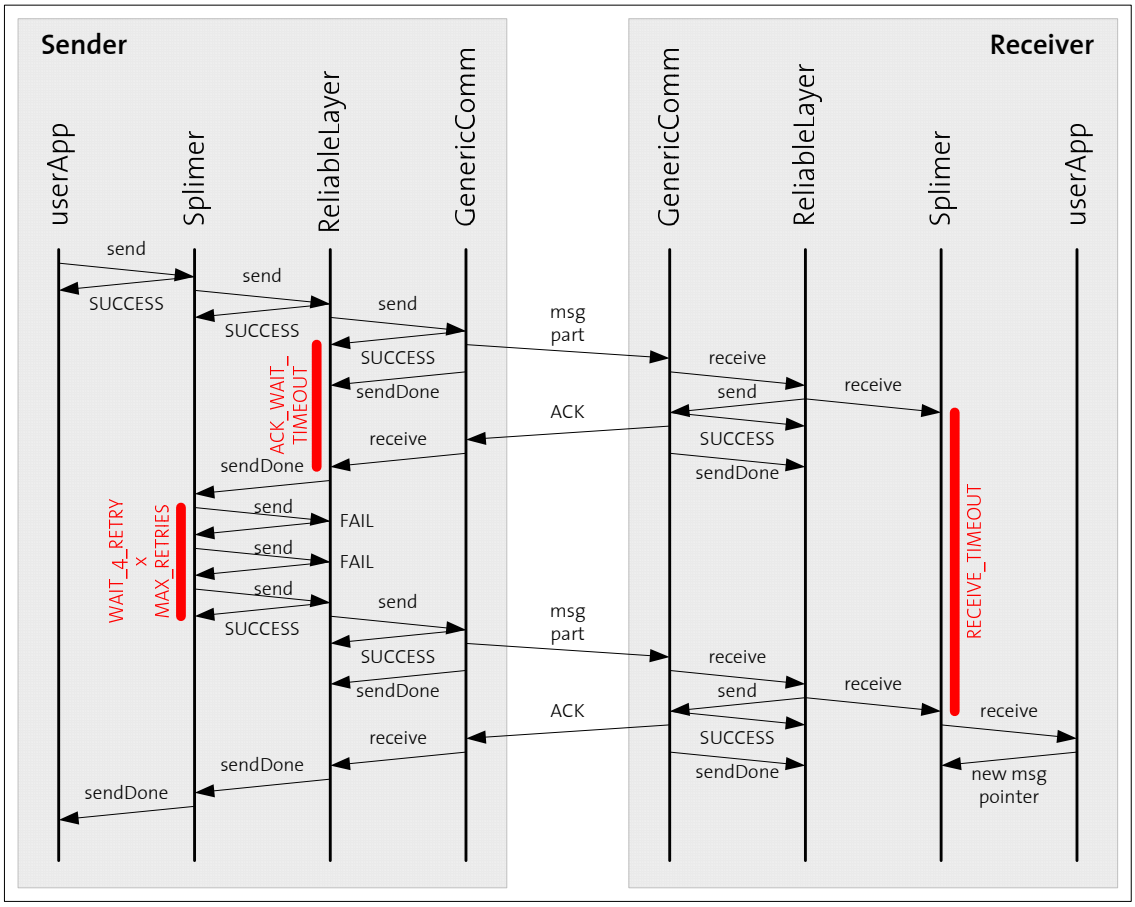


Figure 33: timer dependency visualized

4.4 Consequences for the User when Employing ReliableLayer

The user must provide PACKED_MSG_DATA_LENGTH pointers to the data the application wants to transmit. This is required because packed messages can only be acked after the message hosting the small packets is filled, transmitted and successfully enqueued at the recipient's. If the user application wants to transmit data chunks of 1 byte (worst case) in size, then a pointer for every byte of the payload of the host message is needed, hence PACKED_MSG_DATA_LENGTH pointers.

4.4.1 **Behavior of the Splimer Interface** (reliable communication)

	SUCCESS	FAIL
command **result_t** send(addr, len, data)	data accepted (not sent yet)	wrong args or send in progress
event result_t sendDone(data, **succ**)	successful transmission	sending failed or NACK received
command **result_t** flush()	succ. flushed and ACKed	failed to flush or NACK received

Figure 34: behavior of the Splimer interface (unreliable communication)

Figure 34 illustrates how the behavior of Splimer with ReliableLayer wired can be interpreted. Only a successful sendDone signal guarantees that the data has arrived at the receiver and will be processed.

5 Discussion & Conclusion

The Splimer module enables users to send messages of arbitrary size. Messages bigger than the regular packet size are possible by automatically splitting and merging. Packing and unpacking of small messages is implemented also. The present implementation is optimized for code size; i.e. implementation is kept as simple as possible in favor of the user's application. Splimer is totally independent of ReliabileLayer. This means that you can save memory by not including the reliability layer. (→ROM/RAM size discussion chapter ...)
Following drawbacks have to be considered when using the reliable layer:

- the response time is noticeably slower
- additional traffic is generated with ACK-packets

Apart from this the reliability layer provides the assurance that when the sender receives an ACK from the receiver the packet is guaranteed to be processed.

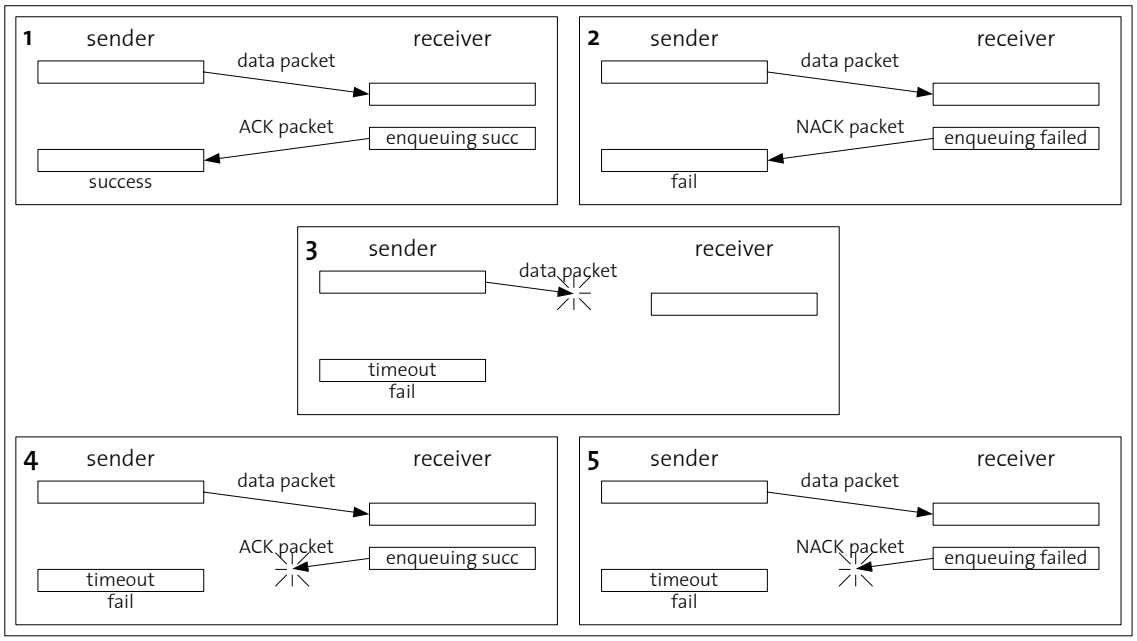


Figure 35: ReliableLayer protocol behavior diagram

The provided semantic for reliable transfer is at least once(if QUICK_SENDDONE is set to 0). See Figure 35 for an illustration of the reliable protocol's behavior. If the packet gets lost or cannot be delivered FAIL is reported(case 3). The only case implying some uncertainty is when a receiving node dies or gets out of reach before sending an ack message(cases 4 and 5): in this case the sender times out and reports FAIL although the message could have been fully processed(case 4).

As availability of memory is limited on wireless nodes, it is important to consider this issue throughout the whole development process of the application. Splimer was designed trying to minimize memory usage.

5.1 ROM, RAM and Runtime Analysis

The TinyNode version 1.x have a maximal ROM of 49152 bytes and 10240 bytes of RAM. TinyOS 1.1 occupies around 30% of the ROM and approximately 5% of the RAM available.
The Splimer module itself uses 6.3% ROM and 6.9% ROM. Adding ReliableLayer adds another 0.6% ROM and 0.6% RAM usage. Hence when using TinyOS with Splimer and ReliableLayer 36.9% of ROM and 12.5% of RAM are used. This leaves 63.1%(27858 bytes) ROM and respectively 87.5%(8872 bytes) RAM space for other applications. The mentioned percentage values were deduced from Figure 36.

[all values in bytes]			difference to no Splimer		difference to Splimer only	
	ROM	RAM	ROM	RAM	ROM	RAM
TinyNode 1.x	49152	10240				
no Splimer	14890	594				
Splimer only	17986	1302	3096	708		
Splimer + ReliableLayer	18294	1368	3404	774	308	66

Figure 36: ROM and RAM analysis[4]

	send mode	runtime [ms]	#trans-missions	payload in avg used in bytes	in %	overhead [bytes]	[%]
no Splimer							
4 x 25 bytes	-	128	4	25 of 29	86.21	7	21.88
20 x 5 bytes	-	384	20	5 of 29	17.24	7	58.33
Splimer							
1 x 100 bytes	split	160	5	20 of 23	86.96	13	39.39
4 x 25 bytes	normal	128	4	25 of 25	100.00	11	30.56
20 x 5 bytes	pack	224	5	20 of 21	95.24	15	42.86
Splimer + ReliableLayer							
1 x 100 bytes	split	256	10	20 of 23	86.96	24	54.55
4 x 25 bytes	normal	192	8	25 of 25	100.00	22	46.81
20 x 5 bytes	pack	512	10	20 of 21	95.24	26	56.52
20 x 5 bytes QUICK_SENDDONE = 0	pack	352	10	20 of 21	95.24	26	56.52

Figure 37: runtime & space analysis

Figure 37 is a table for comparing the runtime and data transmission statistics. Ideal conditions are assumed, i.e. no failed transmissions, no timed resends.

[4] Data collected from release candidate 1 of Splimer.

The packing mode only slightly decreases the runtime but diminishes network traffic and augments the used payload space per packet.
The overhead size is quite high because of the additional information that has to be transmitted (header fields of custom message formats). When enabling reliable communication the overhead percentage increases since the acknowledgement packets add to the overhead.

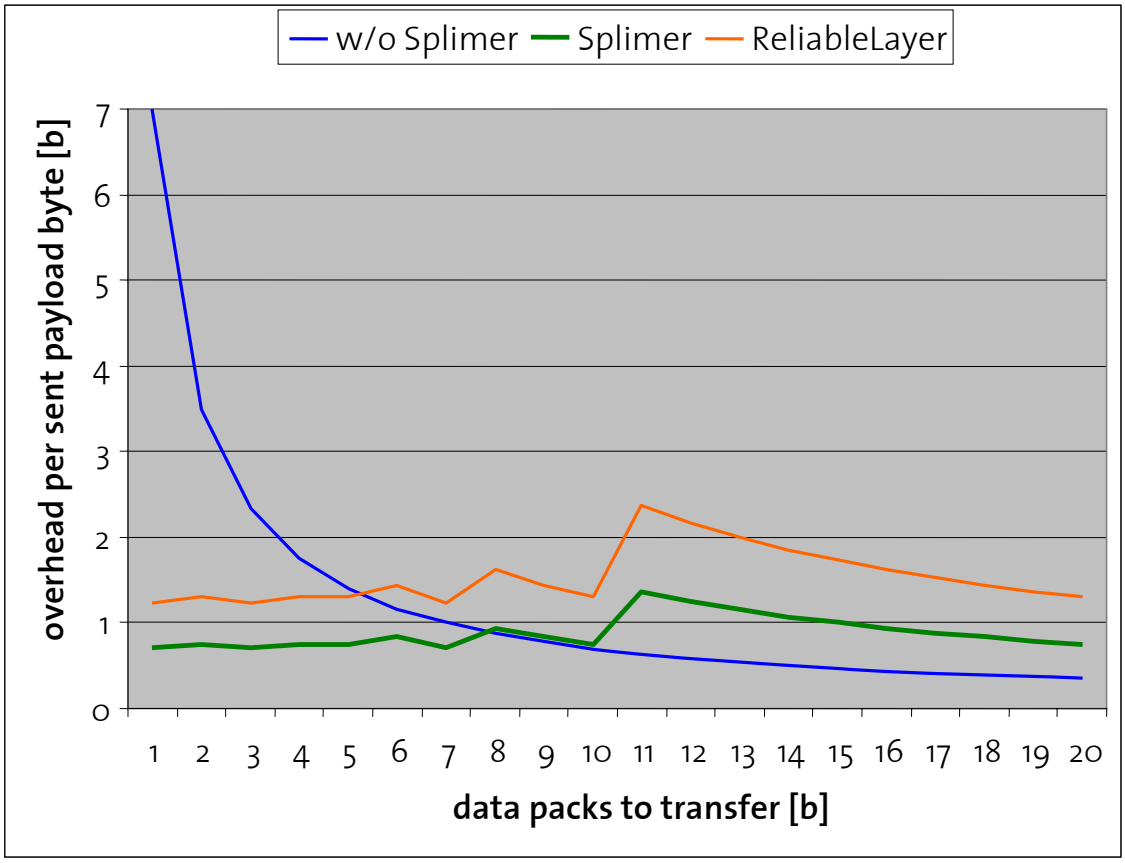


Figure 38: overhead per sent payload byte

Assuming only equally sized packets are sent, Figure 38 shows that the packing mode has a better overhead/payload ratio for data packets that are 7 bytes or less in size. When using ReliableLayer the ratio is better if data packets smaller than 5 bytes are sent.

6 Future Work Proposition

A module able to split outgoing and merge incoming messages is a solid base for handling data of arbitrary size. It further enables handling of faulty received packets without the need of resending the whole data again, e.g. one could only *resend the missing/lost part*, if we wished to implement a more sophisticated data transfer protocol.
Further work may include supporting the 'id' parameter (*parameterized functions*) as in GenericComm (e.g. send[id](addr, len, msg)).
A handy feature would be *pipelined processing of send requests*. In the current version of Splimer a send call only succeeds if there is no other data transfer in progress. That means send calls fail until the sendDone event to the previous message is signaled.
Another improvement would be the *concurrent receiving* and processing of received data from different senders.
If a bounded delay for data transmissions is required, a *repeating timed flush* of the outgoing buffer could solve the task (only needed in pack mode, other modes flush right away).
A measure to *prevent a lot of network traffic* (especially when sending large data packets) are NACK packets containing a field indicating how long a sender should wait before retransmitting the packet that was refused.