CS221 – Computer Networks and Security

Project #7

Fraglets and Sockets

by Jasenko Zivanov

Abstract:

The following document has been written in the course of a students project concerning fraglets, a metabolistic approach to communications protocols, at the University of Basel. It describes the steps taken to integrate basic network functionality into an existing fraglet interpretation environment, some difficulties encountered while designing a reliable fraglet based communications protocol, the methods adopted to solve them and a description of said protocol. It includes data from some measurements of the performance of that protocol and proposals regarding the fraglets instruction set.

Introduction:

Fraglets are a new programming model inspired by metabolic pathways in living organisms. They allow a simple construction of some unusual mechanisms, but the implementation of certain well-known structures (like data containers) can become quite tricky.

In the course of this project, the first step was the implementation of a UDP interface that allows fraglets to be sent from one **interpretation environment** to another.
Then, a fraglet-based **protocol** was created, which was by far the most interesting and difficult part of the assignment.
In the end, some measurements we performed on the **performance** of that protocol under different conditions.
While designing the protocol, certain ideas regarding the further development have emerged. Those are described in the last section, **proposals**.

In order to simplify navigation of this document, the corresponding paragraphs have been colour-coded as follows:

Due to the highly modular nature of this assignment, the report has been kept in a similar structure. Besides of a certain number of cross references between the sections, they are thematically rather far apart. As the order of the sections is more or less the same as the order in which the individual tasks were performed, it still makes a lot of sense to read them in that order.

Part One:

The Interpreter

1. Input Files

Changing the interpreter program made it necessary to also make minimal alterations to the input syntax, though input files written using the old syntax will still be compatible. The only parameter changed is the node configuration parameter '**a**'.

The new syntax goes as follows:

local context:

```
a    name
```

input node:

```
a    name    "local"    port
```

remote node:

```
a    name    ip-address    port
```

Where the string '*local*' means, the node will receive fraglets arriving at port *port* of the machine the interpreter runs on, and an *ip-address* (dot notation) means that the node is running elsewhere, and fraglets sent to that node are actually transmitted to that port at that address.
No instructions are executed for fraglets stored at a node declared remote.
The name used on a remote machine for the node a fraglet is sent to is furthermore irrelevant, fraglets are transported without any node specification and arrive at any node listening on that particular port at that address. In order to maintain a certain readability, it might be wise to still use the same names for a certain node on both sides.

Let us look at an example configuration:

```
Instance A:
a    beta     udp    local      1138
a    gamma    udp    127.0.0.1  1139

Instance B:
a    gamma    udp    local      1139
a    beta     udp    127.0.0.1  1138
```

Here, every fraglet on ***instance A*** sent to ***udp:gamma*** will arrive at node ***gamma*** of ***instance B***, while every fraglet inside ***instance B*** sent to ***udp:beta*** will end up in node ***beta*** of the ***A instance***.

Now, consider the following:

```
Instance A:
a     alpha     udp   local       1138
a     beta      udp   127.0.0.1   1139

Instance B:
a     gamma     udp   local       1139
a     delta     udp   127.0.0.1   1138
```

In this case, each fraglet sent to ***udp:beta*** from ***instance A*** will arrive at node ***gamma*** of ***instance B***, while each fraglet sent to ***udp:delta*** from ***instance B*** will arrive at node ***alpha*** of ***instance A***. This behaviour allows for some additional flexibility, but, in order to conserve readability, its use should be limited to cases where the architecture of the remote instance really is unknown.

2. Program Parameters

Two command line paramaters were added to the program:

-lose N: loses each unreliably sent packet with a probability of 1/*N*, unless *N* equals zero.

-log N F: logs each reaction happening at node *N* to a file *F*, using the same output syntax as the standard console output (at debug-level one).

They were both necessary for the debugging of the new protocol.

3. Notes

- All fraglets are transported in ASCII form. As the unsigned short vocabulary of the environment is created at runtime, that representation generally differs from instance to instance.

- As the impact of interpretation-time on the round-trip-time by far outweighs the one of the time really needed to deliver a packet over loopback* and return the ACK, the **wait** instruction is still based on instruction count and not real-time.
*See section **performance** for more*

*) this would probably even apply if the packet was sent to Australia. No machines located in Australia were at my disposal at the time, so this is not confirmed.

Part Two:

The Protocol

Designing fraglet based programs may seem quite difficult in the beginning, but compared to many other things that get much easier as time goes by, it actually remains quite mind-boggling and nerve-challenging. Still, it is a very interesting programming paradigm, and in the following I would like to demonstrate certain difficulties encountered along the way, and the methods adopted to solve them – just before I face the reader with the complexity of a diagram of the entire protocol.

In order to grant at least some perspective before we begin, here is a rough outline of the new protocol:

- **Circular data store, allowing for continuous sending of data**
- **Data is transmitted one symbol at a time**
- **Each data element is transported along with one token from a ring as header**
- **All tokens from the ring can be away at the same time (pipelining)**
- **Receiver expects one particular token, only that one is acknowledged**
- **Of each data element sent away, a backup is stored and a copy of the token is placed at the end of a list**
- **If no ACKs arrive during a certain time interval, all backups are resent, but neither deleted nor their tokens removed from the list**
- **Each incoming ACK returns the token, deletes the associated backup and removes the first entry in the list**

1. Multiple Contexts

Only after having altered the interpreter program, was it that the possibilities of multiple contexts became apparent. Until then, different nodes were mentally associated with different fraglet environments running on different machines, connected by a slow and possibly lossy connection. Introducing real remote nodes suggested using local nodes as local 'reaction chambers', similar to the different compartments in a living cell.
For one, this allowed modularizing the fraglet code, so single reactions could take place entirely within one context, while fraglets with a certain header could be used as interfaces between contexts, thereby reducing design complexity and making keeping track of used identifiers much easier.
For the other, multiple contexts also allowed storing multiple associations to one symbol. On the following pages there are two examples of the latter.

Example 1.1 - Counting

The data to be sent was supposed to be ordered, so the reordering capabilities of the protocol could be tested. Now, if we were handling real data, it probably would have been wise to store all the data symbols in one singular fraglet with a specific header, and then always slice one symbol off for transportation. In this test however, it was decided to store the data in a circular order, thereby allowing for a measurement of performance over an indefinite amount of time.
This is when multiple contexts came in handy for the first time:

In one node, labeled '**producer**', we have the following fraglets:

N times	**producer[*n(i)*:*data(i)*]**	(0 <= i < N),
and once	**producer[next_data:*n(i)*]**	

*Note that symbols printed italic are placeholders for symbols of the form **n0**, **n1**, etc. (or **data0**, **data1**, etc. respectively).*

This way we know which of the N data elements is supposed to be accessed next, and can access it easily through the reaction:

(Reaction R1):

producer**[get_next]** + producer**[next_data:***n(i)***]**
+ producer**[***n(i)***:***data(i)***]**
→ producer **[***n(i)***:***data(i)***]** + producer**[next_data:***n(i+1)***]**
+ emitter**[next_data:***data(i)***]**

The details of the reaction are left out here, but they can be looked up in the fraglet code to this report. The important thing is, we can retrieve the symbol '**n(i)**' because it carries the known header '**next_data**' in one fraglet, and then access the fraglet carrying the header '***n(i)'*** to retrieve the corresponding data element and write something like '**send:udp:emitter:next_data**' in front of it (although, in the real protocol, copies of the ***n(i)*** and ***data(i)*** symbols need to be created first).

Now, the problem we are faced with is the following: After one data-element has been accessed, how do we know which ***n(i)*** is the next one (labeled '***n(i+1)***' in our case)?
The straightforward solution is to use fraglets containing all ***n(i)***s as headers and the corresponding ***n(i+1)***s as tails. Unfortunately, we already have fraglets carrying the headers ***n(i)*** in this context, so there is no way of knowing if we are attempting to access **[*n(i)*:*data(i)*]** or **[*n(i)*:*n(i+1)*]**. We could try and use a different set of number-identifiers ***m(i)*** and store the current-next relationships in this alphabet, that is, have fraglets of the form **[*m(i)*:*m(i+1)*]** swimming around instead.
But even in this case, in order to translate from ***n(i)*** to ***m(i)***, we would still need fraglets carrying ***n(i)*** as headers floating around - as well as other fraglets with ***m(i)***s as headers, as those would be needed for the reverse-translation.

So the only way to solve the problem using this approach, is by putting the current-next relationships into a separate context - in our case it has been named '**counter**'.
Then, we can use the following simple reaction to get the next ***n(i)*** into our producer:

Reaction R2:

counter**[next:***n(i)***]** + counter**[***n(i)***:***n(i+1)***]**
→ producer**[next_data:***n(i+1)***]** + counter**[***n(i)***:***n(i+1)***]**

The **producer** only has to create a copy of ***n(i)*** at some point during **Reaction R1,** and send it into the counter with the tag '**next**' attached to it.

Combined with **Reaction R1**, we receive the following interface that will deliver the next ***data(i)*** to the node designated **emitter** each time it is used:

Data Retrieval Interface:

emitter**[**send**:**udp**:**producer**:get_next]**
→ emitter**[next_data:***data(i)***]**

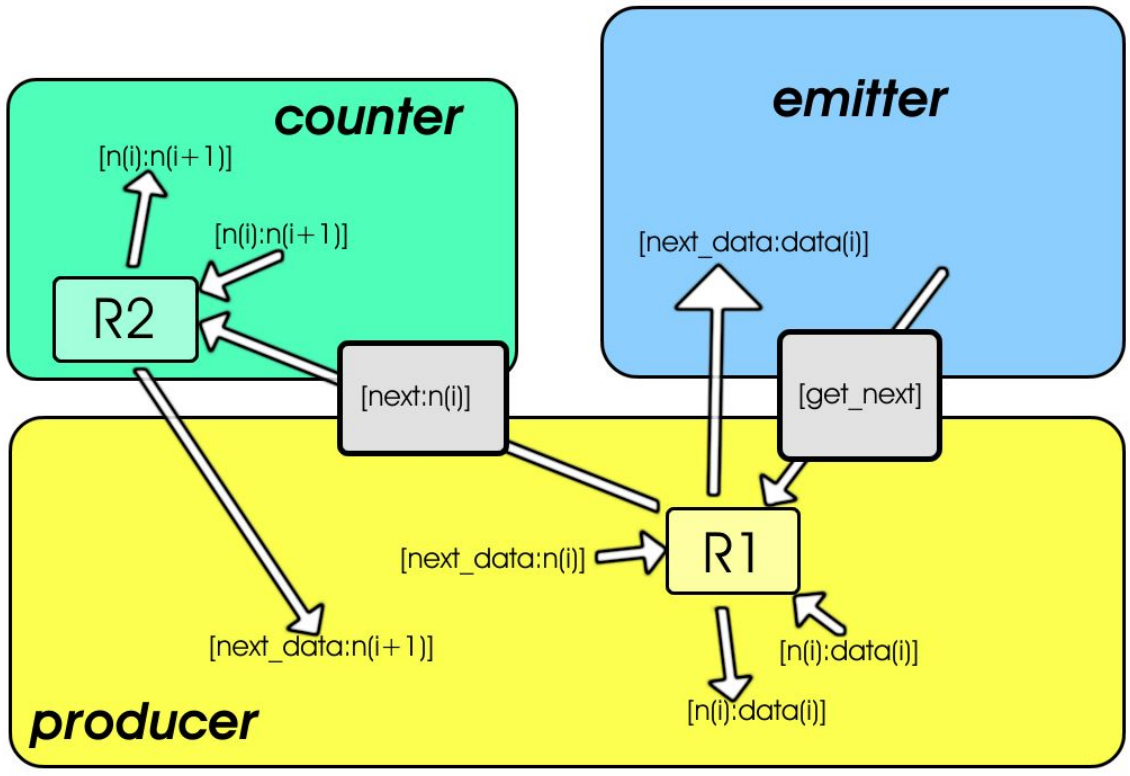


Image 1.1: Data Retrieval Interface

Example 1.2 - Tokens

Similar to the data elements in the data store, tokens, too, are aligned in a predefined circular order. Now that we have used the term 'interface' in conjunction with contexts, we can declare the expected behaviour of our token store in the form of an interface:

Token Retrieval Interface:

emitter**[**send**:**udp**:**token_order**:get_token]**

→ emitter**[token:**t*(i)*]**,** (as soon as the next token has been returned to the store, or instantly if it is there)

In order for it to work, we will of course need a context named **token_order**.
token_order works in a similar way as **counter**, except for the fact that it stores the next token itself, and that it does not send that next token to the **emitter** directly, but instead, it sends it to a context named **tokens**, with the header **confirm** attached to its head.

Reaction R3:

token_order**[get_token]**
+ token_order**[next_token:***t(i)*]** + token_order**[***t(i)*:*t(i+1)*]**

→ tokens**[confirm:***t(i)*]** + token_order**[***t(i)*:*t(i+1)*]**
+ token_order**[next_token:***t(i+1)*]**

Here, as well, most of the fraglets are responsible for copying the symbols from the association fraglet **[*t(i)*:*t(i+1)*]**, in order to restore it after the reaction, so all of them have been left out and can be looked up in the code.

Besides of the fraglets needed for the reaction, **tokens** contains only single-symbol fraglets of the form **[t(i)]** (initially all of them). Upon arrival of a **[confirm:t(i)]**, certain fraglets stored in **tokens** try to match the arriving **t(i)** with one already present in **tokens**, and forward it to the **emitter**. The according **t(i)** is thereby removed from **tokens**. This way, we can assure that no token can be delivered from **token_order** to the **emitter**, unless (or before) we have returned the same token to **tokens** (we do this upon an ACK).

Reaction R4:

tokens**[confirm:***t(i)*]** + tokens**[***t(i)*]** → emitter**[token:***t(i)*]**

To prevent confusion, a **[get_token]** is sent from the **emitter** to **token_order** only when another token has arrived (except for the first one of course, this is actually always the very first instruction that is executed).

There is a diagram on the following page.

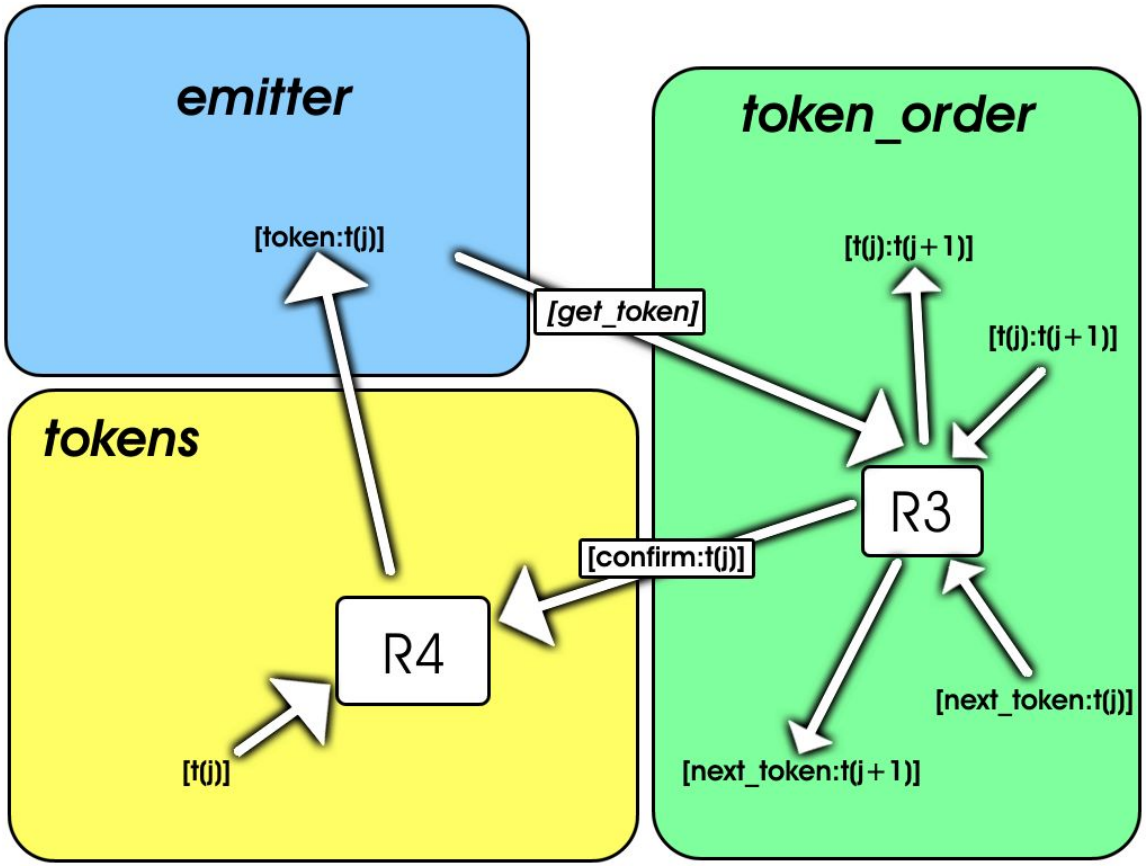


Image 1.2: Token Retrieval Interface

end examples

Those were two things that could only be accomplished using multiple contexts, because they require multiple associations to single symbols. In **example 1.1**, we had to associate numbers with data elements as well as the succeeding numbers, while in **example 1.2** tokens had to be ordered in the same circular fashion as the numbers, and we needed a way to store the information if a token is available or not.

Note that the use of multiple contexts violates the **active networking** paradigm, since it relies on an existing node architecture. This could be solved by introducing an instruction that spawns a new node with a certain name. This instruction could also be executed at constant time (by hanging the new node in at the head of the node-list), the removal of a certain node, however, can not. Allowing fraglets to create nodes would also introduce further complications considering namespace uniqueness, but most of them could be solved by allowing nodes to reside within other nodes and making them invisible from the outside (similarly to the way namespaces help to solve naming conflicts in C++ libraries).

*see section **proposals***

2. The Timer

New to the instruction set is the **wait** instruction.
However, storing a separate timeout for each data packet is quite clumsy (mostly because it is impossible to delete a packet having the symbol **wait** at its head - so all backups would have to be prevented from being resent rather than be explicitly resent). Instead,
a timer context was created, that will send a NAK to the emitter each time its timeout runs out. Each time an ACK arrives, we can delay the timer, so it will send us a NAK only if no ACKs have arrived for a while (it still has to be calibrated).

To build the timer, a fraglet containing a sequence of **N** times a singular symbol **d** and the symbol **ALARM** at the end has been used (**timeout sequence** := **[d:d:d:..:d:ALARM]**).
Furthermore, there is (almost) always a fraglet waiting around (it has a **wait** instruction at its head), to slice one **d** off from the head of the **timeout sequence** and then restore itself.
If the symbol **ALARM** is detected as header in the context, the NAK is sent to the **emitter** and the **timeout sequence** is restored to its initial state.
If a delay is triggered (ie. the fraglet **[delay]** sent to the timer), the **d** at the head of the **timeout sequence** is simply overwritten by multiple **d**s.

As this is the simplest context of all, we can (for once) present all the fraglets right here:

```
timer[wait:match:d:split:X:*]
timer[matchp:X:wait:match:d:split:X:*]

timer[matchp:ALARM:split:send:udp:emitter:nak:*:d:d:d:d:ALARM]

timer[matchp:delay:match:d:d:d:d:d]
timer[matchp:reset:match:d:split:d:d:d:d:ALARM:*:nul]

timer[d:d:d:d:ALARM]      # 'timeout sequence'
```

3. Lists

Storing backups of data sent away, so they can be sent again if the timeout for an ACK runs out, has been handled the following way: Backups of all outbound symbols are stored in one context, **backups**, with the ***t(i)****s,* the tokens sent away along with those data symbols, as headers.
Furthermore, all outbound tokens are stored in a list in a context called **missing_tokens**.
When a token is sent away, a copy of it is appended to the end of the list and the associated backup is sent to **backups**.
When an ACK arrives, the token at the beginning of the list is removed and the corresponding backup in **backups** is deleted.
When a resend is triggered, a copy of the list has to be created and one of the two lists sent to the **backups** context, where it will trigger a resend of all backups associated with the tokens contained in the list.

Fulfilling the first two conditions is easy:

To add an element to the list, we can simply write '**match:T:list**' at the beginning of the list and have the fraglet **[T:new_element]** swim around. This way, **new_element** is attached to the end of the list and the list carries the header '**list**' afterwards.

add:

[list:*t(i)*:*t(i+1)*:..:*t(j-1)*], [missing:*t(j)*]

→ **[match:T:list:*t(i)*:*t(i+1)*:..:*t(j-1)*], [T:*t(j)*]**

→ **[list::*t(i)*:*t(i+1)*:..:*t(j-1)*:*t(j)*]**

Removing an element is not difficult either, all we need to do is **exchange** the first element first with the symbol **list** and then with an asterisk, and then invoke **split** on the list.

remove:

[list:*t(i)*:*t(i+1)*:..:*t(j)*] + [returned:*t(k)*]

→ **[exch:r0:list:*t(i)*:..:*t(j-1)*]** → **[r0:*t(i)*:list:*t(i+1)*..:*t(j-1)*]**

→ **[exch:r1:*:*t(i)*:list:*t(i+1)*..:*t(j-1)*]** → **[r1:*t(i)*:*:list:*t(i+1)*..:*t(j-1)*]**

→ **[split:a:*t(i)*:*:list:*t(i+1)*..:*t(j-1)*]**

→ **[a:*t(i)*] + [list:*t(i+1)*..:*t(j-1)*]**

*Note that **t(k)**, the token returned with the ACK, is not necessarily the same as **t(i)**, since although the receiver only sends an ACK for the next expected token, an ACK could be lost along the way - our protocol cannot cope with that. It would be however rather easy to fix this, if we had the **equals** instruction at our disposal. Note also that if some ACKs only arrived in the wrong order, no damage would occur.*

*see section **proposals***

Creating a copy of the list, however, proved to be quite complicated.
A mechanism was conceived that could split off the first entry from the list and append two copies of it to the ends of two new lists, called **list0** and **list1**. Unfortunately, there is no notification after the original list has been emptied - the former list fraglet just disappears instead. So, in order to have a reliable termination condition, another list (labeled **size)** has to be managed, one that has the single entry '**l**' at all those positions, where the original token list has elements. From the **size** list, we remove the header, put a **list_empty** symbol at the end, and remove an **l** for each copied (token-)list-entry (we have to move all **l**s to another new fraglet called **new_size**, so we do not lose the **size** list in the process). This way, as soon as the **list_empty** symbol has been exposed, all elements have been copied and we can rename one copy of the list back to **list**, and send the other to **backups** (after certain additional modifications). Of course, this means that the **size** list has to be extended by an **l** each time we add an element and an **l** has to be removed each time we remove one.
On the following page, I will try and explain the procedure again in the form of a simplified algorithm.

resend:

[list:*t(i)*:*t(i+1)*:..:*t(j)*] + [resend_all] + [size:l:l:..:l]

→ **[copied_list:*t(i)*:*t(i+1)*..:*t(j)*] + [list0] + [list1]**
+ [list_size:l:..:l:end_list] + [l] + [new_size]

on (match:l) {

[copied_list:*t(i)*:*t(i+1)*..:*t(j)*] + [list0] + [list1]
+ [list_size:l:..:l:end_list] + [l] + [new_size]

→ **[copied_list:*t(i)*:*t(i+1)*..:*t(j)*] + [list0] + [list1]**
+ [list_size:l:..:l:end_list] + [new_size:l]

→ **[copied_list:*t(i+1)*..:*t(j)*] + [list0:*t(i)*] + [list1:*t(i)*]**
+ [list_size:..:l:end_list] + [l] + [new_size:l]

}

on (match:end_list) {

[list0:*t(i)*:*t(i+1)*..:*t(j)*] + [list1:*t(i)*:*t(i+1)*..:*t(j)*] + [new_size:l:l:..:l]

→ **[list:*t(i)*:*t(i+1)*..:*t(j)*] + [size:l:l:..:l]**
+ [send:udp:backups:resend:*t(i)*:*t(i+1)*..:*t(j)*:*end_list*]

}

That was an attempt to describe the process of copying the list of missing tokens in a C-like syntax. Lots of steps have been left out. See the attached code, if you would like to know the details.

4. Controlling Parallelism

Although, as we have seen in **paragraph 3**, fraglets are rather unpractical to build data containers (at least without the '**equals**' instruction), they do make perfect semaphores.
Due to the fact that all list operations from the previous paragraph are handled in multiple steps (since the addition of the **size** list), and because all three events triggering the individual operations (ACK, NAK and SEND) could occur at practically any time, some measures had to be taken to assure that only one of the operations can take place at a time.

The solution was very simple. Each operation was altered so that it assimilates the **[READY]**-fraglet before it continues, ie. '**match:READY**' has been inserted into the fraglet recognizing the interface-call header and after the last step of the operation has been completed, a new **[READY]**-fraglet is set free. This way, as long as there is exactly one **[READY]**-fraglet in the beginning, it acts as a 'natural semaphore'.

5. The Big Picture

And now, the moment we've all been waiting for - a diagram of the entire protocol.
To enhance readability, it was split up into three fractions, corresponding to the three basic operations, **ACK**, **NAK** and **SEND**:

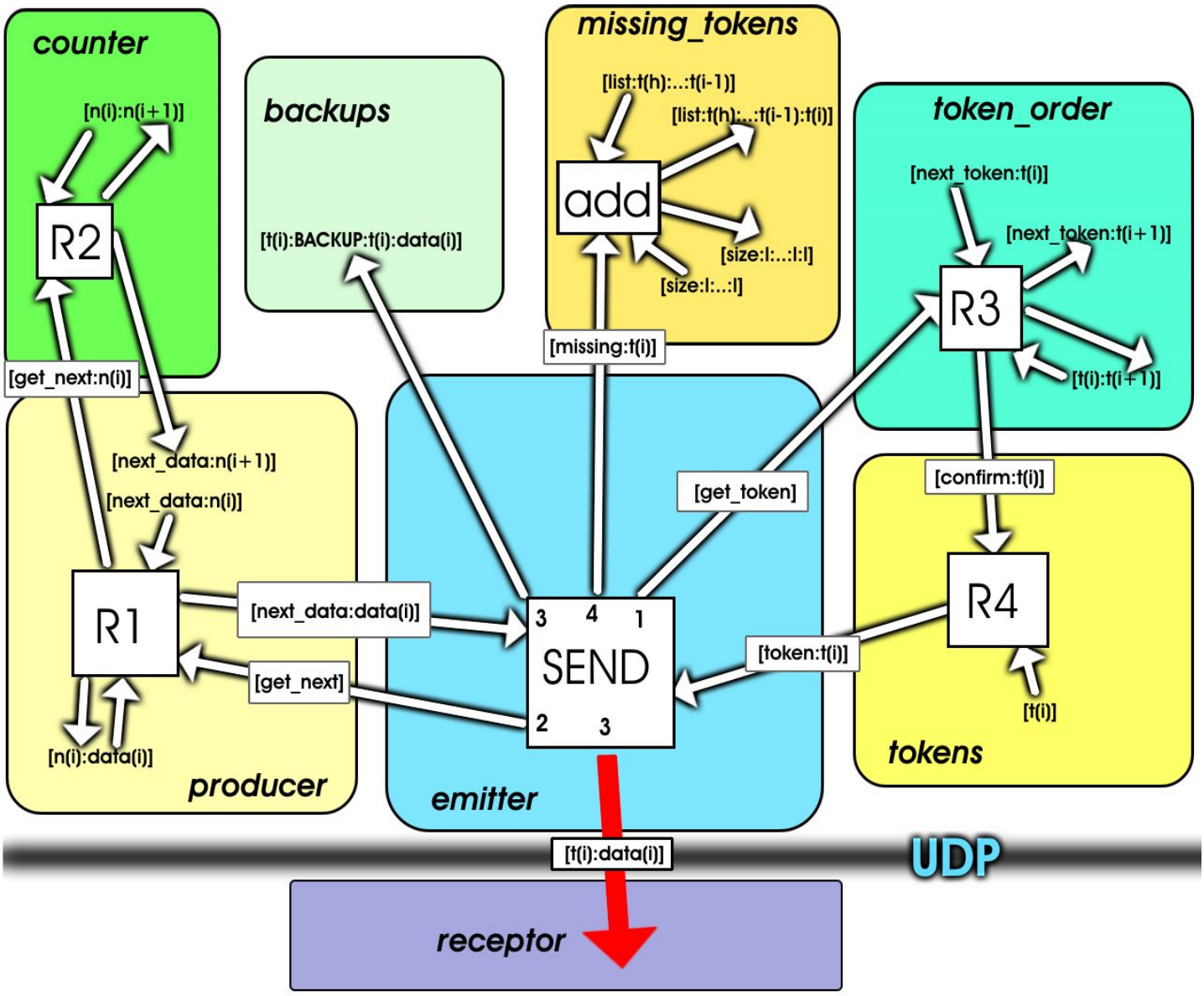
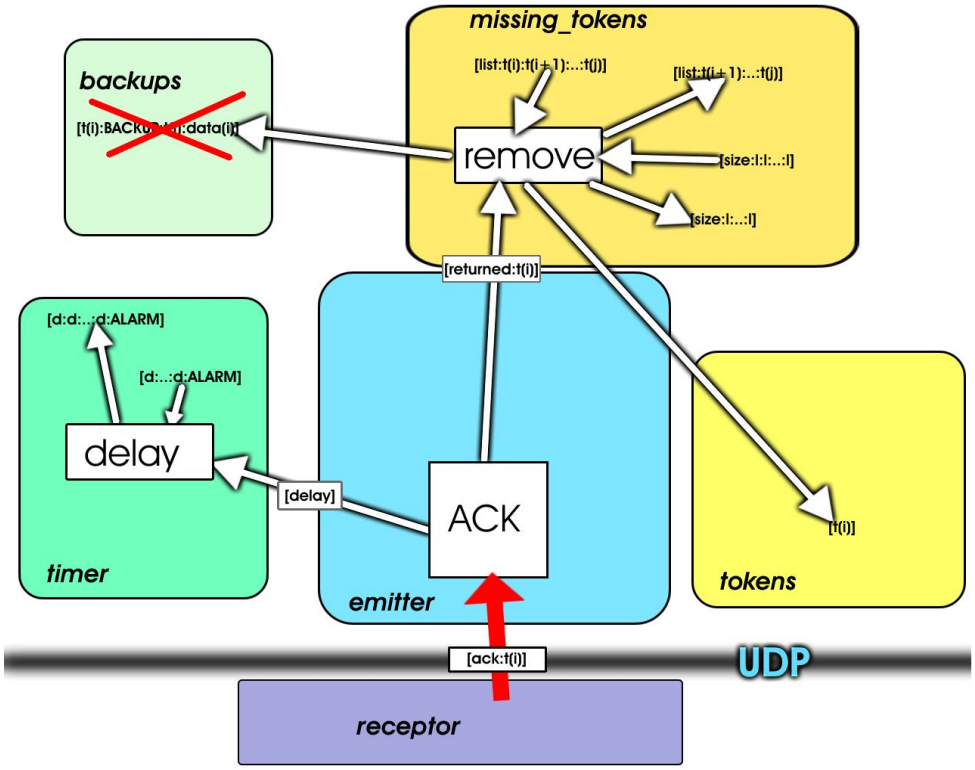
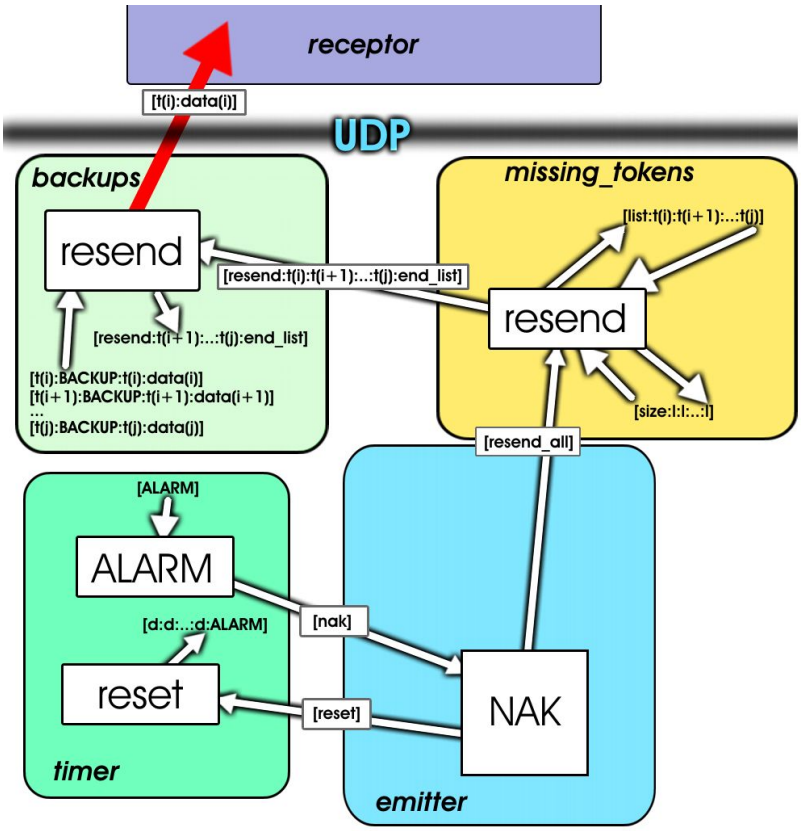


Image 5.1: reactions taking place when a fraglet is sent to the receptor.



Images 5.2 and 5.3: reactions taking place upon arrival of an ACK (above), and when the timer releases a NAK (below).



6. Receptor - The Central Flaw

So far, we have only explained the emitter side of the protocol. There are two reasons for that.
For one, except for its capability to expect a certain token and return an ack with that token when it arrives, it is rather simple. It does need two local nodes, but if we solved the far graver problem, this one would also disappear.
The other reason is a that it does not work as it is supposed to. It only expects one token, but that does not prevent it from receiving fraglets carrying others, so it simply ignores those, and they remain dormant. As the token set is circular, at a certain point, the tokens of the dormant fraglets become the expected ones. At that point, there is no way of forcing the system to favor a newly arrived fraglet over an older one. In fact, as the current fraglet with the expected token might need some time to arrive, the dormant ones are actually being preferred.
A very simple way out of this would be the introduction of the **equals** instruction.
*see section **proposals***

Then, we could simply put a header specific to our protocol in front of every fraglet we send, match that header on the receiver side, and either pick the fraglet or delete it, according to the equality between the token it carries and the expected token.
Using a specific header for incoming fraglets (ie. not a token), would then also remove the need for a second context (that is currently used to find the succeeding token to a specific token - see ***"multiple contexts, example 2"***).

Part Three:

Performance

Five tests have been performed. Each time, one instance of the fraglet environment was left to send data to another instance over loopback during a time interval of two minutes. The rate of packet loss was thereby altered between the tests (using parameter "-lose N").

The following table contains the results:

probability of packet loss	fraglets transmitted during two minutes	Number of instructions executed on all emitter nodes together
0	34	26240
1/ 5	29	27817
1/ 7	30	28384
1/10	23	18289
1/12	23	19976

An interesting effect is the drop in transmission performance at low packet loss probabilities. This can be accounted for by the bad calibration of our timer. The timeout-delay caused by an ACK is probably longer than the average time interval between two succeeding ACKs, so when only a small number of packets is lost, the timeout can grow indefinitely long. If a packet is lost under those conditions, the timeout we have to wait out until the packets are resent can grow very long very quickly.
The main reason for this assumption is the dropping number of instructions executed during the two minutes, which implies that nodes were idle for a part of the time.

Although we were only sending fraglets of the form **[tNN:dataN]** (size = 11 **Bytes**) between the nodes, in order to calculate the average transmission rate, we could just as well have sent packets of the size of one kilobyte.
A function (called **test_send_times(..)**) was introduced into the C-code to prove that the impact of packet size on the number of transmitted packets is insignificant to our measurements. The function first sends one million packets of the size of 3 **Bytes** and then one million packets of the size of 1024 **Bytes** and prints out the time needed for both procedures.
The times the function returns are the following: **3 Byte: 18 secs**, and **1024 Byte: 23 secs**. The difference are only five seconds over one million packets, or, in the average, five microseconds over one packet.

The maximal number of transmitted packets in our test series is **34**, so the time difference between sending 34 3-Byte-packets and 34 1-K-packets is about **170 microseconds**, which is only slightly more than **1.4 E-4 %** of the total time, and thus clearly negligible.

So, if we now assume that we *were* in fact sending packets of the size of one kilobyte (1K would still fit nicely into an IPv4 packet), our highest result, the one with no packet loss involved, would correspond to a transmission rate of about **0.283 K/sec**.

This is very low, but it can be greatly attributed to the low interpretation speed of our environment. In order to make a projection on how performance would look like on a theoretical fraglet-based architecture, another function has been introduced, **get_instruction_count( )**, that adds up all instruction counts of all nodes and sends the sum to the console.

Our prime example, the measurement without packet loss, executes roughly 26K instructions on the emitter during the two minutes, and it is probably never idle (each ack arrives even before the next of the thirteen tokens is away). Now if we imagine an architecture that could execute only **one million fraglet instructions per second**, it would allow us to execute the 26K instructions within about 26 milliseconds. Assuming that each outbound fraglet still carries a payload of one kilobyte, that would correspond to a transmission rate limit set at **1.3 megabytes per second**.

Then, of course, other factors such as the currently neglected packet size and, most of all bandwidth, would play the crucial role.

Part Four:

Proposals

1. the 'equals' instruction

During the design process, one feature was missed the most. Fraglets rely entirely on a positive logic. If a fraglet with a certain header is present, a reaction can be initiated - if it is not, the reaction *will* still be initiated once it arrives. There is no way of reacting to the negative outcome of a condition.

A great help in this matter would be an instruction with the following behaviour:

```
[ equals : s : t : x : y : tail ]

→   if (x == y) [ s : tail ]
    else [ t : tail ]
```

With this, we could easily make the **receptor** in our protocol delete fraglets carrying unexpected tokens. All we would need would be a common header to all fraglets we send to the **receptor**, and the following two fraglets:

Let the header be the symbol **DDP**, and the expected token **t(i)**

```
[match:DDP:equals:EXPECTED:nul:t(i)]
[matchp:EXPECTED: <do something with the right packet> ]
```

Another use could be found in **missing_tokens** in the reaction that removes a token returning with an ACK from the list of missing tokens. Currently, the protocol cannot cope with a lost ACK, because there is no way to know if the token we remove from the list really is the one that has been returned. Using the **equals** instruction, we could simply keep removing tokens from the list until we get the right one (and return them all to the list, if none of them is the right one - that is what would happen if two ACKs happened to arrive in the wrong sequence).

2. the 'spawn' instruction

Another instruction that could make a great difference would be one that can create a node either in the environment or the node where it is executed (the latter would require nodes to be able reside within other nodes).
The heavy use of multiple contexts demonstrated in our protocol could only then be made compatible with the **active networking** paradigm.

3. a graphical high-level fraglet language

Considering the very unorthodox nature of fraglets (compared to the widely popular programming languages), and the fact that 'copy-paste' commands are used much more frequently while writing fraglet programs, one could imagine a graphical environment that would allow the designer to compose large systems of fraglet-nodes in a very short amount of time.
Elementary reactions such as the duplicating of symbols (accounting for a major part of the code to this project) could then be combined to larger reactions (the program would take care of identifier-uniqueness). Those larger reactions would be symbolized by singular 'reaction objects', which in turn could be placed inside nodes.
The overall representation might look a little like the diagrams presented in this document.
The main benefit to this would be a faster assimilation of the fraglet idea by the public, thanks to increased accessibility - and therefore a greater probability of the concept taking root and evolving - possibly in symbiosis with the hardware industry.

Conclusions:

Even without the extensions to the instruction set proposed above, it is possible to create quite complex structures using fraglets. This has only been made possible by the use of multiple contexts, as they allow an isolated development of reactions in one singular node, so systems with a fairly large degree of complexity can be broken down into simpler pieces and worked out individually.
Some reactions, however, cannot be split up, and in those cases the programmer is suddenly confronted with the full amount of confusion inherent to a programming language based on biochemistry (see **lists**).
It is certainly in the interest of a programmer to work on such problems and waste as little time as possible on trivial tasks like duplications of symbols or the insertion of one specific symbol at one specific point in a fraglet, so I think the need for a higher level language is quite apparent - although lots of repetitive tasks can still be simplified by an extensive use of the 'copy-paste' functionality of any text editor.
As diagrams similar to the ones presented in this report were also used in the construction of the protocol itself, the idea of basing a higher level language entirely on such diagrams somehow does not seem that far fetched at all - especially considering that the fraglets paradigm emerged from a world fascinated by the accomplishments made in modern biochemistry, a field that can and does profit greatly from modern visualization techniques (although this applies mostly to 3D visualization techniques).
Of course, one could also easily imagine a purely text-based higher level language, but that would take away part of the popularity potential fraglets definitely exhibit so far.
Besides of that, the performance of the interpreter is still a little problematic, but I suppose something will be worked out in time.

References:

C. Tschudin, "Fraglets - a Metabolistic Execution Model for Communication Protocols",
Proceeding of AINS 2003, http://cn.cs.unibas.ch/pub/