APPENDICES

Appendix A

Examples of RS encoding and decoding

I. RS (255, 205) code: Text to be encoded

By the same showing, the Queen herself can have seen only a head. But if it is possible from a hand to deduce a body, informed with all the attributes of a great Queen, her crabbedness, courage, frailty, a

msg_in:


Code word:

Received word: 


Recovered message word:

Decoded text:

*By the same showing, the Queen herself can have seen only a head. But if it is possible from a hand to deduce a body, informed with all the attributes of a great Queen, her crabbedness, courage, frailty, a*

II. **Systematic and non-systematic forms of coding**

// ‘python’ program running sequence is reproduced below.

*** coloured items are outputs.

*** coloured items are my comments by way of clarification.

```
trp@trp-Veriton-Series:~$ python3
Python 3.4.2 (default, Oct 30 2014, 15:27:09)
[GCC 4.8.2] on linux
Type "help", "copyright", "credits" or "license" for more information.

>>> m = [3,7,11,15,19,23,27,1,5,9,13] #Message of 11 5-bit symbols (4th degree polynomials)
>>> import rsc31
>>> c = rsc31.rs_encode_msg(m,20) #Message is encoded into 31 5-bit symbols - systematic encoding

A. **Systematic form**

```

```
```python
>>> d
[3, 7, 11, 15, 19, 23, 27, 1, 5, 9, 13, 25, 23, 8, 5, 20, 6, 16, 21, 0, 6, 1, 31, 15, 6, 24, 11, 25, 4, 17]

>>> e = [0]*31

>>> e
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> e[0], e[2], e[4], e[6], e[8], e[10], e[12], e[14] = 3, 4, 5, 6, 7, 8, 9, 10

>>> e # Error polynomial with 8 errors
[3, 0, 4, 0, 5, 0, 6, 0, 7, 0, 8, 0, 9, 0, 10, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> r = [0]*31

>>> for i in range(len(c)): r[i] = c[i]**e[i] # Error 'added to c(x)' to get r(x)

...  

>>> r
[0, 7, 15, 15, 22, 23, 29, 1, 2, 9, 5, 25, 30, 8, 15, 5, 20, 6, 16, 21, 0, 6, 1, 31, 15, 6, 24, 11, 25, 4, 17]

>>> d1 = rsc31.rs_correct_msg(r, 20) # Erroneous received polynomial decoded

>>> d1
[3, 7, 11, 15, 19, 23, 27, 1, 5, 9, 13, 25, 23, 8, 5, 20, 6, 16, 21, 0, 6, 1, 31, 15, 6, 24, 11, 25, 4, 17]

>>> ec = [0]*31

>>> ec
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> for i in range(len(c)): ec[i] = d1[i]**c[i] # Check that c(x) & d1(x) are identical

...  

>>> ec
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> e[16], e[18] = 4, 5 # Two more errors added to make a total of 10 errors

>>> e
[3, 0, 4, 0, 5, 0, 6, 0, 7, 0, 8, 0, 9, 0, 10, 0, 4, 0, 5, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> for i in range(len(c)): r[i] = c[i]**e[i] # r(x) with new e(x) formed

...```
>>> r
[0, 7, 15, 15, 22, 23, 29, 1, 2, 9, 5, 25, 30, 8, 15, 5, 16, 6, 21, 21, 0, 6, 1, 31, 15, 6, 24, 11, 25, 4, 17]
>>> d2 = rsc31.rs_correct_msg(r, 20)  #r(x) again decoded
>>> d2
[3, 7, 11, 15, 19, 23, 27, 1, 5, 9, 13, 25, 23, 8, 5, 5, 20, 6, 16, 21, 0, 6, 1, 31, 15, 6, 24, 11, 25, 4, 17]
>>> for i in range(len(c)): ec[i] = d2[i]^c[i]  #Check that decoded d2(x) is identical to c(x)
...

>>> ec
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> e[20] = 6  #Introduce 11th error
>>> e  #Form e(x) with 11 errors
[3, 0, 4, 0, 5, 0, 6, 0, 7, 0, 8, 0, 9, 0, 10, 0, 4, 0, 5, 0, 6, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
>>> for i in range(len(c)): r[i] = c[i]^e[i]  #Form r(x) with 11 errors
...

>>> r
[0, 7, 15, 15, 22, 23, 29, 1, 2, 9, 5, 25, 30, 8, 15, 5, 16, 6, 21, 21, 0, 6, 1, 31, 15, 6, 24, 11, 25, 4, 17]
>>> d3 = rsc31.rs_correct_msg(r, 20)  #Try decoding r(x) with 11 errors - Failure!
Error location failed
>>> d3  #Nothing returned!

B. Non-systematic form

>>> c1 = rsc31.rs_ns_encode(m, 20)  #Encode message in non-systematic form
>>> c1
[3, 10, 22, 29, 3, 12, 30, 5, 30, 14, 12, 19, 27, 21, 23, 11, 15, 24, 4, 2, 9, 2, 29, 0, 12, 6, 1, 30, 14, 25, 14]
>>> d4 = rsc31.rs_correct_msg(c1, 20)  #Decode c1(x) itself - No errors
No errors

>>> d4
[3, 10, 22, 29, 3, 12, 30, 5, 30, 14, 12, 19, 27, 21, 23, 11, 15, 24, 4, 2, 9, 2, 29, 0, 12, 6, 1, 30, 14, 25, 14]

>>> en = [0]*31
>>> en
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> en[30], en[28], en[26], en[24], en[22], en[20], en[18], en[16], en[14], en[12] = 20, 19, 17, 16, 15, 14, 12, 11

>>> en #Form en(x) with 10 errors
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 11, 0, 12, 0, 13, 0, 14, 0, 15, 0, 16, 0, 17, 0, 18, 0, 19, 0, 20]

>>> for i in range(len(c1)): r[i] = c1[i]**en[i] #Combine en(x) with c1(x) to form r(x)

>>> r
[3, 10, 22, 29, 3, 12, 30, 5, 30, 14, 12, 19, 27, 21, 23, 11, 2, 24, 10, 2, 6, 2, 13, 0, 29, 6, 19, 30, 29, 25, 26]

>>> d5 = rsc31.rs_correct_msg(r, 20) #Decode r(X)

>>> d5
[3, 10, 22, 29, 3, 12, 30, 5, 30, 14, 12, 19, 27, 21, 23, 11, 15, 24, 4, 2, 9, 2, 29, 0, 12, 6, 1, 30, 14, 25, 14]

>>> for i in range(len(c)): ec[i] = d5[i]**c1[i] #Confirm d5(x) is same as c1(x)

>>> ec
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> en[1] = 22 #Introduce 11th error

>>> en
[0, 22, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]

>>> for i in range(len(c)): r[i] = c1[i]**en[i] #Form new r(x) with 11 errors

>>> r
[3, 10, 22, 29, 3, 12, 30, 5, 30, 14, 12, 19, 27, 21, 23, 11, 15, 24, 4, 2, 9, 2, 29, 0, 12, 6, 1, 30, 14, 25, 14]
[3, 28, 22, 29, 3, 12, 30, 5, 30, 14, 12, 19, 16, 21, 27, 11, 2, 24, 10, 2, 6, 2, 13, 0, 29, 6, 19, 30, 29, 25, 26]

```python
>>> d6 = rsc31.rs_correct_msg(r, 20) # Error correction effort fails!
Error location failed
>>> d6 # Nothing is returned!
```

```python
>>> mr = rsc31.rs_msg_corr(d5, 20) # Recover message from Corrected code - in non-systematic case

>>> mr
[3, 7, 11, 15, 19, 23, 27, 1, 5, 9, 13]
>>> me = [0] * len(m)

>>> me
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]"
Appendix B

IDA vs ECC dispersal

A few lines on the (subtle) differences between the use of IDA and RS code-based ECC for DSS is in order here. The difference is well brought out by comparing the use of \((n,k)\) code and the expansion by \((n/k)\) in IDA – as below:

- In both cases the storage can be done with \(k + \delta\) servers as long as \(0 < \delta < n - k\).
- Similarly, data retrieval can be done with any set of \(k\) fragments supplied by any of the \(k\) servers out of the existing lot. The retrieval efforts are on par in both the cases.
- The above is true as long as the file fragments supplied by the servers are not corrupted / erroneous.
- In case of erroneous supply of file fragments by servers, ECC can retrieve / reconstruct the correct file as long as the number of erroneous fragments is \((n - k)/2\). Here, the retrieval is effected through direct HDD. In case the number exceeds \((n - k)/2\), ECC declares the file as irretrievable.
- If erroneous fragments are present, file reconstruction / retrieval is much more cumbersome with IDA. One does not know which are the erroneous fragments.
  - One possible approach is to retrieve all possible 'candidate' files by using all possible sets of \(k\) fragments, compare them, and declare the 'majority' as the correct file.
    In the worst case this requires \(\binom{n}{k}\) sets of file reconstructions to be carried out.
    Obviously this is not a practical proposition.
  - An alternative approach is to use cryptographic primitives to construct pre-computed tokens to identify correct fragments for reconstruction.

In short, RS code-based ECC is a more optimal candidate for data retrieval with errors compared to IDA for DSS.
Appendix C

List decoding example

SDD of (7, 5) RS code

Transmitted codeword: $\alpha^3 x + \alpha^3 x^2 + 0 x^3 + x^4 + \alpha^2 x^5 + \alpha^5 x^6$

Received word: $\alpha^3 x + \alpha^3 x^2 + x^3 + x^4 + \alpha^2 x^5$ (with 2 symbol errors at symbol positions $r_3$ and $r_6$).

Some SDD algorithms use an intermediate step to form a ‘short list’ of prospective candidate code words and from that identify the most probable one as the decoded codeword. With the example considered here the SDD procedure used (Yamuna, B., and Padmanabhan, T. R., 2012) identified the most likely error polynomial to be of the form $r_1 x + r_3 x^3 + r_6 x^6$. There are only eight possible error patterns conforming to this constraint. These and the corresponding codewords are given in Table C.1. At the next step the most likely code word is identified from this list based on some likelihood measure.

Table C.1. List of the most reliable codewords nearest to the received word

<table>
<thead>
<tr>
<th>Sl. No.</th>
<th>Number of errors</th>
<th>Error pattern $e(x)$</th>
<th>Codewords identified</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3 errors</td>
<td>$\alpha^3 x + \alpha^6 x^3 + \alpha^6 x^6$</td>
<td>$\alpha^6 x^6 + \alpha^2 x^5 + x^4 + \alpha^2 x^3 + \alpha^3 x^2$</td>
</tr>
<tr>
<td>2</td>
<td>3 errors</td>
<td>$\alpha x + 0 x^3 + \alpha x^6$</td>
<td>$\alpha x^6 + \alpha^2 x^5 + x^4 + \alpha^2 x^3 + x^2$</td>
</tr>
<tr>
<td>3</td>
<td>3 errors</td>
<td>$x + \alpha^2 x^5 + 0 x^6$</td>
<td>$0 x^6 + \alpha^2 x^5 + x^4 + \alpha^3 x^2 + \alpha x$</td>
</tr>
<tr>
<td>4</td>
<td>3 errors</td>
<td>$\alpha^5 x + \alpha^5 x^3 + \alpha^2 x^6$</td>
<td>$\alpha^5 x^5 + \alpha^2 x^5 + x^4 + \alpha^4 x^3 + \alpha^3 x^2 + \alpha^2 x$</td>
</tr>
<tr>
<td>5</td>
<td>2 errors</td>
<td>$\alpha^5 x^6 + x^3$</td>
<td>$\alpha^5 x^6 + \alpha^2 x^5 + x^4 + 0 x^3 + \alpha^3 x^2 + \alpha^3 x$</td>
</tr>
<tr>
<td>6</td>
<td>3 errors</td>
<td>$\alpha^6 x + \alpha^4 x^3 + x^6$</td>
<td>$x^6 + \alpha^2 x^5 + x^4 + \alpha^3 x^2 + \alpha^3 x^2 + \alpha^5 x$</td>
</tr>
<tr>
<td>7</td>
<td>3 errors</td>
<td>$\alpha^2 x + \alpha^3 x^3 + \alpha^4 x^6$</td>
<td>$\alpha^6 x^6 + \alpha^2 x^5 + x^4 + \alpha x^3 + \alpha^3 x^2 + \alpha^2 x$</td>
</tr>
<tr>
<td>8</td>
<td>3 errors</td>
<td>$\alpha^2 x + \alpha x^3 + \alpha^3 x^6$</td>
<td>$\alpha^2 x^6 + \alpha^2 x^5 + x^4 + \alpha^3 x^2 + \alpha x^2 + \alpha^3 x$</td>
</tr>
</tbody>
</table>
Appendix D

File version no. using logical clock

In DSS the stored entity can be the full file, a file share, or the same in encoded form. Associating a logical clock value to the data aids consistent reads in servicing. The logical clock is a serial number representing the version number of the entity conforming to the following:

- each server / node in the scheme storing an entity has an associated logical clock and a clock value; it is a serial number.

- At commencement all logical clock values are initialized to zero.

- Whenever an authorized / identified agency accesses a server / node for a write or update of a file it collects the logical clock values from all participants, identifies the largest amongst them – \( N_m \) – and uses that as the latest version number of the file concerned. The corresponding file is identified / reconstructed with this \( N_m \) as the pivot.

- At rewriting the entity value the logical clock value is updated as \( N_m + 1 \) at all the participating nodes / servers. If necessary the same is propagated to the non-participants as well – as required by the scheme architecture.

The logical clock in [38] suffices for the work presented here. However, more sophisticated ones (vector clock) are also available.
PUBLICATIONS

I. International Journals


II. National Conference


III. Book / Book chapter


II. Papers Communicated
