The mathematics of RAID-6

H. Peter Anvin <*hpa@zytor.com*>

First version 20 January 2004
Last updated 21 May 2009

RAID-6 supports losing any two drives. The way this is done is by computing two syndromes, generally referred **P** and **Q**.

1 A quick summary of Galois field algebra

The algebra used for this is the algebra of a Galois field, $\mathbf{GF}(2^8)$. A smaller or larger field could also be used, however, a smaller field would limit the number of drives possible, and a larger field would require extremely large tables.

$\mathbf{GF}(2^8)$ allows for a maximum of 257 drives, 255 ($2^8 - 1$) of which can be data drives; the reason for this is shown below.

The *representation* of $\mathbf{GF}(2^8)$ used is the same one as used by the Rijndael (AES) cryptosystem. It has the following properties; this is not, however, an exhaustive list nor a formal derivation of these properties; for more in-depth coverage see any textbook on group and ring theory.

Note: A number in {} is a Galois field element (i.e. a byte) in hexadecimal representation; a number without {} is a conventional integer.

1. The *addition* field operator (+) is represented by bitwise XOR.
2. As a result, addition and subtraction are the same operation: $A + B = A - B$.
3. The additive identity element (0) is represented by {00}.
4. Thus, $A + A = A - A = \{00\}$.
5. *Multiplication* ($\cdot$) by {02} is implemented by the following bitwise relations:

$$
\begin{aligned}
(x \cdot \{02\})_7 &= x_6 \\
(x \cdot \{02\})_6 &= x_5 \\
(x \cdot \{02\})_5 &= x_4 \\
(x \cdot \{02\})_4 &= x_3 + x_7 \\
(x \cdot \{02\})_3 &= x_2 + x_7 \\
(x \cdot \{02\})_2 &= x_1 + x_7 \\
(x \cdot \{02\})_1 &= x_0 \\
(x \cdot \{02\})_0 &= x_7
\end{aligned}
$$

Hardware engineers will recognize this as a linear feedback shift register (LFSR), and matematicians as boolean polynomial multiplication modulo the irreducible polynomial $x^8 + x^4 + x^3 + x^2 + 1$.

6. The multiplicative identity element (1) is represented by $\{01\}$.

$$A \cdot \{01\} = \{01\} \cdot A = A$$

7. The following basic rules of algebra apply:

Addition is commutative:	$A + B$	$=$	$B + A$
Addition is associative:	$(A + B) + C$	$=$	$A + (B + C)$
Multiplication is commutative:	$A \cdot B$	$=$	$B \cdot A$
Multiplication is associative:	$(A \cdot B) \cdot C$	$=$	$A \cdot (B \cdot C)$
Distributive law:	$(A + B) \cdot C$	$=$	$A \cdot C + B \cdot C$

8. For every $A \neq \{00\}$, there exists an element A^{-1} such that $A \cdot A^{-1} = \{01\}$. A^{-1} is called the *inverse* (or *reciprocal*) of A. $\{01\}$ is its own inverse, $\{00\}$ lacks inverse, for all other A, $A^{-1} \neq A$.

9. Division is defined as multiplication with an inverse:

$$A/B = A \cdot B^{-1}$$

Any nonzero element can uniquely divide any element:

If $A \cdot B = C$ then $C/B = A$ for any $B \neq \{00\}$.

In particular, $A/A = A \cdot A^{-1} = \{01\}$ for any $A \neq \{00\}$.

10. Multiplying by zero is zero:

$$A \cdot \{00\} = \{00\}$$

11. Any value can be multiplied by observing that bits decompose the same as in ordinary arithmetic, and applying the distributive law:

$$\begin{array}{lcl} \{02\}^2 & = & \{02\} \cdot \{02\} = \{04\} \\ \{02\}^3 & = & \{04\} \cdot \{02\} = \{08\} \\ \{02\}^4 & = & \{08\} \cdot \{02\} = \{10\} \\ \{02\}^5 & = & \{10\} \cdot \{02\} = \{20\} \\ \{02\}^6 & = & \{20\} \cdot \{02\} = \{40\} \\ \{02\}^7 & = & \{40\} \cdot \{02\} = \{80\} \end{array}$$

(Note, however: $\{02\}^8 = \{1d\}$.)

For example:

$$\begin{array}{rcl} \{8d\} & = & \{80\} + \{08\} + \{04\} + \{01\} \\ & = & \{02\}^7 + \{02\}^3 + \{02\}^2 + \{01\} \end{array}$$

Thus:

$$A \cdot \{8d\} = A \cdot \{02\}^7 + A \cdot \{02\}^3 + A \cdot \{02\}^2 + A$$

or, equivalently,

$$A \cdot \{8d\} = (((A \cdot \{02\}^4) + A) \cdot \{02\} + A) \cdot \{02\}^2 + A$$

12. *Raising to a power* (repeated multiplication with the same value) is congruent mod 255 (cardinality of all elements except $\{00\}$). Also note that the exponent is an *ordinary integer modulo 255* (an element in $\mathbb{Z}_{255}$) as opposed to a Galois field element.

$$\left.\begin{array}{lcl} A^{256} & = & \{01\} \cdot A = A \\ A^{255} & = & \{01\} \\ A^{254} & = & A^{255}/A = \{01\}/A = A^{-1} \end{array}\right\} A \neq \{00\}$$

13. There are elements (g), called *generators*, of the field such that g^n doesn't repeat until they have exhausted all elements of the field except $\{00\}$. For the AES field representation, $\{02\}$ is such a generator – as is $\{02\}^n$ for any n which is relative prime to 255.

14. Accordingly, any generator g defines a function from the nonzero elements in $\mathbf{GF}(2^8)$ to the elements in $\mathbb{Z}_{255}$ (the integers 0-254 modulo 255) called the *logarithm with base* g and written $\log_g$. For example, $\{02\}^4 = \{10\}$, so $\log_{\{02\}} \{10\} = 4$.

15. For any nonzero Galois field elements A and B:

$$\begin{aligned} A \cdot B = C &\iff \log_g A \oplus \log_g B = \log_g C \\ A/B = C &\iff \log_g A \ominus \log_g B = \log_g C \end{aligned}$$

... where $\oplus$ and $\ominus$ represents conventional integer addition and subtraction modulo 255. Therefore:

$$\begin{aligned} A \cdot B = C &\iff C = g^{(\log_g A \oplus \log_g B)} \\ A/B = C &\iff C = g^{(\log_g A \ominus \log_g B)} \end{aligned}$$

These relations can be used to do multiplication and division without large tables, as long as $\{00\}$ is handled specially.

2 Application to RAID-6

We treat each disk block as a vector of bytes, and will perform the same calculations on each byte in the vector. Symbols in **boldface** represent vectors (where each byte has a different value); constants, or symbols in *italics* represent scalars (same value across every data byte.)

In order to be able to suffer the loss of any two disks, we need to compute two *syndromes*, here referred to as $\mathbf{P}$ and $\mathbf{Q}$.

For n data disks $\mathbf{D}_0$, $\mathbf{D}_1$, $\mathbf{D}_2$, ... $\mathbf{D}_{n-1}$ ($n \leq 255$) compute:

$$\mathbf{P} = \mathbf{D}_0 + \mathbf{D}_1 + \mathbf{D}_2 + ... + \mathbf{D}_{n-1} \quad (1)$$

$$\mathbf{Q} = g^0 \cdot \mathbf{D}_0 + g^1 \cdot \mathbf{D}_1 + g^2 \cdot \mathbf{D}_2 + ... + g^{n-1} \cdot \mathbf{D}_{n-1} \quad (2)$$

where g is any generator of the field (we use $g = \{02\}$.)

$\mathbf{P}$ is the ordinary XOR parity, since "addition" is XOR. $\mathbf{Q}$ is referred to as a Reed-Solomon code.

If we lose one data drive, we can use the normal XOR parity to recover the failed drive data, just as we would do for RAID-5. If we lose a non-data drive, i.e. $\mathbf{P}$ or $\mathbf{Q}$, then we can just recompute.

If we lose one data drive plus the $\mathbf{Q}$ drive, we can recalculate the data drive using the XOR parity, and then recompute the $\mathbf{Q}$ drive.

If we lose one data drive plus the $\mathbf{P}$ drive, we can recompute the lost data drive ($\mathbf{D}_x$) from the $\mathbf{Q}$ drive by computing $\mathbf{Q}_x$ as if $\mathbf{D}_x = \{00\}$, and observing:

$$\mathbf{Q}_x + g^x \cdot \mathbf{D}_x = \mathbf{Q} \quad (3)$$

Here, x, $\mathbf{Q}$ and $\mathbf{Q}_x$ are known. Since addition and subtraction is the same:

$$g^x \cdot \mathbf{D}_x = \mathbf{Q} + \mathbf{Q}_x \quad (4)$$

$$\mathbf{D}_x = (\mathbf{Q} + \mathbf{Q}_x)/g^x = (\mathbf{Q} + \mathbf{Q}_x) \cdot g^{-x} \quad (5)$$

where, per the algebra rules, $g^{-x} = g^{255-x}$.

If we lose two data drives, $\mathbf{D}_x$ and $\mathbf{D}_y$, but still have the $\mathbf{P}$ and $\mathbf{Q}$ values, we compute $\mathbf{P}_{xy}$ and $\mathbf{Q}_{xy}$ by setting the missing drives to $\{00\}$, and we get:

$$\begin{aligned} \mathbf{P}_{xy} + \mathbf{D}_x + \mathbf{D}_y &= \mathbf{P} \quad (6) \\ \mathbf{Q}_{xy} + g^x \cdot \mathbf{D}_x + g^y \cdot \mathbf{D}_y &= \mathbf{Q} \quad (7) \end{aligned}$$

x, y, $\mathbf{P}$, $\mathbf{P}_{xy}$, $\mathbf{Q}$ and $\mathbf{Q}_{xy}$ are known.

Divide the second equation by g^x:

$$g^{-x} \cdot \mathbf{Q}_{xy} + \mathbf{D}_x + g^{y-x} \cdot \mathbf{D}_y = g^{-x} \cdot \mathbf{Q} \quad (8)$$

Remembering that addition equals subtraction in this algebra:

$$\mathbf{D}_x + g^{y-x} \cdot \mathbf{D}_y = g^{-x} \cdot \mathbf{Q} + g^{-x} \cdot \mathbf{Q}_{xy} \quad (9)$$

$$\mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot \mathbf{D}_y \quad (10)$$

Substitute into the first equation, solve for $\mathbf{D}_y$:

$$\mathbf{D}_y = \mathbf{P} + \mathbf{P}_{xy} + \mathbf{D}_x \quad (11)$$

$$\mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy} + \mathbf{D}_x) \quad (12)$$

$$\mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy}) + g^{y-x} \cdot \mathbf{D}_x \quad (13)$$

$$\mathbf{D}_x + g^{y-x} \cdot \mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy}) \quad (14)$$

$$(g^{y-x} + \{01\}) \cdot \mathbf{D}_x = g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy}) \quad (15)$$

If $g^{y-x} + \{01\} \neq \{00\}$, we can divide by it. This requires $g^{y-x} \neq \{01\}$; this will be true as long as $y \neq x$, mod 255. Since we can have no more than 255 data disks, $0 \leq x, y \leq n - 1 < 255$, this implies the only constraint is $y \neq x$, which is true by assumption. Thus, we can divide:

$$\mathbf{D}_x = \frac{g^{-x} \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) + g^{y-x} \cdot (\mathbf{P} + \mathbf{P}_{xy})}{g^{y-x} + \{01\}} \quad (16)$$

For any particular data reconstruction, we can simplify this by precomputing a few multiplication tables:

$$A = \frac{g^{y-x}}{g^{y-x} + \{01\}} = g^{y-x} \cdot (g^{y-x} + \{01\})^{-1} \quad (17)$$

$$B = \frac{g^{-x}}{g^{y-x} + \{01\}} = g^{-x} \cdot (g^{y-x} + \{01\})^{-1} \quad (18)$$

... which only depend on x and y as opposed to on the data bytes.
The expression then becomes:

$$\mathbf{D}_x = A \cdot (\mathbf{P} + \mathbf{P}_{xy}) + B \cdot (\mathbf{Q} + \mathbf{Q}_{xy}) \quad (19)$$

We can then get $\mathbf{D}_y$ from the previous expression:

$$\mathbf{D}_y = (\mathbf{P} + \mathbf{P}_{xy}) + \mathbf{D}_x \quad (20)$$

3 Making it go fast

The biggest problem with RAID-6 has historically been the high CPU cost of computing the **Q** syndrome. The biggest cost is related to the cost of Galois field multiplication, which doesn't map conveniently onto standard CPU hardware, and therefore has typically been done by table lookup.

Table lookups, however, are inherently serializing; it would be desirable to make use of the wide datapaths of current CPUs.

In order to do this, we factor equation 2 as such:

$$\mathbf{Q} = ((...\mathbf{D}_{n-1}...) \cdot g + \mathbf{D}_2) \cdot g + \mathbf{D}_1) \cdot g + \mathbf{D}_0 \quad (21)$$

The only operations in this is addition, i.e. XOR, and multiplication by $g = \{02\}$. Thus, we only need an efficient way to implement multiplication by $\{02\}$ in order to compute **Q** quickly, not arbitrary multiplication.

Multiplication by $\{02\}$ for a single byte can be implemeted using the C code:

```
uint8_t c, cc;
cc = (c << 1) ^ ((c & 0x80) ? 0x1d : 0);
```

Now, we want to do this on multiple bytes in parallel. Assume for the moment we are on a 32-bit machine (the extension to 64 bits should be obvious), and separate these into two parts:

```
uint32_t v, vv;

vv  = (v << 1) & 0xfefefefe;
vv ^= ((v & 0x00000080) ? 0x0000001d : 0) +
      ((v & 0x00008000) ? 0x00001d00 : 0) +
      ((v & 0x00800000) ? 0x001d0000 : 0) +
      ((v & 0x80000000) ? 0x1d000000 : 0);
```

The `0xfefefefe` of the first statement masks any bits that get shifted into the next byte. The second statement is clearly too complex to be efficiently executed, however. If we can produce a mask based on the top bit in each byte, we could just do:

```
uint32_t v, vv;

vv  = (v << 1) & 0xfefefefe;
vv ^= MASK(v) & 0x1d1d1d1d;
```

In standard portable C, one implemenation of this `MASK()` function looks like:

```
uint32_t MASK(uint32_t v)
{
  v &= 0x80808080;              /* Extract the top bits */
  return (v << 1) - (v >> 7);   /* Overflow on the top bit is OK */
}
```

The result is `0x00` for any byte with the top bit clear, `0xff` for any byte with the top bit set. This is the algorithm used in the file `raid6int.uc`.

For additional speed improvements, it is desirable to use any integer vector instruction set that happens to be available on the machine, such as MMX or SSE-2 on x86, AltiVec on PowerPC, etc. These instruction sets typically have quirks that may make them easier or harder to use than the integer implementation, but usually easier. For example, the MMX/SSE-2 instruction `PCMPGTB` conveniently implements the `MASK()` function when comparing against zero, and the `PADDB` instruction implements the shift and mask in the first line of the operations on `vv` when added with itself.

Note that none of this will avoid the arbitrary multiplications of equations 5 and 19. Thus, in 2-disk-degraded mode, performance will be very slow. However, it is expected that that will be a rare occurrence, and that performance will not matter significantly in that case.

3.1 Special notes on PowerPC AltiVec and x86 SSSE3

The Altivec SIMD vector instruction set for PowerPC has a special instruction, `vperm`, which does a parallel table lookup using the bottom five bits of each byte in a vector.

This can be used to handle arbitrary scalar · vector multiplication (as in equations 5 and 19) quickly, by decomposing the vector.

This decomposition is simply a matter of observing that, from the distributive law:

$$\begin{aligned} \mathbf{V} &= \mathbf{V}_a + \mathbf{V}_b \\ A \cdot \mathbf{V} &= A \cdot \mathbf{V}_a + A \cdot \mathbf{V}_b \end{aligned}$$

For the decomposition to work, there can only be 32 possible values for each byte in $\mathbf{V}_a$ or $\mathbf{V}_b$; the easiest such decomposition is simply $\mathbf{V}_a$ being the low four bits and $\mathbf{V}_b$ being the high four bits; since addition is XOR this is a valid decomposition, and there are only 16 possible values of each.

Thus, for each multiplication (i.e. value of A) we need to set up a pair of vector registers, one which contains ($A \cdot \{00\}$, $A \cdot \{01\}$, $A \cdot \{02\}$, ... $A \cdot \{0f\}$) and one which contains ($A \cdot \{00\}$, $A \cdot \{10\}$, $A \cdot \{20\}$, ... $A \cdot \{f0\}$).

If these vectors are in `v12` and `v13` respectively, and `v14` set up to contain ($\{04\}$, $\{04\}$, ...), we can compute `v1` $\leftarrow A \cdot$ `v0` this way:

```
vsrb  v1, v0, v14
vperm v2, v12, v12, v0
vperm v1, v13, v13, v1
vxor  v1, v2, v1
```

On most Altivec processors, this will execute in three cycles. Note that we don't actually need to mask the top bits for the first `vperm`; since we repeat `v12` twice we effectively ignore bit 4, and bits 5-7 are ignored by the hardware anyway.

The SSSE3 (Supplemental SSE3) extensions to the x86 instruction set includes a `PSHUFB` instruction, which can be used in a similar way. `PSHUFB` uses bit 7 as a control bit, which means that the lower half operation has to be masked; simply replicating the inputs will not help. Furthermore, since no `PSRAB` instruction exists, one also has to mask the high half. Thus, as above, with `xmm14` having the scalar constant $\{0f\}$:

```
movdqa  xmm2, xmm0
psraw   xmm0, 4
movdqa  xmm1, xmm12
movdqa  xmm3, xmm13
pand    xmm0, xmm14
pand    xmm2, xmm14
pshufb  xmm1, xmm0
pshufb  xmm3, xmm2
pxor    xmm1, xmm3
```

4 Single-disk corruption recovery

It is possible to use the RAID-6 syndrome set to recover from a single disk *corruption*, as opposed to one or two known failed drives (called *erasures*.)

This requires recomputation of the syndrome set on read. This can of course also be done as a periodic integrity check, or as recovery if corruption is known or believed.

To consider the case of a single corrupt disk, we first consider the case where the failed disk (z) is one of the data drives ($\mathbf{D}_z$). We will represent the corrupt data on that drive with $\mathbf{X}_z$. Obviously, the value z is unknown, although of course, by definition, $0 \leq z < n \leq 255$.

We compute the standard syndrome set over the corrupt disk set:

$$\mathbf{P}' = \mathbf{D}_0 + \mathbf{D}_1 + \ldots + \mathbf{X}_z + \ldots + \mathbf{D}_{n-1} \quad (22)$$

$$\mathbf{Q}' = g^0 \cdot \mathbf{D}_0 + g^1 \cdot \mathbf{D}_1 + \ldots + g^z \cdot \mathbf{X}_z + \ldots + g^{n-1} \cdot \mathbf{D}_{n-1} \quad (23)$$

It obviously follows that:

$$\mathbf{P}^\star = \mathbf{P} + \mathbf{P}' = \mathbf{D}_z + \mathbf{X}_z \quad (24)$$

$$\mathbf{Q}^\star = \mathbf{Q} + \mathbf{Q}' = g^z \cdot \mathbf{D}_z + g^z \cdot \mathbf{X}_z = g^z \cdot (\mathbf{D}_z + \mathbf{X}_z) = g^z \cdot \mathbf{P}^\star \quad (25)$$

By assumption, $\mathbf{X}_z \neq \mathbf{D}_z$ and thus $\mathbf{P}^\star \neq \{\mathtt{00}\}$. Furthermore, since $g^z \neq \{\mathtt{00}\}$ for any z, $\mathbf{Q}^\star \neq \{\mathtt{00}\}$.

Thus it it valid to state:

$$\mathbf{Q}^\star / \mathbf{P}^\star = g^z \quad (26)$$

Since $0 \leq z < n \leq 255$, it then follows:

$$z = \log_g(\mathbf{Q}^\star / \mathbf{P}^\star) = \log_g \mathbf{Q}^\star \ominus \log_g \mathbf{P}^\star \quad (27)$$

... which will be a well-defined relation for all possible values that fit the required assumptions.

As noted above, for the case of a corrupt data drive, $\mathbf{P}^\star \neq \{\mathtt{00}\}$, and $\mathbf{Q}^\star \neq \{\mathtt{00}\}$. The *other* possible cases can be trivially shown to result in various combinations which involve $\mathbf{P}^\star$ and/or $\mathbf{Q}^\star$ being zero:

	$\mathbf{P}^\star$	$\mathbf{Q}^\star$
No corruption	$= \{\mathtt{00}\}$	$= \{\mathtt{00}\}$
$\mathbf{P}$ drive corruption	$\neq \{\mathtt{00}\}$	$= \{\mathtt{00}\}$
$\mathbf{Q}$ drive corruption	$= \{\mathtt{00}\}$	$\neq \{\mathtt{00}\}$
Data drive corruption	$\neq \{\mathtt{00}\}$	$\neq \{\mathtt{00}\}$

or, equivalently:

	$\mathbf{P}'$	$\mathbf{Q}'$
No corruption	$\mathbf{P} = \mathbf{P}'$	$\mathbf{Q} = \mathbf{Q}'$
$\mathbf{P}$ drive corruption	$\mathbf{P} \neq \mathbf{P}'$	$\mathbf{Q} = \mathbf{Q}'$
$\mathbf{Q}$ drive corruption	$\mathbf{P} = \mathbf{P}'$	$\mathbf{Q} \neq \mathbf{Q}'$
Data drive corruption	$\mathbf{P} \neq \mathbf{P}'$	$\mathbf{Q} \neq \mathbf{Q}'$

Obviously, for the cases of $\mathbf{P}$ or $\mathbf{Q}$ drive corruption, just replace the corrupt data with the recomputed $\mathbf{P}'$ or $\mathbf{Q}'$, respectively. In the case of data drive corruption, once the faulty drive has been identified, recover using the $\mathbf{P}$ drive in the same way as a one-disk erasure failure.

It should be noted that although we have used scalar notation for the corrupt drive, data corruption is actually detected on a *byte by byte* basis. Thus, the zeroness tests should be done for each byte, and z in equation 27 really should be a vector result, $\mathbf{z}$. It it, of course, a quality of implementation issue whether or not it is possible to recover from multiple drives having non-overlapping corruption in corresponding sectors or blocks.

Finally, as a word of caution it should be noted that RAID-6 by itself cannot (in the general case) even detect, never mind recover from, dual-disk corruption. If two disks are corrupt in the same byte positions, the above algorithm will (again, in the general case) introduce *additional* data corruption by corrupting a third drive. However, the following probabilistic patterns are likely to be indicative of such multidisk corruption, and a quality implementation should take appropriate action, such as aborting rather than further corrupting data:

- z values inconsistent with the number of disks, for example $z = 136$ when $n = 20$.
- Inconsistent z values within a single hardware sector or block. This does not apply to occasional bytes with no corruption ($\mathbf{P}^\star = \mathbf{Q}^\star = \{00\}$) – after all, even a standing clock is correct once every 12 hours.

5 Beyond RAID-6

Reed-Solomon coding can be exploited further to allow for any combination of n data disks plus m redundancy disks allowing for any m failures to be recovered. However, with increasing amount of redundancy, the higher the overhead both in CPU time and I/O. The Linux RAID-6 work has been focused on handling the case of $m = 2$ efficiently in order for it to be practically useful.

An excellent paper on implementing arbitrarily complex recovery sets using Reed-Solomon coding can be found at:

`http://www.cs.utk.edu/~plank/plank/papers/CS-96-332.html`