Testing Equality of Strings

Testing equality of strings has a number of applications in networked computing environments. A common problem in a distributed environment is that multiple copies of documents exist in different places (e.g. in the cache of your web browser, on your PC's local disk etc). Any one of these copies may be modified, and then the others will need to be updated. But first you must find out whether the two documents are the still the same or not. Time stamps are a partial solution, but a more general technique is to use fingerprinting.

Suppose you have two documents in different parts of the network, and you would like to check if they are the same without sending an entire copy of one of them. Instead you can use a fingerprint. One way of computing fingerprints uses ideas you have already seen (testing polynomial identities).

Polynomial method 1

Let the two strings consist of n characters each (or there isn't much point in checking equality). Let one string $a = a_1 a_2 \ldots a_n$, and the other $b = b_1 b_2 \ldots b_n$. You can think of a string as a polynomial:

$$a(x) = \sum_{i=1}^{n} a_i x^i$$

And similarly for $b(x)$. Now you can check if $a(x) = b(x)$ by choosing a random r and computing $\alpha = a(r)$ and $\beta = b(r)$. You only need to transmit α or β across the network, and with a little more work we can make sure that they are much smaller than the strings a or b, which have n bytes.

As we have already seen, if r is chosen in the range $\{0, \ldots, M-1\}$ and if $a(x)$ and $b(x)$ are different, then the probability that $\alpha = \beta$ is at most n/M (n is the degree of the polynomials here). So by choosing $M > 2n$, we can be confident that we have a reasonable chance (better than even) of detecting a difference. And we can make the probability of not discovering a difference between a and b exponentially small by using more bits for r. Even if we pick r of say $1024n$, we need only $O(\log n)$ bits for r.

But then we run into a snag. Even though r itself has only $O(\log n)$ bits, the polynomials $a(x)$ and $b(x)$ include a term of the form x^n and when we compute the n^{th} power of r, we will get a number with $O(n \log n)$ bits. That's actually longer than the original strings.

The solution is to work over a finite field (modulo a prime) instead of over the integers. That is, we do all calculations $\bmod\ p$, which guarantees that we never need more than $O(\log p)$ bits. The value of $a(\bmod\ p)$ is the remainder when a is divided by p. Thus $a(\bmod\ p)$ is always in $\{0, \ldots, p-1\}$, so mod controls the size of expressions. The mod operation has the important property that it commutes with other arithmetic operations. That is, $a(\bmod\ p) \times b(\bmod\ p) = (ab)(\bmod\ p)$, etc. This means that we can use mod in the middle of calculations to reduce the size

of intermediate expressions, and the result is the same as if we had applied it only once at the end.

To figure out how large p needs to be, notice that there are only p distinct possible values for $r \pmod p$. So to get enough distinct choices for r to make the random choice part work, we need $p > M$.

Method 1: n = length of a or b

1. Let M be "somewhat larger" than n, say $1024n$ or more
2. Pick a prime $p > M$ (actually pick random numbers until one is prime).
3. Choose a random r from $\{0, \ldots, M-1\}$
4. Compute $\alpha = a(r) \pmod p$ and transmit α, p, r across the network (all $O(\log n)$ bits)
5. At other end of network, receive α, p, r and compare α with $\beta = b(r) \pmod p$

Which will detect a difference between a and b with probability at least $1 - n/M$. Note that since p doesn't need to change it does not need to be transmitted if the two parties agree on it ahead of time.

The running time depends on the time to compute $a(r) \pmod p$. If we assume each arithmetic step has unit cost, that takes only $O(n)$ time. But that is a little bit loose because each arithmetic step involves $O(\log n)$ bits. However, its rare to get strings with more than 2^{20} characters (1 MB), and 32 bits is plenty to hold the results for smaller strings, so its not unreasonable to make that assumption in practice.

Polynomial Method 2

This method is very similar to the first method. Only instead of using a fixed p and a randomly chosen r, we use a fixed r, and a randomly chosen prime p. First of all, we want to find an r such that $a(r)$ and $b(r)$ are always different if a and b are different strings. Assuming each string contains 8-bit bytes, then it suffices to take $r = 256 = 2^8$. Why?: Prove this for yourself.

Now we compute $\alpha = a(256) \pmod p$ and transmit it. The recipient compares it to $\beta = b(256) \pmod p$. Those two will be the same if $\alpha = \beta \pmod p$, that is, if $(\alpha - \beta)$ is divisible by p.

Now if we choose p from a "large enough set" then the probability that $(\alpha - \beta)$ is divisible by p will be low. To specify how large a set, we need to know a little more about prime numbers:

Prime Number Theorem Let $\pi(k) =$ number of distinct prime numbers less than k. Then

$$\pi(k) \approx k / \ln k$$

where the approximation symbol $\approx$ means "asymptotically approaches", i.e.

$$\lim_{k \to \infty} \frac{\pi(k)}{(k / \ln k)} = 1$$

Corollary: Let N be the product of the first m distinct primes, then $m < 2\ln N/\ln\ln N$.

Proof: Let k be the median prime factor of N. Half the factors are bigger than k, so

$$N > k^{m/2} \quad \text{or} \quad (m/2)\ln k < (\ln N)$$

Now half the factors are smaller than k, so

$$m/2 = \pi(k) \approx k/\ln k$$

and substituting $m/2 \approx k/\ln k$ in the inequality above it

$$(k/\ln k)\ln k < \ln N \quad \text{or} \quad k < \ln N$$

and then substituting this bound on k in the formula $m/2 \approx k/\ln k$ gives:

$$m < 2\ln N/\ln\ln N \qquad \text{QED}$$

Note: the identity $m < 2\ln N/\ln\ln N$ holds for any product of m distinct primes, because they and their product will be larger than for the first m primes. Going back to the string comparison, if $\alpha - \beta = N$, then N has at most $8n$ bits for n-character string comparisons. The maximum number of factors of N is $2\ln N/\ln\ln N$, or $O(n/\log n)$.

When we choose a prime p, we will fail if p divides $\alpha - \beta$. By the above argument, there are at most $O(n/\log n)$ bad choices for p. So we should pick p in a range $\{2,\ldots,M\}$ which contains substantially more than $n/\log n$ primes. i.e. we want

$$\pi(M) \approx M/\ln M >> n/\log n$$

and choosing $M = \Omega(n)$ or say $1000n$ will do this.

Method 2: n = length of a or b

1. Let M be "somewhat larger" than n, say $1000n$ or more.
2. Pick a random prime p from $\{2,\ldots,M\}$.
3. Compute $\alpha = a(256)(\mathrm{mod}\ \ p)$ and transmit α, p across the network (both $O(\log n)$ bits).
4. At other end of network, receive α, p and compare α with $\beta = b(256)(\mathrm{mod}\ \ p)$.

So to contrast the two methods: both compare $a(r)(\mathrm{mod}\ \ p)$ and $b(r)(\mathrm{mod}\ \ p)$. Method 1 uses a fixed p and a random r with $O(\log n)$ bits each. Method 2 uses a fixed $r = 256$, and a random p with $O(\log n)$ bits. Computing the primes for method 2 involves generating random numbers and then testing them for primality. This takes time polynomial in $\log n$, but the constant is high in practice. So instead it is better to precompute some primes, and pick one at random. For n-character strings, you would need $O(n/\log n)$ primes on the source machine.