From the EDVAC to WEBVACs

Cloud computing for computer scientists

Daniel C. Wang

By now everyone has heard of cloud computing and realized that it is changing how both traditional enterprise IT and emerging startups are building solutions for the future. Is this trend toward the cloud just a shift in the complicated economics of the hardware and software industry, or is it a fundamentally different way of thinking about computing? Having worked in the industry, I can confidently say it is both.

Most articles on cloud computing focus too much on the economic aspects of the shift and miss the fundamental changes in thinking. This article attempts to fill the gap and help a wider audience better appreciate some of the more fundamental issues related to cloud computing. Much of what is written here should not be earth-shattering to people who work with these systems day to day, but the article may encourage even expert practitioners to look at their day-to-day issues in a more nuanced way.

Here are the key points to be covered:

- A cloud computer is composed of so many components that systematically dealing with the failure of those components is an essential factor to consider when thinking about the software that runs on a cloud computer or interacts with one.
- A common architectural pattern used to deal with transient failures is to divide the system into a pure computational layer and a separate layer that maintains critical system state. This provides reliability, scalability, and simplicity.
- A well-established existing best practice is to have systems expose idempotent interfaces so simple retry logic can be used to mask most transient failures.
- Simple analytic techniques can allow quantitative statements about various retry policies and compare how they impact reliability, worst-case latency, and average-case latency under an idealized failure model.

The first point about dealing with failure may seem new to many who are now hosting even small applications on large multitenant cloud computers in order to benefit from economies of scale. This is actually a very old issue, however, so the discussion should begin not by talking about the latest trends but by going back to the early years of the electronic computer.

LESSONS FROM EDVAC AND VON NEUMANN

In 1945 John von Neumann described the computational model of the first fully electronic stored program computer. This was a side effect of his acting as a consultant with ENIAC inventors John Mauchly and J. Presper Eckert. Although von Neumann was not the originator of many of the key ideas, his name is associated with the design approach, and the von Neumann architecture was soon a standard design used for building electronic computers. The original EDVAC (electronic discrete variable automatic computer) draft report³ contains these interesting passages from von Neumann:

1.4 The remarks of 1.2 on the desired automatic functioning of the device must, of course, assume that it functions faultlessly. Malfunctioning of any device has, however, always a finite probability—and for a complicated device and a long sequence of operations it may not be possible to keep this probability negligible. Any error may vitiate the entire output of the device. For the recognition and correction of such malfunctions intelligent human intervention will in general be necessary.

However, it may be possible to avoid even these phenomena to some extent. The device may recognize the most frequent malfunctions automatically, indicate their presence and location by externally visible signs, and then stop. Under certain conditions it might even carry out the necessary correction automatically and continue (cf. {3.3}).

...

3.3 In the course of this discussion the viewpoints of 1.4, concerned with the detection, location, and under certain conditions even correction, of malfunctions must also receive some consideration. That is, attention must be given to facilities for checking errors. We will not be able to do anything like full justice to this important subject, but we will try to consider it at least cursorily whenever this seems essential.²

The practical problems that concerned von Neumann and the designers of the EDVAC in 1945 were the reliability of vacuum tubes and the main memory stored with mercury delay lines. (A modern hard drive is an amazing electromechanical device as well, which finally is starting to be replaced with solid-state memory.) The invention of the transistor, integrated circuit, and error-correcting codes make von Neumann's concerns seem quaint today. Single-bit errors and even multi-bit errors in computer systems, while still possible, are sufficiently rare that these problems are considered unimportant. The possible failure of system components, however, can no longer be ignored with the advent of cloud computers that fill acres of space with commodity servers.

MURPHY'S LAW TRIUMPHS OVER MOORE'S LAW

A cloud computer is composed of so many components, each with a finite probability of failure, that the probability of all the components running without error at any point in time is close to zero. Failure and automated recovery are hence essential areas of concern not only at the hardware layer but also with the software components. In short, we are at the point where Murphy's law has conquered Moore's law. Assuming that all the components make a system work flawlessly is a luxury that is no longer possible. Fortunately, techniques for handling bit-level corruption can be adjusted and scaled to cloud computers, so most bit-level errors can be detected, if not fixed. The types of failures we are worried about, however, are those of a server or whole groups of servers. We are also at a point where the rates of certain failures are so high that von Neumann's suggestion that the system simply detect the error and wait for a human operator to intervene is no longer economically sensible.

You might ask if we can work harder to build more reliable systems, but when the reliability of your main power supply is inversely proportional to the density of wire-eating squirrels in your region or the probability that a worker will drop an uninsulated wrench into a power-distribution cabinet, it is hard to imagine a cost-effective and systematic approach to address the reliability of data centers that commonly house as many as 100,000 servers.

FROM EDVAC TO WEBVAC

One very popular approach to dealing with frequent server-level failures in a data center is to decompose the system into one or more tiers of servers that process requests on a best-effort basis and store any critical application state in a dedicated storage tier. Typically, there is a request load balancer in front of each tier so that the individual servers in the tier can fail and have requests rerouted automatically. The key aspect of this design is a complete separation between long-term system state and computation. This is the same separation that existed between the processor and memory in the EDVAC design. For lack of a better term let's call these systems WEBVACs, for Worldwide Elastic Big Very Automated Clusters, shown in figure 1.

These WEBVACs are not conceptually different from the farms of Web servers seen today in traditional data centers. WEBVACs use a proven architecture that provides resiliency, scalability, and a very familiar programming model based on stateless HTTP requests. The chief innovation is the degree and ease of configurability, as well as elasticity and scale. One important feature of EDVAC that distinguished it from earlier computers such as ENIAC was that EDVAC executed a program that was stored in its memory as data, while ENIAC was programmed by physically rewiring it for different problems.

Like EDVAC, modern cloud computers allow for the automatic configuration of a complete server farm with a few simple artifacts. This eliminates the need for tedious and error-prone manual configuration of servers, as is often done in more traditional systems.

Building a reliable storage tier that meets the needs of the compute tier is a challenging task. Requests in the storage tier need to be replicated across several servers using complex distributed consensus protocols. There is a wide range of approaches to building storage tiers, as well as a great diversity in their APIs and consistency models. (It is hard to do justice to this topic in the limited space here, so please see the suggested reading at the end of this article.) In the end, however, the storage tier is just an abstraction that is callable from the compute tier. The compute tier can rely on the guarantees provided by the storage tier and therefore uses a much simpler programming model.

This simpler programming model, in which all the important state of the system is stored in a generic storage tier, also simplifies disaster-recovery scenarios since simple backup and restore of the

storage tier is often sufficient to restore an entire system into a working state. Well-designed systems have asynchronous continuous backup of the storage tier to a replica in a physically different location. This location needs to be close enough that data can be efficiently and cost-effectively replicated, but distant enough that the probability of it encountering the same “act of God” is low. (Putting both your primary and backup data centers near the same earthquake fault is a bad idea.)

Since the backup is asynchronous, failover to the replica may incur some data loss. That data loss, however, can be bounded to acceptable and well-defined limits that come into play only if an act of God causes the complete destruction of the primary system. Carefully determining the physical location of your data centers is the first case where there is a need to treat failure in an end-to-end way. This same end-to-end focus on failure is also important in the design and implementation of software running on and interacting with the system.

HOW TO DESIGN INTERFACES FOR WEBVACS

WEBVACs ultimately provide APIs that allow desktop computers, mobile devices, or other WEBVACs to submit requests and receive responses to those requests. In any case, you end up with two agents that must communicate with each other via some interface over an unreliable channel. Reusing traditional designs from client-server systems or standard RPC (remote procedure call) methods is not the best approach. Andrew Tanenbaum and Robbert van Renesse¹ describe some common pitfalls when doing naïve refactoring of code that was not designed for distributed scenarios, which are generally applicable to the APIs here as well. One particular problem they call out is the 2AP (two-army problem), that is, it is impossible to design a fully reliable method for two agents to reach consensus over an unreliable channel that may silently drop messages.

This is a restricted version of the more general problem of dealing with Byzantine failure, where the failure does not include data corruption. As a consequence, there is simply no way of building a system that can process any request with 100 percent reliability if the channel itself is unreliable. The 2AP result, however, does not rule out protocols that asymptotically approach 100 percent reliability. A simple solution is continually transmitting a request up to some finite bound until some acknowledgment is received. If the error rate of the channel is fixed and failures are independent, then the likelihood of success increases exponentially with the number of transmissions.

In a large data center, not only is the communication between servers unreliable, but the servers themselves are also prone to failure. If a server in the compute tier fails, then a request that targeted it can be quickly rerouted to an equivalent compute server. The process of rerouting the request is often not fully transparent, however, and the request may be lost during rerouting, because the routing logic cannot immediately detect the server failure or because the server was in the middle of processing a request when it failed. These lost requests appear to users as transient faults.

In the context of cloud computing, therefore, the observed request failure rate is really the combined error rate of the communication channel and the failure rate of the servers involved in the computation. Rather than reasoning about the individual failure rates of several components, you can make the simplifying assumption that a system of two unreliable agents communicating over an unreliable channel is equivalent to two idealized reliable agents communicating over an unreliable channel whose failure rate is increased appropriately to account for the failure of either of the original unreliable agents. An extended example illustrates this in more detail in the following section.

ENUMERATING A SET OF FILES OVER AN UNRELIABLE CHANNEL

The following is a simple interface definition in ANSI C that can be used to enumerate a set of file names. The interface has been exposed without careful consideration for failure, beyond the introduction of a new status code `Fault`, which indicates a failure likely caused by unreliable delivery. Assume that calling any one of these functions sends a request and waits synchronously for a response. The assumption is that the `Fault` status is returned if no response to a request is received after some fixed timeout.

```
enum Result {
    Ok,          /* Completed without errors. */
    NoMore,      /* No more names left to enumerate.*/
    Fault        /* Message lost in transit or unknown failure. */
};

/* Moves cursor before first element in list of files. */
Result SetCursorToStart();

/* Get the current file name pointed to by the cursor.
 * Returns NoMore if the cursor is moved past the last name.
 */
Result GetCurrentFileName(char fileName[MAXLENGTH]);

/* Move the cursor to the next file name.
 * Returns NoMore if there is none.
 */
Result MoveToNextFileName();
```

Here is a simple client-side function that attempts to enumerate all the files but returns immediately on the first `Fault` received by any call to the primitive functions above:

```
Result ListFilesStopAtAnyFault()
{
    char fileName[MAXLENGTH];
    Result res;
    res = SetCursorToStart();
    if (res != Ok) { return res; }
    printf("Start\n");
    for (;;)
    {
        res = MoveToNextFileName();
        if (res == NoMore) { break; }
        if (res != Ok) { return res; }
        res = GetCurrentFileName(fileName);
    }
}
```

```

        printf("File: %s", fileName);
    }
    printf("End\n");
    return Ok;
}

```

You want to estimate the probability this function will return `Ok` under the assumption that calling any of the three functions mentioned earlier has a success rate of 0.99999 (S), which is to say that on average one out of a million invocations of the functions returns `Fault`. First you need to compute how many requests (M) are required to enumerate N files. Inspection of the code reveals that M is equal to

$$1 + 2 * N + 1$$

which can be simplified to

$$2 * (N + 1).$$

Since the function fails immediately on any fault, the probability of no faults is simply the probability that all the requests sent succeed, which is S^M assuming failures are uniformly distributed. For purposes of this analysis, let's assume failures are independent and uniformly distributed. This simplifying assumption allows a comparison of the tradeoffs of various approaches under equivalent ideal failure models. In practice, however, the distribution of failures is typically neither uniform nor completely independent. The results are summarized in figure 2.

Depending on the workload characteristics, this success rate for the first attempt at listing files may be acceptable. In this example, the success rate of 0.99999 (a five-nines success rate results in fewer than 5.3 minutes of downtime a year for a continuously running system) is extremely high and typically can be achieved only with significant investment in expensive hardware infrastructure. A more realistic error rate would be 0.999 (a three-nines success rate results in fewer than 8.8 hours of downtime a year for a continuously running system), which is more typically seen with commodity components. A three-nines success rate produces the graph and table of values in figure 3.

Clearly, a three percent failure rate for enumerating 10 files is not a usable system. You can improve the probability of success by simply retrying the whole function, but not only is this inefficient, but for large N , the success rate is so low that it would require an unreasonable number of retries. If the probability of the function `ListFilesStopAtAnyFault` enumerating N files successfully is

$$LS(N) = S^{2*(N+1)}$$

then the probability of failure is

$$LF(N) = 1 - LS(N).$$

FIGURE 2

Probability of Success with Five Nines

Request Success Rate (S)	Number of files (N)	Requests Sent (M)	Probability of Success
0.99999	N	$M = 2 * (N + 1)$	S^M
0.99999	1	4	0.99996
0.99999	10	22	0.99978
0.99999	100	202	0.99798
0.99999	1,000	2,002	0.98017
0.99999	10,000	20,002	0.81871

The probability that after at most K retries the function succeeds is the same as the probability

$$1 - LF(N)^K$$

which is the complement of the probability that all invocations fail. For this discussion, if the probability of success is at least 0.999, when N is 100, you must retry, on average, five times; when N is 1,000, the number of retries is at least 50 to get a three-nines success rate; for $N = 10,000$, the number is close to 3 billion. The smarter approach is to keep `ListFilesStopAtAnyFault` from immediately failing on any single fault.

This can be accomplished by creating simple wrapper functions that add some basic retry logic over the original primitives so there is a new set of more robust primitives.

FIGURE 3

Probability of Success with Three Nines

Request Success Rate (S)	Number of files (N)	Requests Sent (M)	Probability of Success
0.999	N	$M = 2 * (N + 1)$	S^M
0.999	1	4	0.99600
0.999	10	22	0.97822
0.999	100	202	0.81701
0.999	1,000	2,002	0.13492
0.999	10,000	20,002	$2 * 10^{-9}$

```

#define MAX_RETRIES 3
Result SetCursorToStartWithRetry()
{
    Result res;
    for (int i = 0; i < MAX_RETRIES; i++)
    {
        res = SetCursorToStart();
        if (res != Fault) { return res; }
    }
    return Fault;
}

```



```

Result GetCurrentFileNameWithRetry(char fileName[MAXLENGTH])
{
    Result res;
    for (int i = 0; i < MAX_RETRIES; i++)
    {
        res = GetCurrentFileName(fileName);
        if (res != Fault) { return res; }
    }
    return Fault;
}

Result MoveToNextFileNameWithRetry()
{
    Result res;
    for (int i = 0; i < MAX_RETRIES; i++)
    {
        res = MoveToNextFileName();
        if (res != Fault) { return res; }
    }
    return Fault;
}

```

Production code would likely include an exponential back-off that delays each retry with an exponentially increasing time delay. This avoids the so-called “thundering herd” problem when many clients are simultaneously trying to recover from a network partition to a given server. For simplicity, this discussion will ignore it. Assuming a success rate of 0.999 for the underlying primitives, performing three simple retries makes the probability of each of these returning without a fault

$$1 - (1 - 0.999)^3 = 1 - (1 - 0.999)^3$$

or 0.999999999 (nine nines). You can now write a new routine that uses these more reliable wrappers:

```

Result ListFilesWithRetry()
{
    char fileName[MAXLENGTH];
    Result res;
    res = SetCursorToStartWithRetry();
    if (res != Ok) { return res; }
    printf("Start\n");
    for (;;)
    {

```

```

    res = MoveToNextFileNameWithRetry();
    if (res == NoMore) { break; }
    if (res != Ok) { return res; }
    res = GetCurrentFileNameWithRetry(fileName);
    printf("File: %s", fileName);
}
printf("End\n");
return Ok;
}

```

Now you can evaluate the reliability of the function `ListFilesWithRetry`, but instead of computing this with respect to primitive requests, you compute it with respect to the number of times each request wrapper is called:

Wrapper Success Rate (W)	Number of files (N)	Wrappers Called (C)	Probability of Success
$1 - [1 - 0.999]^3$	N	$C = 2 * [N + 1]$	W^C
0.999999999	1	4	0.999999996
0.999999999	10	22	0.999999978
0.999999999	100	202	0.999999798
0.999999999	1,000	2,002	0.999997998
0.999999999	10,000	20,002	0.999979998

Now that each wrapper has a nine-nines success rate, the overall success rate for this function, even when $N = 10,000$, is more than 0.9999 (a four-nines success rate results in fewer than 53 minutes of downtime a year for a continuously running system). There is still a nonzero chance this function will return `Fault`, so this approach has not solved the 2AP but has significantly increased the likelihood that the system will make progress. The insertion of retries, of course, increases the overall latency when there are errors, and, with a reasonable model of latency, the expected time for enumerating N files assuming a specific request failure rate can be computed. The latency impact of these changes is discussed later.

Astute readers should notice a fatal flaw in the code above. The function will continue to enumerate under the presence of request failures, but the naïve addition of retries will cause files to be skipped when there are failures. Specifically, this wrapper function may cause files to be skipped:

Result `MoveToNextFileNameWithRetry();`

The fundamental issue here is that the underlying primitive request `MoveToNextFileName` is not idempotent—one invocation of it is not observationally equivalent to multiple invocations of the function. Because of the 2AP, there is no way to have the server and client agree about whether the cursor has moved forward on a fault. The only way to resolve this issue is to make `MoveToNextFileName` idempotent.

There are a variety of techniques to do this. One way is to include sequence numbers to detect retries and have the server track these numbers. These sequence numbers now become important state that must be placed in the storage tier for every client in the system, and this can result in

scalability issues. A more scalable approach is to use an opaque state token similar to how cookies are used in HTTP to offload state from the server to the client. The client can maintain the needed state rather than have the server track it. This leaves the following API, which includes only idempotent functions:

```
typedef int tok_t;
/* Get start token with cursor before first element in list of files. */
Result GetStartToken(tok_t *init);

/*
 * Get the current file name pointed to by the cursor relative to a state token.
 * Returns NoMore if the cursor is moved past the last name.
 */
Result GetCurrentFileNameWithToken(tok_t current, char fileName[MAXLENGTH]);

/* Move the given token and return the next state token with the cursor advanced.
 * Returns NoMore if there is none.
 */
Result MoveToNextFileNameWithToken(tok_t current, tok_t *next);
```

In this example the state token is simply an integer in a realistic system; it would more likely be a variable-length byte array that the system verifies is valid so that malicious clients cannot harm the system.

The retry wrappers can also be defined as earlier for these new primitives:

```
Result GetStartTokenWithRetry(tok_t *init)
{
...
}

Result GetCurrentFileNameWithTokenAndRetry(tok_t current, char fileName[MAXLENGTH])
{
...
}

Result MoveToNextFileNameWithTokenAndRetry(tok_t current, tok_t *next)
{
...
}
```

Finally, let's define a function that uses wrapped primitives over the idempotent API:

```

Result ListFilesWithTokenAndRetry()
{
    char fileName[MAXLENGTH];
    Result res;
    tok_t current;
    res = GetStartTokenWithRetry(&current);
    if (res != Ok) { return res; }
    printf("Start\n");
    for (;;)
    {
        res = MoveToNextFileNameWithTokenAndRetry(current, &current);
        if (res == NoMore) { break; }
        if (res != Ok) { return res; }
        res = GetCurrentFileNameWithTokenAndRetry(current, fileName);
        printf("File: %s", fileName);
    }
    printf("End\n");
    return Ok;
}

```

The analysis performed earlier on the slightly buggy version that lacked the idempotent primitives is still valid, but now the function works correctly and reliably. When $N = 10,000$, the success rate is 0.999979998 (four nines).

ESTIMATING ADDITIONAL LATENCY CAUSED BY RETRIES

Because the only practical way to detect message loss between senders and receivers is via timeouts, it is important to document a time bound on how long either party should wait for a request to be processed. If a request is processed after that time bound has been exceeded, then consider the request failed. Services typically define an upper bound for each API function they support (e.g., 99.9 percent of all requests will be successfully processed within one second). If no upper time bound is specified, a guaranteed 99.9 percent success rate is somewhat meaningless; you may have to wait an infinite amount of time for any single request to complete successfully.

Determining a reasonable upper bound for a system can be quite complex. It can be estimated observationally by looking at the distribution of latencies across a large sample of requests and choosing the maximum, or by simply having the server include a watchdog timer with a clear upper bound that fails any request that exceeds the threshold. In either case, the worst-case bound is needed so that clients of the service can set timeouts appropriately, but these worst-case bounds typically are very conservative estimates of the actual average-case performance. This article analyzes both worst-case and average-case latency for a file-listing function. To simplify the analysis, let's assume that every primitive request to the server has a worst-case successful latency of R_{max} and an average time of R_{avg} .

Given these values, let's estimate the worst-case and average-case times for the function `ListFilesWithRetry`. Since we are only interested in successful calls, the naïve worst-case latency is

directly related to the number of calls to the wrapper functions and their worst-case latency WR_{max} .

The worst-case scenario is if all the retry functions encounter two failures that take exactly R_{max} followed by one successful call that is R_{max} . The equation is

$$WR_{max} = 3 * R_{max}$$

since MAX_RETRIES is defined to be 3. The number of wrapper calls to enumerate N files is

$$2 * (N + 1)$$

so the worst-case latency is

$$2 * (N + 1) * WR_{max} = 3 * R_{max}$$

or

$$L_{max}(N) = 6 * (N + 1) * R_{max}$$

This worst-case estimate is extremely pessimistic. There is an alternative approach that gives a more reasonable upper bound.

The average-case analysis is a little more complicated, because all possible outcomes must be considered. As a simplifying assumption, all failed requests take R_{max} to complete. This is a slightly conservative assumption, but since the latency distribution of a failed request is not *a priori* distributed normally, this bit of conservatism is justified. To compute WR_{avg} you must first consider the time for all possible outcomes weighted by their likelihood including failures:

Weight	Time
0.999	R_{avg}
$0.001 * 0.999$	$R_{max} + R_{avg}$
$0.001 * 0.001 * 0.999$	$R_{max} + R_{max} + R_{avg}$
$0.001 * 0.001 * 0.001$	$R_{max} + R_{max} + R_{max}$

Since we are interested only in successes, the last outcome is dropped and the weights are renormalized by the subpopulation of interest (0.999999999):

Weight	Time
$0.999 / 0.9999999$	R_{avg}
$(0.001 * 0.999) / 0.999999999$	$R_{max} + R_{avg}$
$(0.001 * 0.001 * 0.999) / 0.999999999$	$R_{max} + R_{max} + R_{avg}$

This leaves WR_{avg} equal to

$$\frac{0.999 * R_{avg} + (0.001 * 0.999) * (R_{avg} + R_{max}) + (0.001 * 0.001 * 0.999) * (R_{max} + R_{max} + R_{avg})}{0.999999999}$$

and

$$L_{avg}(N) = 2 * (N + 1) * WR_{avg} = 2 * (N + 1) * (R_{avg} + 0.001000998001001 * R_{max})$$

Here is a table of values when $R_{max} = 1,000$ milliseconds and $R_{avg} = 10$ milliseconds, with those values of WR_{avg} approximately 11.000998 milliseconds:

N	$L_{max}(N)$	$L_{avg}(N)$
1	12 seconds	44 milliseconds
10	1.1 minutes	242 milliseconds
100	10.1 minutes	2.22 seconds
1,000	1.66 hours	22 seconds
10,000	16.6 hours	3.7 minutes

Note that the worst-case latency is an extremely conservative estimate and reflects the extremely improbable sequence of events where failures always occur on the first two requests and the third retry succeeds with the worst-case latency. An alternative upper bound is based on the average-case analysis where you simply assume $R_{avg} = R_{max}$ and compute $WR_{slow} = 1001.000998001$. This provides a more likely distribution of events that model very slow requests rather than pathological behavior. Let's call this bound

$$L_{slow}(N) = 2 * (N + 1) * WR_{slow}.$$

The table compares the three values with different values of N :

N	$L_{max}(N)$	$L_{slow}(N)$	$L_{avg}(N)$
1	12 seconds	4 seconds	44 milliseconds
10	1.1 minutes	22 seconds	242 milliseconds
100	10.1 minutes	3.34 minutes	2.22 seconds
1000	1.66 hours	33.4 minutes	22 seconds
10000	16.6 hours	5.56 hours	3.7 minutes

IMPROVING LATENCY

Some readers may feel this final implementation is too “chatty” and a more efficient protocol could reduce server round-trips. Indeed, three functions in the API can be replaced with one:

```

/* Well known start token. */
tok_t StartToken = -1;

/* Given a token return the next state token with the cursor advanced
 * and current file name.
 * Returns NoMore if there are no files.
 */
Result GetCurrentFileNameAndNextToken(
    tok_t curent,
    char fileName[MAXLENGTH],
    tok_t *next);

```

This protocol has a fixed globally known start token and a single function that returns both the current file name and the next token in one request. We can define a simple retry wrapper for it as well:

```

Result GetCurrentFileNameAndNextTokenWithRetry(
    tok_t curent,
    char fileName[MAXLENGTH],
    tok_t *next)
{
    ...
}

```

Finally, the function is adjusted to use the new interface and we can analyze the reliability:

```

Result ListFilesWithTokenAndRetryOpt()
{
    char fileName[MAXLENGTH];
    Result res;
    tok_t current = StartToken;
    printf("Start\n");
    for (;;)
    {
        res = GetCurrentFileNameAndNextTokenWithRetry(current, fileName, &current);
        if (res == NoMore) { break; }
        if (res != Ok) { return res; }
        printf("File: %s", fileName);
    }
    printf("End\n");
    return Ok;
}

```

Wrapper Success Rate (W)	Number of files (N)	Wrappers Called (C)	Probability of Success
$1 - [1 - 0.999]^3$	N	$C = N + 1$	W^C
0.999999999	1	2	0.999999998
0.999999999	10	11	0.999999989
0.999999999	100	101	0.999999899
0.999999999	1,000	1,001	0.999998999
0.999999999	10,000	10,001	0.999989999

Compared with the previous version, the improvement in success rate is hardly noticeable, but reliability is not the only consideration here. There should be an improvement in expected latency, which can be seen clearly by redoing the latency analysis with this modified approach:

N	$L'_{\max}(N)$	$L'_{\text{slow}}(N)$	$L'_{\text{avg}}(N)$
1	6 seconds	2 seconds	22 milliseconds
10	33 seconds	11 seconds	121 milliseconds
100	5.05 minutes	1.68 minutes	1.11 seconds
1,000	50.5 minutes	16.7 minutes	11.0 seconds
10,000	8.33 hours	2.78 hours	1.83 minutes

FRACTIONAL RETRIES POLICY: DOES IT HELP?

After looking at the reliability tables for the last two versions of these file-listing functions, readers may observe that the probability of success is well above three nines, even when $N = 10,000$, so a natural question arises: Are we retrying too much? Can some reliability be traded for better average latency? One observation is that in the best-case scenario, when there are no retries, the timing for $N = 10,000$ is 1.67 minutes, which means the current approach of three retries adds, at most, 10 percent in increased latency. Let's assume no interest in a success rate of more than 0.999 when $N \leq 10,000$. Working backward, you can compute a minimal retry policy so that when $N = 10,000$, you are just above three nines:

$$W^{10001} > 0.999$$

Solving for WW :

$$W > e^{\left(\frac{\ln(0.999)}{10001}\right)}$$

W is greater than approximately 0.999999899959976, which works out to seven nines. Remember

$$W = 1 - (1 - 0.999)^K$$

where K is the number of retries. Solving for K , you will find that K needs to be approximately 2.4. The fractional retry value may seem absurd, but it is easy to implement by deterministically retrying the first two times and nondeterministically executing the final retry based on a fair coin toss.


```

Result GetCurrentFileNameAndNextTokenWithRetry(tok
_t curent, char fileName[MAXLENGTH], tok_t *next)
{
    Result res;
    res = GetCurrentFileNameAndNextToken(curent, fileName, next);
    if (res != Fault) { return res; }
    res = GetCurrentFileNameAndNextToken(curent, fileName, next);
    if (res != Fault) { return res; }
    if (CoinFlip()) { return Fault; }
    return GetCurrentFileNameAndNextToken(curent, fileName, next);
}

```

This function provides 2.5 expected retries, yielding an expected success rate of 0.9996 for $N = 10,000$. Now let's compute the average latency based on this fractional policy. Start by enumerating the outcomes and weighted likelihoods:

Weight	Time
0.999	R_{avg}
$0.001 * 0.999$	$R_{max} + R_{avg}$
$0.001 * 0.001 * 0.5 * 0.999$	$R_{max} + R_{max} + R_{avg}$
$0.001 * 0.001 * 0.5 * 0.001$	$R_{max} + R_{max} + R_{max}$
$0.001 * 0.001 * 0.5$	$R_{max} + R_{max}$

Again normalize the weights so only the successes are considered:

Weight	Time
$0.999 / 0.9999994995$	R_{avg}
$(0.001 * 0.999) / 0.9999994995$	$R_{max} + R_{avg}$
$(0.001 * 0.001 * 0.5 * 0.999) / 0.9999994995$	$R_{max} + R_{max} + R_{avg}$

$$\frac{0.999 * R_{avg} + (0.001 * 0.999) * (R_{max} + R_{avg}) + (0.001 * 0.001 * 0.5 * 0.999) * (R_{max} + R_{max} + R_{avg})}{0.999999499}$$

When $R_{avg} = 10$ and $R_{max} = 1000$, the $WR'_{avg} = 10.999999$. When the expected time is computed for $N = 10,000$, the expected improvement is almost unnoticeable, which may not be too surprising since the likelihood of the first two retries falling is already very low. The example, however, uses a simple linear retry policy, while in practice you are more likely to use an exponential back-off algorithm. Readers may want to redo this analysis with a more realistic exponential back-off retry

policy. All of the basic techniques presented in this article can be used to analytically estimate the average- and worst-case latencies with such a policy as well.

CONCLUSIONS

This article is a far-from-exhaustive survey of the many interesting issues surrounding cloud computing. The goal is to demonstrate that there is a wide variety of deep problems still to be solved. Some of the trailblazers who developed the electronic computer would be dumbfounded by the computation that we now carry in our pockets. They would be equally surprised at how robustly some of their earliest ideas have stood the test of time. Taken in historical context, the modern WEBVAC should not be seen as the culmination of 70 years of human progress, but just the start of a promising future that we can't imagine.

ACKNOWLEDGMENTS

Special thanks go to Gang Tan for providing early feedback and encouraging me to write this article, as well as Steve Zdancewic, who provided early review feedback.

REFERENCES

1. Tanenbaum, A. S., van Renesse, R. 1988. A critique of the remote procedure call paradigm. In EUTECO (European Teleinformatics Conference) Proceedings, Participants Edition: 775-783.
2. von Neumann, J. 1945. *First Draft of a Report on the EDVAC*. Technical Report. <https://web.archive.org/web/20130314123032/http://qss.stanford.edu/~godfrey/vonNeumann/vnedvac.pdf>.
3. Wikipedia. First draft of a report on the EDVAC; http://en.wikipedia.org/wiki/First_Draft_of_a_Report_on_the_EDVAC.

FURTHER READING

Calder, B., Wang, J., Ogus, A., et al. 2011. Windows Azure Storage: a highly available cloud storage service with strong consistency. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*: 143-157. DOI=10.1145/2043556.2043571; <http://doi.acm.org/10.1145/2043556.2043571>

DeCandia, G., Hastorun, D., et al. 2007. Dynamo: Amazon's highly available key-value store. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles*: 205-220. DOI=10.1145/1294261.1294281; <http://doi.acm.org/10.1145/1294261.1294281>.

Ghemawat, S., Gobioff, H., Leung, S.-T. 2003. The Google file system. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*: 29-43. DOI=10.1145/945445.945450; <http://doi.acm.org/10.1145/945445.945450>.

Related content on queue.acm.org:

Describing the Elephant: The Different Faces of IT as Service
 Ian Foster and Steven Tuecke
<http://queue.acm.org/detail.cfm?id=1080874>

Lessons from the Floor
 Daniel Rogers
<http://queue.acm.org/detail.cfm?id=1113334>

Distributed Computing Economics

Jim Gray

<http://queue.acm.org/detail.cfm?id=1394131>

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

DANIEL C. WANG has a Ph.D from Princeton University in computer science and has been working in the computing industry for more than 15 years. He now works on the Azure Web Workload team. Any opinions in the article are those of the author and not of his employer.

© 2015 ACM 1542-7730/14/0300 \$10.00