Transient fault handling

07/13/2016 • 19 minutes to read • Contributors all

In this article

Why do transient faults occur in the cloud?
Challenges
General guidelines
More information

All applications that communicate with remote services and resources must be sensitive to transient faults. This is especially the case for applications that run in the cloud, where the nature of the environment and connectivity over the Internet means these types of faults are likely to be encountered more often. Transient faults include the momentary loss of network connectivity to components and services, the temporary unavailability of a service, or timeouts that arise when a service is busy. These faults are often self-correcting, and if the action is repeated after a suitable delay it is likely to succeed.

This document covers general guidance for transient fault handling. For information about handling transient faults when using Microsoft Azure services, see Azure service-specific retry guidelines.

Why do transient faults occur in the cloud?

Transient faults can occur in any environment, on any platform or operating system, and in any kind of application. In solutions that run on local on-premises infrastructure, the performance and availability of the application and its components is typically maintained through expensive and often underused hardware redundancy, and components and resources are located close to each other. While this approach makes a failure less likely, it can still result in transient faults - and even an outage through unforeseen events such as external power supply or network issues, or other disaster scenarios.

Cloud hosting, including private cloud systems, can offer higher overall availability by using shared resources, redundancy, automatic failover, and dynamic resource allocation across many commodity compute nodes. However, the nature of these environments can mean that transient faults are more likely to occur. There are several reasons for this:

- Many resources in a cloud environment are shared, and access to these resources is subject to throttling in order to protect the resource. Some services will refuse connections when the load rises to a specific level, or a maximum throughput rate is reached, in order to allow processing of existing requests and to maintain performance of the service for all users. Throttling helps to maintain the quality of service for neighbors and other tenants using the shared resource.
- Cloud environments are built using vast numbers of commodity hardware units. They deliver performance by dynamically distributing the load across multiple computing units and infrastructure components, and deliver reliability by automatically recycling or replacing failed units. This dynamic nature means that transient faults and temporary connection failures may occasionally occur.
- There are often more hardware components, including network infrastructure such as routers and load balancers, between the application and the resources and services it uses. This additional infrastructure can occasionally introduce additional connection latency and transient connection faults.
- Network conditions between the client and the server may be variable, especially when communication crosses the Internet. Even in on-premises locations, heavy traffic loads may slow communication and cause intermittent connection failures.

Challenges

Transient faults can have a huge effect on the perceived availability of an application, even if it has been thoroughly tested under all foreseeable circumstances. To ensure that cloud-hosted applications operate reliably, they must be able to respond to the following challenges:

- The application must be able to detect faults when they occur, and determine if these faults are likely to be transient, more long-lasting, or are terminal failures. Different resources are likely to return different responses when a fault occurs, and these responses may also vary depending on the context of the operation; for example, the response for an error when reading from storage may be different from response for an error when writing to storage. Many resources and services have well-documented transient failure contracts. However, where such information is not available, it may be difficult to discover the nature of the fault and whether it is likely to be transient.

- The application must be able to retry the operation if it determines that the fault is likely to be transient and keep track of the number of times the operation was retried.

- The application must use an appropriate strategy for the retries. This strategy specifies the number of times it should retry, the delay between each attempt, and the actions to take after a failed attempt. The appropriate number of attempts and the delay between each one are often difficult to determine, and vary based on the type of resource as well as the current operating conditions of the resource and the application itself.

General guidelines

The following guidelines will help you to design a suitable transient fault handing mechanism for your applications:

- **Determine if there is a built-in retry mechanism:**

  - Many services provide an SDK or client library that contains a transient fault handling mechanism. The retry policy it uses is typically tailored to the nature and requirements of the target service. Alternatively, REST interfaces for services may return information that is useful in determining whether a retry is appropriate, and how long to wait before the next retry attempt.

  - Use the built-in retry mechanism where available, unless you have specific and well-understood requirements that make a different retry behavior more appropriate.

- **Determine if the operation is suitable for retrying**:

  - You should only retry operations where the faults are transient (typically indicated by the nature of the error), and if there is at least some likelihood that the operation will succeed when reattempted. There is no point in reattempting operations that indicate an invalid operation such as a database update to an item that does not exist, or requests to a service or resource that has suffered a fatal error.

  - In general, you should implement retries only where the full impact of this can be determined, and the conditions are well understood and can be validated. If not, leave it to the calling code to implement retries. Remember that the errors returned from resources and services outside your control may evolve over time, and you may need to revisit your transient fault detection logic.

  - When you create services or components, consider implementing error codes and messages that will help clients determine whether they should retry failed operations. In particular, indicate if the client should retry the operation (perhaps by returning an **isTransient** value) and suggest a suitable delay before the next retry attempt. If you build a web service, consider returning custom errors defined within your service contracts. Even though generic clients may not be able to read these, they will be useful when building custom clients.

- **Determine an appropriate retry count and interval:**

- It is vital to optimize the retry count and the interval to the type of use case. If you do not retry a sufficient number of times, the application will be unable to complete the operation and is likely to experience a failure. If you retry too many times, or with too short an interval between tries, the application can potentially hold resources such as threads, connections, and memory for long periods, which will adversely affect the health of the application.

- The appropriate values for the time interval and the number of retry attempts depend on the type of operation being attempted. For example, if the operation is part of a user interaction, the interval should be short and only a few retries attempted to avoid making users wait for a response (which holds open connections and can reduce availability for other users). If the operation is part of a long running or critical workflow, where canceling and restarting the process is expensive or time-consuming, it is appropriate to wait longer between attempts and retry more times.

- Determining the appropriate intervals between retries is the most difficult part of designing a successful strategy. Typical strategies use the following types of retry interval:

  - **Exponential back-off**. The application waits a short time before the first retry, and then exponentially increasing times between each subsequent retry. For example, it may retry the operation after 3 seconds, 12 seconds, 30 seconds, and so on.

  - **Incremental intervals**. The application waits a short time before the first retry, and then incrementally increasing times between each subsequent retry. For example, it may retry the operation after 3 seconds, 7 seconds, 13 seconds, and so on.

  - **Regular intervals**. The application waits for the same period of time between each attempt. For example, it may retry the operation every 3 seconds.

  - **Immediate retry**. Sometimes a transient fault is brief, perhaps due to an event such as a network packet collision or a spike in a hardware component. In this case, retrying the operation immediately is appropriate because it may succeed if the fault has cleared in the time it takes the application to assemble and send the next request. However, there should never be more than one immediate retry attempt, and you should switch to alternative strategies, such as exponential back-off or fallback actions, if the immediate retry fails.

  - **Randomization**. Any of the retry strategies listed above may include a randomization to prevent multiple instances of the client sending subsequent retry attempts at the same time. For example, one instance may retry the operation after 3 seconds, 11 seconds, 28 seconds, and so on, while another instance may retry the operation after 4 seconds, 12 seconds, 26 seconds, and so on. Randomization is a useful technique that may be combined with other strategies.

- As a general guideline, use an exponential back-off strategy for background operations, and immediate or regular interval retry strategies for interactive operations. In both cases, you should choose the delay and the retry count so that the maximum latency for all retry attempts is within the required end-to-end latency requirement.

- Take into account the combination of all the factors that contribute to the overall maximum timeout for a retried operation. These factors include the time taken for a failed connection to produce a response (typically set by a timeout value in the client) as well as the delay between retry attempts and the maximum number of retries. The total of all these times can result in long overall operation times, especially when using an exponential delay strategy where the interval between retries grows rapidly after each failure. If a process must meet a specific service level agreement (SLA), the overall operation time, including all timeouts and delays, must be within the limits defined in the SLA.

- Overly aggressive retry strategies, which have intervals that are too short or retries that are too frequent, can have an adverse effect on the target resource or service. This may prevent the resource or service from recovering from its overloaded state, and it will continue to block or refuse requests. This results in a vicious

circle where more and more requests are sent to the resource or service, and consequently its ability to recover is further reduced.

  - Take into account the timeout of the operations when choosing the retry intervals to avoid launching a subsequent attempt immediately (for example, if the timeout period is similar to the retry interval). Also consider if you need to keep the total possible period (the timeout plus the retry intervals) to below a specific total time. Operations that have unusually short or very long timeouts may influence how long to wait, and how often to retry the operation.

  - Use the type of the exception and any data it contains, or the error codes and messages returned from the service, to optimize the interval and the number of retries. For example, some exceptions or error codes (such as the HTTP code 503 Service Unavailable with a Retry-After header in the response) may indicate how long the error might last, or that the service has failed and will not respond to any subsequent attempt.

- **Avoid anti-patterns**:

  - In the vast majority of cases, you should avoid implementations that include duplicated layers of retry code. Avoid designs that include cascading retry mechanisms, or that implement retry at every stage of an operation that involves a hierarchy of requests, unless you have specific requirements that demand this. In these exceptional circumstances, use policies that prevent excessive numbers of retries and delay periods, and make sure you understand the consequences. For example, if one component makes a request to another, which then accesses the target service, and you implement retry with a count of three on both calls there will be nine retry attempts in total against the service. Many services and resources implement a built-in retry mechanism and you should investigate how you can disable or modify this if you need to implement retries at a higher level.

  - Never implement an endless retry mechanism. This is likely to prevent the resource or service recovering from overload situations, and cause throttling and refused connections to continue for a longer period. Use a finite number or retries, or implement a pattern such as Circuit Breaker to allow the service to recover.

  - Never perform an immediate retry more than once.

  - Avoid using a regular retry interval, especially when you have a large number of retry attempts, when accessing services and resources in Azure. The optimum approach is this scenario is an exponential back-off strategy with a circuit-breaking capability.

  - Prevent multiple instances of the same client, or multiple instances of different clients, from sending retries at the same times. If this is likely to occur, introduce randomization into the retry intervals.

- **Test your retry strategy and implementation:**

  - Ensure you fully test your retry strategy implementation under as wide a set of circumstances as possible, especially when both the application and the target resources or services it uses are under extreme load. To check behavior during testing, you can:

    - Inject transient and non-transient faults into the service. For example, send invalid requests or add code that detects test requests and responds with different types of errors. For an example using TestApi, see Fault Injection Testing with TestApi and Introduction to TestApi – Part 5: Managed Code Fault Injection APIs.

    - Create a mock of the resource or service that returns a range of errors that the real service may return. Ensure you cover all the types of error that your retry strategy is designed to detect.

    - Force transient errors to occur by temporarily disabling or overloading the service if it is a custom service that you created and deployed (of course, you should not attempt to overload any shared resources or shared services within Azure).

    - For HTTP-based APIs, consider using the FiddlerCore library in your automated tests to change the outcome of HTTP requests, either by adding extra roundtrip times or by changing the response (such as the HTTP

status code, headers, body, or other factors). This enables deterministic testing of a subset of the failure conditions, whether transient faults or other types of failure. For more information, see FiddlerCore. For examples of how to use the library, particularly the **HttpMangler** class, examine the source code for the Azure Storage SDK.

    - Perform high load factor and concurrent tests to ensure that the retry mechanism and strategy works correctly under these conditions, and does not have an adverse effect on the operation of the client or cause cross-contamination between requests.

- **Manage retry policy configurations:**

    - A *retry policy* is a combination of all of the elements of your retry strategy. It defines the detection mechanism that determines whether a fault is likely to be transient, the type of interval to use (such as regular, exponential back-off, and randomization), the actual interval value(s), and the number of times to retry.

    - Retries must be implemented in many places within even the simplest application, and in every layer of more complex applications. Rather than hard-coding the elements of each policy at multiple locations, consider using a central point for storing all the policies. For example, store the values such as the interval and retry count in application configuration files, read them at runtime, and programmatically build the retry policies. This makes it easier to manage the settings, and to modify and fine-tune the values in order to respond to changing requirements and scenarios. However, design the system to store the values rather than rereading a configuration file every time, and ensure suitable defaults are used if the values cannot be obtained from configuration.

    - In an Azure Cloud Services application, consider storing the values that are used to build the retry policies at runtime in the service configuration file so that they can be changed without needing to restart the application.

    - Take advantage of built-in or default retry strategies available in the client APIs you use, but only where they are appropriate for your scenario. These strategies are typically general purpose. In some scenarios they may be all that is required, but in other scenarios they may not offer the full range of options to suit your specific requirements. You must understand how the settings will affect your application through testing to determine the most appropriate values.

- **Log and track transient and non-transient faults:**

    - As part of your retry strategy, include exception handling and other instrumentation that logs when retry attempts are made. While an occasional transient failure and retry are to be expected, and do not indicate a problem, regular and increasing numbers of retries are often an indicator of an issue that may cause a failure, or is currently degrading application performance and availability.

    - Log transient faults as Warning entries rather than Error entries so that monitoring systems do not detect them as application errors that may trigger false alerts.

    - Consider storing a value in your log entries that indicates if the retries were caused by throttling in the service, or by other types of faults such as connection failures, so that you can differentiate them during analysis of the data. An increase in the number of throttling errors is often an indicator of a design flaw in the application or the need to switch to a premium service that offers dedicated hardware.

    - Consider measuring and logging the overall time taken for operations that include a retry mechanism. This is a good indicator of the overall effect of transient faults on user response times, process latency, and the efficiency of the application use cases. Also log the number of retries occurred in order to understand the factors that contributed to the response time.

    - Consider implementing a telemetry and monitoring system that can raise alerts when the number and rate of failures, the average number of retries, or the overall times taken for operations to succeed, is increasing.

- **Manage operations that continually fail:**

- There will be circumstances where the operation continues to fail at every attempt, and it is vital to consider how you will handle this situation:
  - Although a retry strategy will define the maximum number of times that an operation should be retried, it does not prevent the application repeating the operation again, with the same number of retries. For example, if an order processing service fails with a fatal error that puts it out of action permanently, the retry strategy may detect a connection timeout and consider it to be a transient fault. The code will retry the operation a specified number of times and then give up. However, when another customer places an order, the operation will be attempted again - even though it is sure to fail every time.
  - To prevent continual retries for operations that continually fail, consider implementing the Circuit Breaker pattern. In this pattern, if the number of failures within a specified time window exceeds the threshold, requests are returned to the caller immediately as errors, without attempting to access the failed resource or service.
  - The application can periodically test the service, on an intermittent basis and with long intervals between requests, to detect when it becomes available. An appropriate interval will depend on the scenario, such as the criticality of the operation and the nature of the service, and might be anything between a few minutes and several hours. At the point where the test succeeds, the application can resume normal operations and pass requests to the newly recovered service.
  - In the meantime, it may be possible to fall back to another instance of the service (perhaps in a different datacenter or application), use a similar service that offers compatible (perhaps simpler) functionality, or perform some alternative operations in the hope that the service will become available soon. For example, it may be appropriate to store requests for the service in a queue or data store and replay them later. Otherwise you might be able to redirect the user to an alternative instance of the application, degrade the performance of the application but still offer acceptable functionality, or just return a message to the user indicating that the application is not available at present.

- **Other considerations**
  - When deciding on the values for the number of retries and the retry intervals for a policy, consider if the operation on the service or resource is part of a long-running or multistep operation. It may be difficult or expensive to compensate all the other operational steps that have already succeeded when one fails. In this case, a very long interval and a large number of retries may be acceptable as long as it does not block other operations by holding or locking scarce resources.
  - Consider if retrying the same operation may cause inconsistencies in data. If some parts of a multistep process are repeated, and the operations are not idempotent, it may result in an inconsistency. For example, an operation that increments a value, if repeated, will produce an invalid result. Repeating an operation that sends a message to a queue may cause an inconsistency in the message consumer if it cannot detect duplicate messages. To prevent this, ensure that you design each step as an idempotent operation. For more information about idempotency, see Idempotency patterns.
  - Consider the scope of the operations that will be retried. For example, it may be easier to implement retry code at a level that encompasses several operations, and retry them all if one fails. However, doing this may result in idempotency issues or unnecessary rollback operations.
  - If you choose a retry scope that encompasses several operations, take into account the total latency of all of them when determining the retry intervals, when monitoring the time taken, and before raising alerts for failures.
  - Consider how your retry strategy may affect neighbors and other tenants in a shared application, or when using shared resources and services. Aggressive retry policies can cause an increasing number of transient faults to occur for these other users and for applications that share the resources and services. Likewise, your application may be affected by the retry policies implemented by other users of the resources and services. For

mission-critical applications, you may decide to use premium services that are not shared. This provides you with much more control over the load and consequent throttling of these resources and services, which can help to justify the additional cost.

More information

- Azure service-specific retry guidelines
- Circuit Breaker pattern
- Compensating Transaction pattern
- Idempotency patterns