ジッターを伴うタイムアウト、再試行、およびバックオフ

Marc Brooker

失敗が発生

あるサービスまたはシステムが別のサービスまたはシステムを呼び出すたびに、障害が発生する可能性があります。これらの失敗は、さまざまな要因から発生することがあります。サーバー、ネットワーク、ロードバランサー、ソフトウェア、オペレーティングシステム、またはシステムオペレーターからのミスが含まれます。故障の可能性を減らすようにシステムを設計していますが、故障しないシステムを構築することは不可能です。そのため、Amazon では、障害の可能性を許容して削減するようにシステムを設計し、わずかな割合の障害を完全に停止するまで拡大することを防いでいます。回復力のあるシステムを構築するために、タイムアウト、再試行、およびバックオフの 3 つの重要なツールを採用しています。

リクエストが通常より長くかかり、完了しない可能性があるため、障害の種類の多くが明らかになります。クライアントがリクエストの完了を通常よりも長く待っている場合、そのリクエストに長時間使用していたリソースも保持します。多くのリクエストがリソースを長時間保持すると、サーバーがそれらのリソースを使い果たす可能性があります。これらのリソースには、メモリ、スレッド、接続、エフェメラルポートなど、制限されているものが含まれます。この状況を回避するために、クライアントはタイムアウトを設定します。タイムアウトは、クライアントがリクエストの完了まで待機する最大時間です。

多くの場合は、同じリクエストを再試行すると、リクエストが成功します。私たちが構築するシステムの種類が単一のユニットとして失敗することがほとんどないためです。むしろ、部分的な障害または一時的な障害が発生します。部分的な障害とは、リクエストの一部が成功した場合を言います。一時的な障害とは、リクエストが短期間失敗した場合を言います。再試行を使用すると、クライアントは同じリクエストを再送信することで、これらのランダムな部分的障害と短期間の一時的障害に耐えることができます。

再試行は必ずしも安全ではありません。再試行は、システムが過負荷に近づいたせいで既に障害が発生している場合、呼び出されるシステムの負荷を増加させる可能性があります。この問題を回避するために、バックオフを使用するようにクライアントを実装します。これにより、後続の再試行間の時間が長くなり、バックエンドの負荷が均等に維持されます。再試行に関する他の問題は、一部のリモート呼び出しに副作用があることです。タイムアウトまたは失敗は、

必ずしも副作用が発生しなかったことを意味するわけではありません。副作用を複数回行うことが望ましくない場合、ベストプラクティスは、べき等になるように API を設計することです。つまり、安全に再試行できることを意味します。

最後に、トラフィックは一定の速度で Amazon サービスに到着するわけではありません。代わりに、リクエストの到着率には大きなバーストが頻繁に発生します。これらのバーストは、クライアントの動作、障害回復、さらには定期的な cron ジョブのような単純なものによっても発生する可能性があります。負荷が原因でエラーが発生した場合、すべてのクライアントが同時に再試行すると、再試行が無効になる可能性があります。この問題を回避するために、ジッターを使用します。これは、リクエストを作成または再試行する前のランダムな時間で、到着率を拡大することで大きなバーストを防ぐことができます。

これらの各ソリューションについては、以降のセクションで説明します。

タイムアウト

Amazon のベストプラクティスは、リモートコールにタイムアウトを設定することです。通常、同じボックス上であっても、プロセス間のコールにはタイムアウトを設定します。これには、接続タイムアウトとリクエストタイムアウトの両方が含まれます。多くの標準クライアントは、堅牢な組み込みタイムアウト機能を提供します。

通常、最も難しい問題は、設定するタイムアウト値を選択することです。タイムアウトの設定が高すぎると、クライアントがタイムアウトを待つ間もリソースが消費されるため、有用性が低下します。タイムアウトの設定が低すぎると、2 つのリスクがあります。

- 再試行されるリクエストが多すぎるため、バックエンドのトラフィックが増加し、レイテンシーが増加する。
- すべてのリクエストの再試行が開始されるため、小さなバックエンドレイテンシーが増加し、完全な停止に至る。

AWS リージョン内の呼び出しのタイムアウトを選択するには、ダウンストリームサービスのレイテンシーメトリックから開始するのが良い方法です。そのため、Amazon では、あるサービスから別のサービスを呼び出すときに、許容される間違ったタイムアウトの割合 (0.1% など) を

選択します。次に、ダウンストリームサービスの対応するレイテンシーパーセンタイルを調べます (この例では p99.9)。このアプローチはほとんどの場合うまく機能しますが、次のようにいくつかの落とし穴があります。

- このアプローチは、インターネット経由など、クライアントにかなりのネットワークレイテンシーがある場合には機能しない。このような場合は、クライアントが世界中に広がる可能性があることを念頭に置いて、最悪の事態における合理的なネットワークレイテンシーを考慮します。

- このアプローチは、p99.9 が p50 に近い、レイテンシーの境界が厳しいサービスでも機能しない。このような場合、間隔を追加することで、多数のタイムアウトが発生する小さなレイテンシーが増えることを回避できます。

- タイムアウトを実装するときによくある落とし穴に遭遇する。Linux の SO_RCVTIMEO は強力ですが、エンドツーエンドのソケットタイムアウトとしては不適切となるいくつかの欠点があります。Java などの一部の言語は、このコントロールを直接公開します。Go などの他の言語は、より堅牢なタイムアウトメカニズムを提供します。

- DNS や TLS ハンドシェイクなど、タイムアウトがすべてのリモートコールを処理しない実装もある。一般的に、十分にテストされたクライアントに組み込まれたタイムアウトを使用することを好みます。独自のタイムアウトを実装する場合は、タイムアウトソケットオプションの正確な意味と実行中の作業に注意を払います。

Amazon で作業していた 1 つのシステムでは、デプロイの直後に依存関係と通信する少数のタイムアウトが見られました。タイムアウトは非常に低く、約 20 ミリ秒に設定されました。デプロイ以外では、タイムアウト値がこのように低くても、タイムアウトが定期的に発生することはありませんでした。掘り下げてみると、タイマーには新しい安全な接続の確立が含まれ、以降のリクエストで再利用されることがわかりました。接続の確立に 20 ミリ秒以上かかったため、デプロイ後に新しいサーバーがサービスを開始すると、少数のリクエストがタイムアウトすることがわかりました。場合によっては、リクエストは再試行されてから成功しました。私たちは、最初に、接続が確立された場合にタイムアウト値を増やすことでこの問題を回避しました。後で、プロセスの起動時、ただしトラフィックを受信する前に、これらの接続を確立

することにより、システムを改善しました。これにより、タイムアウト問題を完全に回避することができました。

再試行とバックオフ

再試行は「利己的」です。 つまり、クライアントが再試行する場合、サーバーの時間をより多く費やして成功可能性を高めます。障害が稀に、または一時的に発生した場合は、問題ではありません。これは、再試行されたリクエストの総数が少なく、見かけの可用性を高めるというトレードオフがうまく機能するのが理由です。過負荷が原因で障害が発生した場合、負荷を増加させる再試行により事態が著しく悪化する可能性があります。元の問題が解決された後も、長時間、負荷を高く保つことで、回復を遅らせることさえあります。再試行は強力な薬に似ています。適切な用量を服用すると有用ですが、使いすぎると重大な損傷を引き起こす可能性があります。残念ながら、分散システムでは、すべてのクライアント間で調整して適切な回数の再試行を行う方法はほとんどありません。

Amazon で使用する推奨ソリューションは、バックオフです。すぐに積極的に再試行する代わりに、クライアントは試行の間にある程度の時間を待ちます。最も一般的なパターンはエクスポネンシャルバックオフで、試行ごとに待機時間が急激に増加します。エクスポネンシャルバックオフは急速に成長するため、非常に長いバックオフ時間につながる可能性があります。再試行が長すぎるのを避けるために、実装は通常、バックオフを最大値に制限します。これは、予想通り、上限付きエクスポネンシャルバックオフと呼ばれます。ただし、これにより別の問題が発生します。現在、すべてのクライアントが上限レートで継続的に再試行を行っています。ほとんどすべての場合、私たちのソリューションは、クライアントが再試行する回数を制限し、サービス指向アーキテクチャーの初期段階で発生する障害を処理することです。ほとんどの場合、クライアントは独自のタイムアウトを持っているため、とにかく呼び出しを放棄します。

再試行には他にも次のような問題があります。

- 分散システムには多くの場合、複数のレイヤーがあります。顧客の呼び出しがサービス呼び出しの 5 つの深度のスタックを引き起こすシステムについて検討してください。データベースへのクエリで終了し、各レイヤーで 3 回再試行します。データベースが負荷によってクエリに失敗し始めるとどうなりますか? 各レイヤーが個別に再試行すると、

データベースの負荷が 243 倍に増加し、回復する可能性が低くなります。最初の 3 回の試行、次の 9 回の試行など、各レイヤーでの再試行が複数回行われるためです。対照的に、スタックの最上位レイヤーで再試行すると、以前の呼び出し作業が無駄になり、効率が低下します。一般的に、低コストのコントロールプレーンおよびデータプレーン操作の場合、スタックの単一ポイントで再試行することがベストプラクティスです。

- 負荷がかかる。再試行の単一レイヤーを使用しても、エラーが開始されるとトラフィックは大幅に増加します。エラーのしきい値を超えるとダウンストリームサービスへの呼び出しが完全に停止するサーキットブレーカーは、この問題を解決するために広く薦められています。サーキットブレーカーはシステムにモーダル動作を導入しますが、残念ながら、これはテストが困難な場合があり、復旧にかなりの時間を費やす可能性があります。トークンバケット使用してローカルで再試行を制限することにより、このリスクを軽減できることがわかりました。これにより、トークンがある限りすべての呼び出しが再試行され、トークンが使い果たされると固定レートで再試行されます。AWS は 2016 年にこの動作を AWS SDK に追加しました。そのため、SDK を使用するお客様には調整動作が組み込まれています。

- 再試行するタイミングを決める。一般的に、副作用のある API は、べき等性を提供しない限り、再試行しても安全ではないと考えています。これにより、再試行の頻度に関係なく、副作用が 1 回だけ発生することが保証されます。通常、読み取り専用 API には冪等性がありますが、リソース作成 API はそうではありません。Amazon Elastic Compute Cloud (Amazon EC2) RunInstances API などの一部の API は、明示的なトークンベースのメカニズムを提供して、べき等性を提供し、安全に再試行できるようにします。副作用の重複を防ぐには、適切な API 設計を行い、クライアントが実装時に注意を向ける必要があります。

- 再試行する価値のある障害を把握する。HTTP は、クライアントエラーとサーバーエラーを明確に区別します。クライアントエラーは後で成功することはないため、同じリクエストで再試行することはできませんが、サーバーエラーは後続の試行で成功する可能性があることを示しています。残念ながら、システムの結果整合性はこの線を大幅に曖

昧にします。状態が伝播されるにつれて、クライアントエラーが次の瞬間に成功に変わる場合があります。

これらのリスクと課題にもかかわらず、再試行は、一時的なエラーやランダムなエラーに直面したときに高可用性を提供するための強力なメカニズムです。各サービスの適切なトレードオフを見つけるには、判断が必要です。私たちの経験では、再試行が利己的であることを思い出せる場所が開始場所として良好です。再試行は、クライアントがリクエストの重要性を主張し、サービスがそれを処理するためにより多くのリソースを費やすことをリクエストする方法です。クライアントが利己的すぎると、広範囲にわたる問題を引き起こす可能性があります。

ジッター

過負荷または競合が障害の原因である場合、バックオフが非常に役に立たないことがよくあります。これは相関関係が原因です。失敗したすべてのコールが同じ時間にバックオフされた場合、再試行時に再び競合または過負荷が発生します。私たちのソリューションはジッターです。ジッターはバックオフにある程度のランダム性を追加して、再試行を時間内に分散させます。追加するジッターの量と最適な追加方法については、エクスポネンシャルバックオフとジッターをご覧ください。

ジッターは再試行専用ではありません。運用上の経験から、コントロールプレーンとデータプレーンの両方を含むサービスへのトラフィックが急増する傾向があることがわかりました。これらのトラフィックの急上昇は非常に短い場合があり、多くの場合、集約されたメトリックによって非表示になります。システムを構築する場合、すべてのタイマー、定期的なジョブ、およびその他の遅延作業にジッターを追加することを検討します。これにより、作業の急増が分散され、ダウンストリームサービスがワークロードに合わせて拡張しやすくなります。

スケジュールされた作業にジッターを追加する場合、各ホストのジッターをランダムに選択することはありません。代わりに、同じホスト上で毎回同じ数を生成する一貫したメソッドを使用します。このように、過負荷されているサービスまたは競合状態がある場合、パターン内で同じように発生します。私たち人間はパターンを識別するのが得意なので、根本原因を特定する可能性が高くなります。ランダムなメソッドを使用すると、リソースが圧倒されている場合にのみ、つまりランダムに発生します。これにより、トラブルシューティングがはるかに困難になります。

Amazon Elastic Block Store (Amazon EBS) や AWS Lambda など、私が取り組んでいるシステムでは、クライアントが 1 分に 1 回などの定期的な間隔でリクエストを頻繁に送信することがわかりました。ただし、クライアントが同じように動作する複数のサーバーを持っている場合、それらは同時に並んでリクエストをトリガーできます。1 分のうち最初の数秒、毎日のジョブの場合は真夜中以降の最初の数秒です。1 秒あたりの負荷に注意を払い、クライアントと定期的なワークロードをジッターさせることで、より少ないサーバー容量で同じ量の作業を達成できました。

顧客のトラフィックの急増に対する制御はあまりありません。ただし、顧客がトリガーするタスクであっても、顧客エクスペリエンスに影響を与えない場所にジッターを追加することをお勧めします。

まとめ

分散システムでは、リモート対話での一時的な障害やレイテンシーは避けられません。タイムアウトはシステムが不当に長くぶら下げられるのを防ぎ、再試行はそれらの障害を隠し、バックオフとジッターは使用率を改善し、システムの混雑を軽減します。

Amazon では、再試行に注意を向けることが重要であることを学びました。再試行により、依存システムの負荷が増大する可能性があります。システムへの呼び出しがタイムアウトになり、そのシステムが過負荷になると、再試行により過負荷が改善される代わりに悪化する可能性があります。依存関係が正常であることが確認された場合にのみ再試行することにより、この増加を回避します。再試行が可用性の改善に役立たない場合は、再試行を停止します。