ヘルスチェックの実装

David Yanacek

サービスはあらゆる種類の信頼性と復元力を組み込んで設計できますが、実務で信頼できるようにするには、発生した予測可能な障害にも対処する必要があります。Amazon では、ハードウェアは最終的には機能しなくなるように設計されているため、水平方向にスケーラブルで冗長なサービスを構築しています。どのハードドライブにも最大予想寿命があり、ソフトウェアのどの部分もある時点でクラッシュする可能性があります。サーバーの正常性はバイナリのように見える場合があります。動作するか、まったく動作せず、正常に機能しないかのどちらかです。ですが、そうではありません。障害が発生したサーバーは、シャットダウンするだけでなく、予測できないか、場合によってはシステムへ不均衡な損害をもたらす可能性があります。ヘルスチェックは、これらの種類の問題を自動的に検出して対応します。

この記事では、ヘルスチェックを使用して単一サーバーの障害を検出して対処する方法、ヘルスチェックを使用しない場合に発生すること、ヘルスチェックの失敗に過剰に反応するシステムにより小さな問題が完全な停止につながる経緯について説明します。また、Amazon での経験から、さまざまな種類のヘルスチェック実装間のトレードオフのバランスについての洞察を提供します。

大きな影響を伴う小さな障害

私が Amazon の新人のソフトウェア開発者だったとき、Amazon.com の背後にあるウェブサイトレンダリングフリートに取り組んでいました。いくつかのインストルメンテーションを追加し、ソフトウェアの実行状況を可視化するための変更に取り組んでいる間、残念ながらバグを書いてしまいました。バグはめったにトリガーしませんでしたが、トリガーすると、特定のウェブサーバーがすべてのリクエストで空白のエラーページをレンダリングしました。問題を修正するには、ウェブサーバープロセスを再起動するしかありませんでした。バグを検出し、変更を迅速にロールバックし、多くのテストを追加し、プロセスを改善して将来このような状況をキャッチするようにしました。けれども、バグが本番であった間に、大規模なフリートの少数のサーバーがこの壊れた状態になりました。

バグを見つけるのを特に困難にしたのは、サーバーがそれが異常であることを認識しなかったためです。また、サーバーの状態をモニタリングシステムに報告する機能が失われたため、サーバーは自動的に停止せず、定期的なアラームもトリガーされませんでした。さらに悪いことに、サーバーは非常に高速になり、ピアの「正常なサーバー」が適切なウェブページをレンダリングするよりもはるかに速く空白のエラーページを生成し始めました。当時使用していたロードバランシングテクノロジーは、低速サーバーよりも高速サーバーを優先していたため、不均衡な量のトラフィックを不健全なサーバーに誘導し、さらに影響を大きくしました。

モニタリングにはシステム内の複数のポイントからのエラー率とレイテンシーの測定が含まれるため、他のアラームがトリガーされます。これらの種類のモニタリングシステムと運用プロセスは、問題を封じ込めるためのバックストップとして機能しますが、適切なヘルスチェックは、障害を迅速に検出して対処することにより、このクラスのエラーの影響を大幅に最小限に抑えることができます。

ヘルスチェックのトレードオフ

ヘルスチェックは、特定のサーバー上のサービスに、作業を正常に実行できるかどうかを確認する方法です。ロードバランサーは、各サーバーにこの質問を定期的に行い、トラフィックを転送しても安全なサーバーを判断します。キューからメッセージをポーリングするサービスは、キューからさらに作業をポーリングすることを決定する前に、自身が正常であるかどうかを自問する場合があります。モニタリングエージェント (各サーバーまたは外部モニタリングフリートで実行) は、サーバーが正常であるかどうかを確認して、アラームを発したり、障害が発生したサーバーに自動的に対処したりすることができます。

私のウェブサイトのバグの例で見たように、異常なサーバーがサービスを継続すると、サービス全体の可用性が不均衡に低下する可能性があります。10 台のサーバーのフリートでは、1 つの不良サーバーは、フリートの可用性が 90% 以下になることを意味します。さらに悪いことに、「最小リクエスト」などの一部の負荷分散アルゴリズムが、最速のサーバーにより多くの作業を与えます。サーバーに障害が発生すると、多くの場合、リクエストがすぐに失敗し始め、正常なサーバーよりも多くのリクエストを引き付けることで、サービスフリートに「ブラックホール」が作成されます。場合によっては、追加の保護を加えて、失敗したリクエストの速度を落としてブラックホールを防ぎ、成功したリクエストの平均レイテンシーに一致させます。ただし、キューポーラーなど、この問題を回避するのがより難しい他のシナリオがあります。たとえば、キューポーラーがメッセージを受信できる速度でポーリングしている場合、障害が発生したサーバーもブラックホールになります。作業を分散するためのこのような多様な環境のセットでは、部分的に障害が発生したサーバーの保護について考える方法は、システムによって異なります。

書き込み不能になり、リクエストがすぐに失敗するディスク、突然スキューし、依存関係の呼び出しが認証に失敗するクロック、更新された暗号素材を取得できず、復号化と暗号化の失敗を引き起こすサーバー、それにバグ、メモリリーク、および処理をフリーズするデッドロックのためにクラッシュする重要なサポートプロセスなど、さまざまな理由でサーバーに独立して障害が発生することがわかっています。

フリート内の多くのサーバーまたはすべてのサーバーが同時に障害を引き起こす相関関係のある理由でも、サーバーに障害が発生します。相関する理由には、共有依存関係の停止と大規模なネットワークの問題が含まれます。

理想的なヘルスチェックは、サーバーおよびアプリケーションのヘルスのあらゆる側面をテストし、おそらく重要ではないサポートプロセスが実行されていることをも検証するものでしょう。ただし、重大ではない理由でヘルスチェックが失敗し、その失敗がサーバー間で相関している場合、トラブルが発生します。サーバーがまだ有用な作業を実行できたときに、自動化によってサーバーがサービスから削除されると、自動化は害をもたらします。

ヘルスチェックの難しさは、一方では徹底的なヘルスチェックの利点と、単一サーバー障害の迅速な軽減の利点、他方ではフリート全体での誤検出による障害との間のこの緊張状態にあります。したがって、良好なヘルスチェックを構築するための課題の 1 つは、誤検出を慎重に防ぐことです。一般的に、これは、ヘルスチェックを取り巻く自動化がトラフィックを単一の不良サーバーに転送するのを停止し、フリート全体に問題があるように見える場合はトラフィックを許可し続けることを意味します。

正常性を測定する方法

サーバー上で破損する可能性のあるものは数多くあり、システム内にはサーバーの状態を測定する多くの場所があります。特定のサーバーが独立して故障していることを明確に報告できるヘルスチェックもあれば、相関障害の場合にファジーで誤検出を報告するヘルスチェックもあります。一部のヘルスチェックは実装が困難です。その他は、Amazon Elastic Compute Cloud (Amazon EC2) や Elastic Load Balancing などのサービスを使用してセットアップ時に実装されます。ヘルスチェックの各タイプには長所があります。

ライブ状態チェック

ライブ状態チェックは、サービスへの基本的な接続とサーバープロセスの存在をテストします。多くの場合、ロードバランサーまたは外部モニタリングエージェントによって実行され、アプリケーションの動作に関する詳細を認識しません。ライブ状態チェックはサービスに含まれる傾向があり、アプリケーションの作成者が何かを実装する必要はありません。Amazon で使用するライブ状態チェックの例には次のものがあります。

- サーバーが予想されるポートでリッスンし、新しい TCP 接続を受け入れることを確認するテスト。
- 基本的な HTTP リクエストを実行し、サーバーが 200 ステータスコードで応答することを確認するテスト。
- ネットワーク到達可能性など、システムの動作に必要な基本事項をテストする Amazon EC2 のステータスチェック。

ローカルヘルスチェック

ローカルヘルスチェックは、ライブ状態チェックよりもさらに進んで、アプリケーションが機能する可能性が高いことを確認します。これらのヘルスチェックは、サーバーのピアと共有されていないリソースをテストします。したがって、フリート内の多くのサーバーで同時に障害が発生することはほとんどありません。これらのヘルスチェックは、以下についてテストします。

- ディスクへの書き込みまたはディスクからの読み取りができない—ステートレスサービスが書き込み可能なディスクを必要としないと思いがちです。ただし、Amazon のサービスは、非同期計測データの監視、ロギング、公開などの目的でディスクを使用する傾向があります。
- 重大なプロセスのクラッシュまたは破壊—一部のサービスは、オンサーバープロキシ (NGINX と同様) を使用してリクエストを受け取り、別のサーバープロセスでビジネスロジックを実行します。ライブ状態チェックでは、プロキシプロセスが実行されているかどうかをテストするのみでしょう。ローカルヘルスチェックプロセスでは、プロキシからアプリケーションにパススルーして、両方が実行されており、リクエストに正しく応答していることを確認する可能性があります。興味深いことに、記事の冒頭のウェブサイトの例では、既存のヘルスチェックはレンダリングプロセスが実行されて応答するかを確認するのには十分な深さでしたが、正しく応答しているかを確認するのに十分な深さではありませんでした。
- 不足しているサポートプロセス—モニタリングデーモンが不足しているホストは、オペレーターを「盲目に飛行」させ、サービスの健全性を認識させない可能性があります。その他のサポートプロセスでは、計測および請求の使用記録をプッシュするか、資格情報の更新を受け取ります。サポートプロセスが壊れているサーバーは、機能を微妙で検出が困難な方法で危険にさらします。

依存関係のヘルスチェック

依存関係のヘルスチェックは、アプリケーションが隣接システムと対話する能力を徹底的に検査します。このチェックは、理想的には、期限切れの資格情報など、依存関係との対話を妨げているサーバーに依存する問題をキャッチします。ただし、依存関係自体に問題がある場合は、誤検出される可能性もあります。これらの誤検出のため、依存関係のヘルスチェックの失敗にどう反応するかに注意する必要があります。依存関係のヘルスチェックでは、次のことをテストできます。

- 不正な設定または古いメタデータ—プロセスがメタデータまたは設定の更新を非同期で検索しているが、サーバーで更新メカニズムが破損している場合、サーバーはピアとの同期が大幅に遅れ、予測不能でテストされていない動作をする可能性があります。ただし、サーバーがしばらく更新を認識しない場合、更新メカニズムが壊れているのか、中央更新システムがすべてのサーバーへの更新の公開を停止したかどうかはわかりません。
- ピアサーバーまたは依存関係と通信できない—奇妙なネットワーク動作は、フリート内のサーバーのサブセットが依存関係と通信する能力に影響を与え、そのサーバーに送信されるトラフィックの能力には影響を与えないことがわかっています。デッドロックや接続プールのバグなどのソフトウェアの問題も、ネットワーク通信を妨げる可能性があります。
- プロセスのバウンスを必要とするその他の異常なソフトウェアバグ—デッドロック、メモリリーク、または状態破損のバグにより、サーバーでエラーが発生する可能性があります。

異常検出

異常検出は、フリート内のすべてのサーバーを調べて、ピアと比較して異常に動作しているサーバーがあるかどうかを判断します。サーバーごとにモニタリングデータを集約することにより、エラー率、遅延データ、またはその他の属性を継続的に比較して異常なサーバーを見つけ、それらをサービスから自動的に削除できます。異常検出では、次のように、サーバーがそれ自体について検出できないフリートの相違を検出できます。

- クロックスキュー—特にサーバーの負荷が高い場合、クロックが急激かつ大幅にスキューすることがわかっています。AWS への署名付きリクエストを評価するために使用するようなセキュリティ対策では、クライアントのクロックの時間が実際の時間の 5 分以内であることが必要です。そうでない場合、AWS のサービスへのリクエストは失敗します。

- 古いコード—サーバーが長時間ネットワークから切断されるか電源が切断され、その後オンラインに戻ると、他のフリートと互換性のない危険なほど古いコードが実行されている可能性があります。
- 予期しない障害モード—サーバーが、エラーはサーバーのエラーではなくクライアントのエラー (500 ではなく HTTP 400) として識別するエラーを返すような障害が発生する場合があります。サーバーは、失敗する代わりにスローダウンするか、ピアよりも速く応答する場合があります。これは、発信者に誤った応答を返していることを示しています。異常検出は、予期しない障害モードに対する驚くべきキャッチオールです。

異常検出が実際に機能するためには、次のいくつかのことが当てはまります。

- サーバーはほぼ同じことを行う必要がある—さまざまな種類のトラフィックをさまざまな種類のサーバーに明示的にルーティングする場合、サーバーは異常値を検出するほど十分に動作しない場合があります。ただし、ロードバランサーを使用してトラフィックをサーバーに送信する場合、同様の方法で応答する可能性があります。
- フリートは比較的同種である必要があります—さまざまなインスタンスタイプを含むフリートでは、一部のインスタンスが他のインスタンスよりも遅くなる可能性があり、受動的な不良サーバー検出を誤ってトリガーする可能性があります。このシナリオを回避するために、インスタンスタイプごとにメトリクスを照合します。
- エラーや動作の違いを報告する必要がある—エラーを報告するのはサーバー自体に依存しているため、モニタリングシステムも故障した場合はどうなるのでしょうか? 幸いなことに、サービスのクライアントは、インスツルメンテーションを追加するのに最適な場所です。Application Load Balancer などのロードバランサーは、アクセスログを発行します。このログには、リクエストごとにアクセスしたバックエンドサーバー、応答時間、リクエストが成功したか失敗したかが示されます。

ヘルスチェックの失敗に対する安全な対応

サーバーが正常でないと判断した場合、実行できるアクションには 2 種類あります。最も極端な場合、ロードバランサーのヘルスチェックに失敗するか、キューのポーリングを停止することにより、作業を行わないことをローカルで決定し、サービスを停止する可能性があります。サーバーが反応する

別の方法は、問題があることを中央機関に通知し、中央システムに問題の処理方法を決定させることです。中央システムは、自動化にフリート全体をダウンさせることなく、問題に安全に対処できます。

ヘルスチェックを実装して対処する方法は複数あります。このセクションでは、Amazon で使用しているいくつかのパターンについて説明します。

フェールオープン

一部のロードバランサーは、スマートな中央機関として機能できます。個々のサーバーがヘルスチェックに失敗すると、ロードバランサーはトラフィックの送信を停止します。しかし、すべてのサーバーが同時にヘルスチェックに失敗すると、ロードバランサーはオープンに失敗し、すべてのサーバーへのトラフィックが許可されます。ロードバランサーを使用して、依存関係のヘルスチェックの安全な実装をサポートできます。たとえば、データベースにクエリを実行し、重要でないサポートプロセスが実行されていることを確認するチェックを含めます。

たとえば、正常であると報告しているサーバーがない場合、AWS Network Load Balancer はオープンに失敗します。また、アベイラビリティーゾーンのすべてのサーバーが異常を報告した場合、異常なアベイラビリティーゾーンから失敗します。(ヘルスチェックに Network Load Balancer を使用する方法の詳細については、Elastic Load Balancing のドキュメントを参照してください)。 Amazon Route 53 と同様に、Application Load Balancer はフェールオープンもサポートしています。(Route 53 を使用したヘルスチェックの設定の詳細については、Route 53 のドキュメントを参照してください)。

フェールオープン動作に依存する場合、依存関係のヘルスチェックの失敗モードを必ずテストします。たとえば、サーバーが共有データストアに接続するサービスを考えてみてください。そのデータストアが遅くなるか、低いエラーレートで応答する場合、サーバーは依存関係のヘルスチェックに失敗することがあります。この条件により、サーバーはサービスを開始および終了しますが、フェールオープンのしきい値はトリガーされません。このヘルスチェックを使用して依存関係の部分的な障害を推論してテストすることは、障害によって深いヘルスチェックが事態を悪化させる可能性がある状況を回避する上で重要です。

フェールオープンは有用な動作ですが、Amazon では、すべての状況について完全に推論したりテストしたりできないものには懐疑的です。システムまたはそのシステムの依存関係のすべてのタイプの過負荷、部分的な障害、または灰色の障害に対して予想されるように、フェールオープンがトリガーする一般的な証明はまだ考えられていません。この制限のため、Amazon のチームは、高速で動作するロードバランサーのヘルスチェックをローカルのヘルスチェックに制限し、集中化されたシステ

ムに依存して、より深い依存関係のヘルスチェックに慎重に対応する傾向があります。これは、フェールオープン動作を使用しないとか、特定の場合に動作することを証明しないということではありません。しかし、ロジックが多数のサーバーで迅速に機能できる場合、そのロジックについて非常に慎重になります。

サーキットブレーカーなしのヘルスチェック

サーバーが自身の問題に対応できるようにすることは、復旧への最も迅速で簡単なパスのように思えるかもしれません。ただし、サーバーの状態が間違っているか、フリート全体で何が起こっているか全体像を把握していない場合は、最もリスクの高いパスです。フリート全体のすべてのサーバーが同じ誤った決定を同時に行うと、隣接するサービス全体で連鎖障害が発生する可能性があります。このリスクにはトレードオフが伴います。ヘルスチェックとモニタリングにギャップがある場合、サーバーは問題が検出されるまでサービスの可用性を低下させる可能性があります。ただし、このシナリオでは、フリート全体での予期しないヘルスチェック動作による完全なサービス停止を回避します。

以下は、サーキットブレーカーが組み込まれていない場合にヘルスチェックを実装するためのベストプラクティスです。

- ワークプロデューサー (ロードバランサー、キューポーリングスレッド) を設定して、ライブ状態とローカルヘルスチェックを実行する。サーバーは、不良ディスクなど、そのサーバーに決定的にローカルな問題がある場合にのみ、ロードバランサーによって自動的にサービスから除外されます。
- 他の外部モニタリングシステムを設定して、依存関係のヘルスチェックと異常検出を実行する。システムは、インスタンスを自動的に終了したり、オペレーターに警告したり従事させたりすることができます。

依存関係のヘルスチェックの失敗に自動的に対応するシステムを構築する場合、自動化されたシステムが思いがけない行動を起こさないように適切な量のしきい値を組み込む必要があります。Amazon DynamoDB、Amazon S3、Amazon Relational Database Service (Amazon RDS) などのステートフルサーバーを運用する Amazon のチームには、サーバーの交換に関する重要な耐久性要件があります。また、慎重なレート制限と制御フィードバックループを構築し、しきい値を超えると自動化が停止し、人間が関与できるようにしました。このような自動化を構築するときは、サーバーが依

存関係のヘルスチェックに失敗したときに気づくようにする必要があります。一部のメトリクスについては、個々のステータスを中央モニタリングシステムに自己報告するサーバーに依存しています。サーバーが壊れ過ぎているためにサーバーの状態を報告できない場合に備えて、サーバーの状態を確認するために積極的に連絡を取ることもしています。

正常性を優先する

特に過負荷状態では、サーバーが通常の作業よりもヘルスチェックを優先することが重要です。この状況では、ヘルスチェックに失敗するか、ゆっくりと応答すると、悪い電圧低下状況がさらに悪化する可能性があります。

サーバーがロードバランサーのヘルスチェックに失敗すると、ロードバランサーはすぐに、無視できないほどの時間、サービスを停止するように要求します。単一のサーバーに障害が発生した場合、それは問題ではありませんが、サービスへのトラフィックが急増している状況では、サービスのサイズを縮小することは避けたいところです。過負荷時にサーバーのサービスを停止すると、下向きのスパイラルが発生する可能性があります。残りのサーバーにさらに多くのトラフィックを強制すると、サーバーが過負荷になりやすくなり、ヘルスチェックに失敗し、フリートがさらに縮小してしまいます。

問題は、過負荷状態のサーバーが過負荷状態のときにエラーを返すことではありません。むしろ、サーバーがロードバランサーの ping リクエストに時間内に応答しないことにあります。結局のところ、ロードバランサーのヘルスチェックは、他のリモートサービス呼び出しと同様に、タイムアウトで設定します。電圧が低下したサーバーは、CPU の競合が高い、ガベージコレクターサイクルが長い、ワーカースレッドが不足するなど、さまざまな理由で応答が遅くなります。過剰な追加リクエストを処理する代わりに、ヘルスチェックにタイムリーに応答するようにリソースを確保するようにサービスを設定する必要があります。

幸いなことに、この種の下向きのスパイラルを防ぐために従う簡単な設定のベストプラクティスがいくつかあります。iptables などのツール、および一部のロードバランサーでさえ、「最大接続」の概念をサポートしています。 この場合、OS (またはロードバランサー) は、サーバーへの接続数を制限するため、サーバープロセスが、処理速度を低下させる同時リクエストであふれることはありません。

サービスがプロキシまたは最大接続をサポートするロードバランサーによって処理される場合、HTTP サーバー上のワーカースレッドの数をプロキシの最大接続と一致させるのが論理的なようです。ただし、この設定では、電圧低下時にサービスを下向きのスパイラルに設定します。プロキシヘルスチェックにも接続が必要なので、追加のヘルスチェックリクエストに対応できるようにサーバーのワ

ーカープールを十分に大きくすることが重要です。アイドルワーカーは安価であるため、余分なワーカーを設定する傾向があります。少数の余分なワーカーから、設定されたプロキシの最大接続数を 2 倍に増やすことです。

ヘルスチェックの優先順位付けに使用するもう 1 つの戦略は、サーバーが独自の最大同時リクエスト強制を実装することです。この場合、ロードバランサーのヘルスチェックは常に許可されますが、サーバーが既に何らかのしきい値で動作している場合、通常のリクエストは拒否されます。Amazon 周辺の実装は、Java の単純なセマフォから、CPU 使用率の傾向のより複雑な分析まで多岐にわたります。

サービスがヘルスチェック ping リクエストに時間内に応答するようにする別の方法は、バックグラウンドスレッドで依存関係ヘルスチェックロジックを実行し、ping ロジックがチェックする isHealthy フラグを更新することです。この場合、サーバーはヘルスチェックに迅速に応答し、依存関係のヘルスチェックにより、対話する外部システムに予測可能な負荷が生成されます。チームがこれを行う場合、ヘルスチェックスレッドの失敗を検出することに特に注意を払っています。そのバックグラウンドスレッドが終了した場合、サーバーは将来のサーバー障害 (または回復) を検出しません。

依存関係のヘルスチェックと影響範囲のバランス

依存関係のヘルスチェックは、サーバーの健全性の徹底的なテストとして機能するため、魅力的です。残念ながら、依存関係はシステム全体に連鎖的な障害を引き起こす可能性があるため、危険である可能性があります。

Amazon のサービス指向アーキテクチャを見ると、ヘルスチェックの依存関係の処理に関する洞察が得られます。Amazon の各サービスは、少数のことを行うように設計されています。すべてを行うモノリスはありません。この方法でサービスを構築するのには、多くの理由があります。たとえば、小規模なチームによる迅速なイノベーションや、1 つのサービスに問題がある場合の影響範囲の縮小などです。このアーキテクチャ設計は、ヘルスチェックにも適用できます。

あるサービスが別のサービスを呼び出すとき、そのサービスに依存しています。サービスがたまに依存関係を呼び出すだけの場合、依存関係と通信できない場合でもサービスはいくつかの種類の作業を行うことができるため、依存関係を「ソフトな依存関係」と見なす場合があります。フェールオープン保護がなければ、依存関係をテストするヘルスチェックを実装すると、その依存関係が「ハードな

依存関係」に変わります。依存関係がダウンすると、サービスもダウンし、影響範囲が拡大する連鎖障害が発生します。

機能を異なるサービスに分離しても、各サービスはおそらく複数の API を提供します。サービスの API には独自の依存関係がある場合があります。1 つの API が影響を受ける場合でも、サービスが他の API の提供を継続することを好みます。たとえば、サービスは、コントロールプレーン (長期にわたるリソースで CRUD API と呼ばれることもある) とデータプレーン (高スループットビジネス超クリティカル API) の両方にすることができます。コントロールプレーン API の依存関係との通信に問題がある場合でも、データプレーン API の動作を継続する必要があります。

同様に、単一の API でも、データの入力または状態に応じて異なる動作をする場合があります。一般的なパターンは、データベースにクエリを実行するが、しばらくの間ローカルで応答をキャッシュする読み取り API です。データベースがダウンした場合でも、データベースがオンラインに戻るまで、サービスはキャッシュされた読み取りを提供できます。1 つのコードパスのみが正常でない場合にヘルスチェックに失敗すると、依存関係と通信する問題の影響範囲が広がります。

ヘルスチェックへの依存関係に関するこの議論は、マイクロサービスと比較的モノリシックなサービスとのトレードオフについて興味深い質問を提起します。サービスを分割するデプロイ可能なユニットまたはエンドポイントの数について明確なルールが存在することはめったにありませんが、「ヘルスチェックへの依存関係」と「障害を起こして影響範囲を広げる」という質問は、マイクロまたはマクロでサービスを作成する方法を決定する興味深いレンズです。

ヘルスチェックで問題が発生した場合に起こること

これはすべて理論上理にかなっているかもしれませんが、ヘルスチェックが正しく行われないと実際のシステムはどうなるのでしょうか? 全体像を説明するために、AWS のお客様や Amazon 周辺のストーリーのパターンを探しました。また、補正要因、つまり、ヘルスチェックの弱点が広範囲に及ぶ問題を引き起こすのを防ぐためにチームが実装する「ベルトとサスペンダー」の種類も検討しました。

デプロイ

ヘルスチェックの問題の 1 つのパターンには、デプロイが含まれます。AWS CodeDeploy などのデプロイシステムは、新しいコードを一度にフリートのサブセットにプッシュし、1 つのデプロイウェ

ーブが完了するのを待ってから次の移行に進みます。このプロセスは、サーバーが新しいコードで実行されると、デプロイシステムに報告するサーバーに依存しています。報告がない場合、デプロイシステムは新しいコードに問題があると判断し、デプロイをロールバックします。

最も基本的なサービススタートアップデプロイスクリプトは、サーバープロセスを単純に分岐し、すぐにデプロイシステムに「デプロイ完了」と応答します。ただし、これは非常に多くのことが新しいコードでうまくいかない可能性があるため、危険です。新しいコードが起動直後にクラッシュしたり、ハングアップしてサーバーソケットでリッスンを開始できなかったり、リクエストを正常に処理するために必要な設定をロードできなかったり、バグに遭遇したりする可能性があります。依存関係のヘルスチェックをテストするようにデプロイシステムが設定されていない場合、不適切なデプロイをプッシュしていることに気付きません。サーバーを次々に破壊しながら進行します。

幸いなことに、実際には、Amazon チームは複数の緩和システムを実装して、このシナリオがフリート全体をダウンすることを防いでいます。そのような緩和策の 1 つは、フリート全体のサイズが小さすぎる場合や高負荷で実行されている場合、またはレイテンシーやエラー率が高い場合にトリガーするアラームを設定することです。これらのアラームのいずれかがトリガーされると、デプロイシステムはデプロイを停止し、ロールバックします。

別のタイプの緩和策は、段階的なデプロイを使用することです。フリート全体を単一のデプロイメントでデプロイする代わりに、そのゾーンに対して統合テストの完全なスイートを一時停止して実行する前に、サブセット、おそらくアベイラビリティーゾーンをデプロイするようにサービスを設定できます。サービスはすでに単一のアベイラビリティーゾーンに問題がある場合に動作を継続できるように設計されているため、このアベイラビリティーゾーンごとのデプロイが便利です。

そしてもちろん、本番環境にデプロイする前に、Amazon チームはこれらの変更をテスト環境にプッシュし、このタイプの障害を検出する自動統合テストを実行します。ただし、本番環境とテスト環境の間には微妙で避けられない違いが存在する可能性があるため、本番環境に影響を与える前にあらゆる種類の問題をキャッチするために、デプロイの安全性の多くのレイヤーを組み合わせることが重要です。ヘルスチェックは、不適切なデプロイからサービスを保護するために重要ですが、そこで終わらないようにします。フリートをこれらのミスや他のミスから守るためのバックストップとして機能する「ベルトとサスペンダー」アプローチについて考えます。

非同期プロセッサー

別の障害のパターンは、SQS キューまたは Amazon Kinesis Stream をポーリングすることで動作するサービスなど、非同期メッセージ処理に関するものです。ロードバランサーからリクエストを受け取るシステムとは異なり、サーバーをサービスから削除するためにヘルスチェックを自動的に実行するものはありません。

サービスに十分なヘルスチェックがない場合、個々のキューワーカーサーバーは、ディスクがいっぱいになったり、ファイル記述子が不足するなどの障害が発生する可能性があります。この問題は、サーバーがキューから作業を引き出すことを停止しませんが、サーバーがメッセージを正常に処理することを停止します。この問題により、メッセージ処理が遅延し、不良サーバーがキューから作業を急速に引き離し、処理に失敗します。

このような状況では、多くの場合、影響を抑えるのに役立ついくつかの補正要因があります。たとえば、サーバーが SQS を取得するメッセージの処理に失敗した場合、SQS は、設定されたメッセージ可視性タイムアウト後にそのメッセージを別のサーバーに再配信します。エンドツーエンドのレイテンシーは増加しますが、メッセージはドロップされません。もう 1 つの補正要因は、メッセージの処理中にエラーが多すぎるとアラームが鳴り、オペレーターに調査を促すことです。

ディスクがいっぱいになる

別のクラスの障害は、サーバー上のディスクがいっぱいになり、処理とログの両方が失敗する場合です。この障害は、サーバーがその障害をモニタリングシステムに報告できない可能性があるため、モニタリングの可視性にギャップをもたらします。

繰り返しになりますが、いくつかの緩和制御により、サービスが「盲目に飛行」することを防ぎ、影響を迅速に緩和します。Application Load Balancer や API Gateway などのプロキシが前面にあるシステムには、そのプロキシによって生成されたエラーレートとレイテンシーメトリクスがあります。この場合、サーバーがそれらを報告していない場合でも、アラームが発生します。キューベースのシステムの場合、Amazon Simple Queue Service (Amazon SQS) などのサービスは、一部のメッセージの処理が遅れていることを示すメトリクスを報告します。

これらのソリューションに共通することは、モニタリングの複数のレイヤーがあることです。サーバー自体はエラーを報告しますが、外部システムもエラーを報告します。同じ原則がヘルスチェックでも重要です。外部システムは、特定のシステムの状態を、それ自体をテストするよりも正確にテスト

できます。これが、AWS Auto Scaling を使用して、チームが外部 ping ヘルスチェックを行うようにロードバランサーを設定する理由です。

また、チームは独自のカスタムヘルスチェックシステムを作成して、各サーバーが正常かどうかを定期的に確認し、サーバーが正常でない場合は AWS Auto Scaling に報告します。このシステムの一般的な実装の 1 つには、毎分実行される Lambda 関数が含まれ、すべてのサーバーの正常性をテストします。これらのヘルスチェックは、DynamoDB のようなもので各実行間の状態を保存することもできるため、一度に異常なサーバーを不注意にマークすることはありません。

ゾンビ

問題の別のパターンには、ゾンビサーバーがあります。サーバーは、しばらくネットワークから切断され、実行を再開する場合があります。または、長時間電源がオフになり、後で再起動する場合があります。

ゾンビサーバーが復旧すると、他のフリートとは大幅に同期が取れなくなり、深刻な問題が発生する可能性があります。たとえば、ゾンビサーバーがはるかに古い互換性のないソフトウェアバージョンを実行している場合、異なるスキーマのデータベースとやり取りしようとするとエラーが発生したり、間違った設定を使用したりする可能性があります。

ゾンビに対処するために、システムは現在実行中のソフトウェアバージョンでヘルスチェックに応答することがよくあります。次に、中央モニタリングエージェントがフリート全体の応答を比較して、予期しない古いバージョンを実行しているものを探し、これらのサーバーがサービスに戻らないようにします。

まとめ

サーバー、およびそれらで実行されるソフトウェアは、あらゆる種類の奇妙な理由で失敗します。ハードウェアは最終的に物理的に破損します。ソフトウェア開発者は、最終的にはソフトウェアを破損状態にする上記のようなバグを作成してしまいます。あらゆる種類の予期しない障害モードをキャッチするには、軽量のライブ状態チェックからサーバーごとのメトリクスのパッシブモニタリングまで、複数のレイヤーのチェックが必要です。

これらの障害が発生した場合、それらを検出し、影響を受けるサーバーを迅速にサービスから除外することが重要です。ただし、フリートの自動化と同様に、自動化をオフにし、不確実な状況または極端な状況で人間を巻き込むレート制限、しきい値設定、およびサーキットブレーカーを追加します。オープンに失敗して集中型アクターを構築することは、レート制限された自動化の安全性を備えたディープヘルスチェックの利点を享受するための戦略です。