WINDOWS AZURE キュー

2008 年 12 月

目次

1 はじめに

Windows Azure はマイクロソフトのクラウド プラットフォームの基盤であり、アプリケーション開発者がスケーラブルで可用性の高いサービスを記述するのに不可欠な構成要素を提供する、"クラウド向けオペレーティング システム" です。Windows Azure は以下のものを提供します。

- コンピューティングの仮想化

- スケーラブルなストレージ
- 管理の自動化
- 豊富な開発者向け SDK

Windows Azure ストレージを使用すると、アプリケーション開発者はデータをクラウドに格納することができます。そのため、アプリケーションではいつでもどこからでも任意の量のデータを任意の期間格納することができ、データに堅牢性がありデータが紛失しないことが保証されます。Windows Azure ストレージは、以下の優れたデータ抽象化セットを提供します。

- Windows Azure ブログ – 大きなデータ項目用のストレージを提供します。
- Windows Azure テーブル – サービスの状態を保持するための構造化ストレージを提供します。
- Windows Azure キュー – 非同期の作業ディスパッチを提供し、サービス通信を可能にします。

このドキュメントでは、Windows Azure キュー、およびその使い方について説明します。Windows Azure キューは、信頼できるメッセージ配信メカニズムを提供します。Windows Azure キューは単純な非同期の作業ディスパッチ メカニズムを提供し、このメカニズムは、クラウド アプリケーションのさまざまなコンポーネントを結び付けるのに使用することができます。Windows Azure キューは可用性、堅牢性、およびパフォーマンス効率が優れています。Windows Azure キューのプログラミング セマンティクスは、各メッセージを最低 1 回は処理できることを保証します。さらに、Windows Azure キューは REST インターフェイスを持つので、任意の言語でアプリケーションを記述することができ、アプリケーションはいつでも、インターネット上のどこからでも Web 経由でキューにアクセスすることができます。

2　Azure キューを使用してクラウド アプリケーションを構築する

Windows Azure キューを使用すると、クラウド アプリケーションの各部分を分離することができます。これにより、さまざまなテクノロジを使用してクラウド アプリケーションを容易に構築したり、トラフィック ニーズに応じてクラウド アプリケーションを容易にスケール変更したりできるようになります。

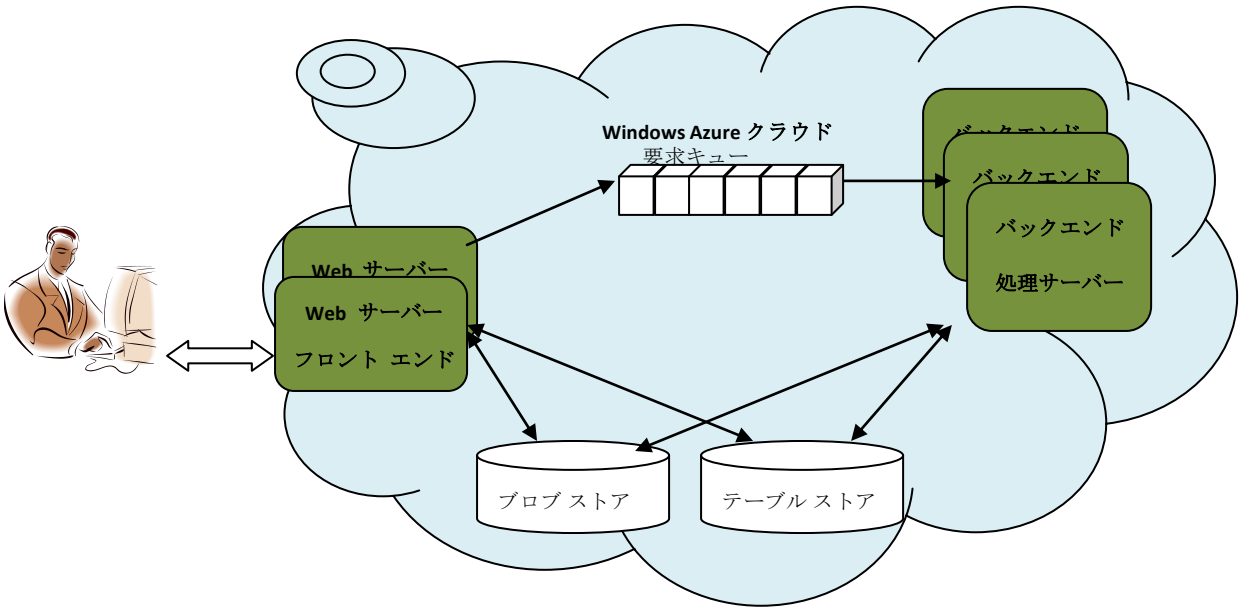

図 1: Azure キュー を使用してクラウド アプリケーションを構築する

上の図は、単純ですが一般的なクラウド アプリケーション シナリオを示しています。Web 要求の処理のフロントエンド ロジックをホストしている、一連の Web サーバーがあります。また、アプリケーションのビジネス ロジックを実装している、一連のバックエンド処理サーバーがあります。Web サーバーのフロントエンド ノードは、一連のキューを通じてバックエンド処理ノードと通信します。アプリケーションの永続的な状態は、Windows Azure ブロブ ストレージおよび Windows Azure テーブル ストレージに格納することができます。

例として、オンライン ビデオ ホスティング サービス アプリケーションを考えてみましょう。ユーザーはこのアプリケーションにビデオをアップロードすることができます。ビデオがアップロードされたら、アプリケーションでは、自動的にビデオ ファイルを異なるメディア形式に変換して格納することができます。さらに、アプリケーションは、ビデオの説明情報に自動的にインデックスを付けます。こうした情報に基づいて (たとえば、説明に含まれるキーワード、俳優、監督、タイトルなどに基づいて) 容易にビデオを検索できるようにするためです。

このようなアプリケーションでは、前述のアーキテクチャを使用することができます。Web フロントエンドはプレゼンテーション層を実装し、ユーザーからの Web 要求を処理します。ユーザーは、Web フロントエンドを通じてビデオをアップロードすることができます。ビ

デオ メディア ファイルは、ブロブ としてブロブ ストアに格納することができます。このアプリケーションでは、保持しているビデオ ファイルを追跡するための一連のテーブルを保持したり、検索に使用されるインデックスを保持したりすることもできます。バックエンド処理サーバーは、アップロードされたビデオ ファイルを異なる形式に変換してそれを ブロブ ストレージに格納する役割を担います。バックエンド サーバーは、テーブル ストレージ内にある、このアプリケーション用のテーブルを更新する役割も担います。フロントエンド サーバーは、ユーザーの要求 (たとえばビデオ アップロード要求) を受け取ったら、作業項目を生成してそれを要求キューに格納することができます。その後、バックエンド サーバーは、こうした作業項目をキューから取り出して、しかるべく処理することができます。各作業項目が正常に処理されたら、バックエンド サーバーではその項目をキューから削除する必要があります。別のバックエンド サーバーが重複して処理してしまうのを避けるためです。

以下に示すように、このアーキテクチャには、Windows Azure キュー の使用によるいくつかのメリットがあります。

1. **スケーラビリティ** - トラフィック ニーズに応じて、より容易にアプリケーションをスケール変更することができます。以下に示すように、メリットはいくつかあります。

   まず第 1 に、キューの長さは、バックエンド処理ノードの処理能力が全体的なワークロードに対してどれくらい十分であるかを直接反映します。キューが長くなっていくということは、バックエンド サーバーが作業を処理するスピードが不十分であることを示しています。その場合、アプリケーションでは、作業がより迅速に処理されるようにするために、バックエンド ノードの数を増やす必要があるかもしれません。キューの長さが常にゼロに近い場合は、バックエンドの処理能力が、トラフィックが必要とする処理能力よりも大きいということです。その場合、アプリケーションでは、リソースを節約するためにバックエンド ノードの数を減らすことができます。キューの長さを監視し、それに応じてバックエンド ノードの数を調整することにより、アプリケーションはトラフィックの量に応じて効率的にスムーズなスケール変更を行うことができます。

   第 2 に、キューを使用するとアプリケーションの各部分が分離され、アプリケーションの各部分を別々にスケール変更するのがより容易になります。この例では、フロントエンド サーバーとバックエンド サーバーが分離されており、両者はキューを

通じて通信します。これにより、アプリケーション ロジックに影響を与えることなく、バックエンド サーバーの数とフロントエンド サーバーの数を別々に調整することが可能になります。これにより、アプリケーションでは、リソースやコンピューターをそのコンポーネントに追加することによって重要なコンポーネントを容易にスケール アウトできるようになります。

第 3 に、キューを使用すると、アプリケーション内でリソースを効率的に使用できるという柔軟性がもたらされ、アプリケーションをより効率的にスケール変更できるようになります。つまり、優先度や重要性が異なる作業項目に対しては別々のキューを使用することができ、別々のバックエンド サーバー プールでこうした別々のキューを処理することができるということです。このようにして、アプリケーションでは、(たとえばサーバーの数の観点から) 適切なリソースを各プールに割り当てることができ、その結果、特性の異なるトラフィック ニーズを満たすように、利用可能なリソースを効率的に使用することができます。たとえば、他の作業が完了するのを待たずに先に処理できるように、ミッション クリティカルな作業項目を別個のキューに格納することができます。また、大量のリソースを消費する作業項目 (ビデオの変換など) は独自のキューを使用するようにすることができます。別々のバックエンド サーバー プールを使用して、このようなキューそれぞれに格納されている作業項目を処理することができます。アプリケーションでは、こうしたプールそれぞれのサイズを、そのプールに対するトラフィックに応じて別々に調整することができます。

2. **フロントエンド ロールをバックエンド ロールから分離する** - キューを使用することによって、アプリケーションの各部分が分離されます。これにより、アプリケーションの構築方法に大幅な柔軟性と拡張性がもたらされます。キュー内のメッセージは、標準的な形式でも拡張可能な形式 (XML など) でもかまいません。キューの両端で通信を行っているコンポーネントが、キュー内のメッセージさえ理解できていれば互いに依存しなくて済むようにするためです。

   異なるテクノロジやプログラミング言語を使用して、システムの各部分を最大限の柔軟性で実装することができます。たとえば、キューの片側に位置するコンポーネントを .NET Framework で記述し、もう一方の側に位置するコンポーネントを Python で記述することができます。

また、コンポーネント内の変更をシステムのその他の部分が意識する必要はありません。たとえば、まったく異なるテクノロジやプログラミング言語を使用して 1 つのコンポーネントを記述し直したとしても、キューを使用してコンポーネントが分離されているので、その他のコンポーネントに変更を加えなくてもシステムはシームレスに機能します。

これにより、アプリケーションをより新しいテクノロジにスムーズに移行させることができるようにもなります。キューを使用することにより、このようなシステム内に同じコンポーネントの異なる実装を共存させることができるからです。上記の例では、レガシテクノロジを使用して構築されたコンポーネントを段階的に減らしていき、それを新しい実装に置き換えることができます。古い実装と新しい実装を別々のサーバーで同時に実行することができ、古い実装と新しい実装は同じキューから取り出した作業項目を処理します。これらはすべて、アプリケーションのその他のコンポーネントが意識する必要はありません。

3. **トラフィックの集中** - キューは、トラフィックの集中を吸収し個々のコンポーネントのエラーが及ぼす影響を軽減するためのバッファー処理を提供します。前述の例では、短期間に大量の要求が届き、バックエンド サーバーがすべての要求をすぐには処理できない場合があるかもしれません。この場合、要求は失われるのではなくキュー内にバッファーされ、バックエンド サーバーはバッファーされたキューをそれぞれのペースで処理して、いずれ、すべての要求を処理することができます。これにより、アプリケーションでは、可用性を失うことなく爆発的なトラフィックを処理できるようになります。

   また、キューを使用すると、個々のコンポーネントのエラーが及ぼす影響が軽減されます。前述の例で、いくつかのバックエンド サーバーがクラッシュした場合、すべての作業項目が失われるのではなく、キューでは、バックエンド サーバーがダウンしている間に送信されたすべての作業項目をバッファーすることができます。バックエンド サーバーが復活したら、復活したサーバーでは、作業項目をキューから取り出して処理するという作業を続行することができ、いずれ、データがキューに格納されるペースに追いつきます。その他のコンポーネントがこうしたエラーを意識する必要はありません。バックエンド サーバーがクラッシュ時に処理していた作

業項目も失われないということを覚えておいてください。このような作業項目は VisibilityTimeout が経過したら再びキュー内に現れるからです。こうして、コンポーネントのエラーによるデータ損失が防止されます。これにより、アプリケーションでは、可用性を失うことなくエラーに耐えることができるようになります。

要約すると、キュー モデルは、アプリケーションの下にあるコンポーネントでたびたびエラーが発生してもアプリケーションのデータや可用性が失われないことを保証します。このモデルが適切に機能するようにするためには、アプリケーション記述者はキュー作業項目のバックエンド処理をべき等にする必要があります。これにより、作業項目を (エラーが原因で) 何度も部分的に処理し、最終的に完全に処理してキューから削除することが可能になります。

3 データ モデル

Windows Azure キュー には、以下のデータ モデルが用意されています。

- **ストレージ アカウント** – Windows Azure ストレージ へのアクセスはすべてストレージ アカウントを通じて行われます。
    - これは、キューおよびキュー内のメッセージにアクセスするための名前空間のうち最も上のレベルです。Windows Azure ストレージ を使用するためには、ユーザーはストレージ アカウントを作成する必要があります。これは、Windows Azure ポータル Web インターフェイスを通じて行われます。アカウントが作成されると、256 ビットのシークレット キーがユーザーに与えられます。その後、このシークレット キーは、ストレージ システムに対するユーザーの要求を認証するために使用されます。具体的には、このシークレット キーを使用して、要求に対する HMAC SHA256 署名が作成されます。HMAC 署名を検証してユーザーの要求を認証するために、要求ごとに署名が渡されます。
    - 1 つのアカウントは多くのキューを持つことができます。
- **キュー** – キューには多くのメッセージが格納されます。キュー名は、アカウントという名前空間の下にあります。
    1. キューに格納されるメッセージの数に制限はありません。

2. メッセージの格納期間は最大 1 週間です。格納期間が 1 週間を超えたメッセージはシステムによってガベージ コレクトされます。
3. キューにはメタデータを関連付けることができます。メタデータは名前と値のペアという形式で、メタデータのサイズは 1 つのキューあたり最大 8 KB です。

- **メッセージ –** メッセージはキューに格納されます。各メッセージのサイズは最大 8 KB です。大きなデータを格納するには、Azure ブロブ ストアや Azure テーブル ストアにデータを格納し、ブロブ やエンティティの名前をメッセージに格納します。メッセージをストアに格納する際は、メッセージ データはバイナリでかまいません。しかし、ストアからメッセージを取得する際は、応答は XML 形式であり、メッセージ データは base64 でエンコードされた形で返されます。**メッセージがどのような順序でキューから返されるかは決まっていません。また、1 つのメッセージが複数回返される場合があります。** Azure キュー サービスによって使用されるいくつかのパラメーターの定義を以下に示します。
    1. **MessageID**: キュー内のメッセージを識別する GUID 値です。
    2. **VisibilityTimeout**: メッセージの可視性のタイムアウト (秒単位) を指定する整数値です。最大値は 2 時間です。既定ではメッセージの可視性のタイムアウトは 30 秒です。
    3. **PopReceipt**: 取得されるメッセージごとに返される文字列です。この文字列および MessageID は、メッセージをキューから削除するために必要です。これは不明確なものとして扱う必要があります。形式や内容が今後変わる可能性があるためです。
    4. **MessageTTL**: メッセージの有効期間 (秒単位) を指定します。許容される最大有効期間は 7 日間です。このパラメーターが省略されている場合、既定の有効期間は 7 日間です。メッセージが有効期間内にキューから削除されない場合、そのメッセージはストレージ システムによってガベージ コレクトされ削除されます。

個々のキューの URI は、以下のような構成になっています。

http://<アカウント>.**queue**.core.windows.net/<キュー名>

ストレージ アカウント名がホスト名の最初の部分として指定され、その後に"queue" というキーワードが続きます。これによって、Windows Azure ストレージの中の、キューの要求を

処理する部分に要求が送信されます。ホスト名の後にはキュー名が続きます。アカウントとキューの名前付けに関しては制約があります (詳細については、Windows Azure SDK (英語) ドキュメントを参照してください)。たとえば、キュー名に "/" を含めることはできません。

4　キューの REST インターフェイス

Windows Azure キュー へのアクセスはすべて HTTP REST インターフェイスを通じて行われます。HTTP と HTTPS の両方がサポートされています。

アカウント レベルの HTTP/REST コマンドには、以下のようなものがあります。

- List Queues – 指定されたアカウントの下にあるキューをすべて列挙します。

キュー レベルの HTTP/REST コマンドには、以下のようなものがあります。

- Create Queue - 指定されたアカウントの下にキューを作成します。
- Delete Queue - 指定されたキューとその中身を完全に削除します。
- Set Queue Metadata - ユーザー定義のキュー メタデータを設定/更新します。メタデータは、名前と値のペアとしてキューに関連付けられています。このコマンドを実行すると、既存のメタデータがすべて新しいメタデータで上書きされます。Get Queue Metadata - ユーザー定義のキュー メタデータ、および指定されたキューに格納されているメッセージのおおよその数を取得します。

キュー レベルの操作は以下の URL を使用して行うことができます。

http://<アカウント>.**queue**.core.windows.net/<キュー名>

メッセージ レベルの操作を実装するためにサポートされている HTTP/REST コマンドには、以下のようなものがあります。

- PutMessage (QueueName, Message, MessageTTL) - キューの末尾に新しいメッセージを追加します。MessageTTL では、このメッセージの有効期間を指定します。メッセージは、格納する際はテキスト形式またはバイナリ形式でかまいませんが、取得する際は base64 でエンコードされた形で返されます。
- GetMessages(QueueName, NumOfMessages N VisibilityTimeout T) - キューの先頭から N 個のメッセージを取得し、取得されたメッセージを指定された期間 (VisibilityTimeout T) 見えない状態にします。この操作を実行すると、返されるメッセージのメッセージ ID が PopReceipt と共に返されます。**メッセージがどのような順序でキューから返**

されるかは決まっていません。また、1 つのメッセージが複数回返される場合があります。

- DeleteMessage(QueueName, MessageID, PopReceipt) - 指定された PopReceipt (前の GetMessage 呼び出しで返されたもの) に関連付けられているメッセージを削除します。メッセージが削除されていない場合、そのメッセージは VisibilityTimeout が経過したら再びキューに現れます。
- PeekMessage(QueueName, NumOfMessages N) - メッセージを見えない状態にすることなく、キューの先頭から N 個のメッセージを取得します。この操作を実行すると、返されるメッセージそれぞれのメッセージ ID が返されます。
- ClearQueue - 指定されたキューからすべてのメッセージを削除します。キュー内のメッセージがすべて削除されていることを確認するために、呼び出し元では、成功が返されるまでこの操作を再試行する必要があります。

メッセージ レベルの操作は以下の URL を使用して行うことができます。

http://<アカウント>.**queue**.core.windows.net/<キュー名>/messages

REST API の完全な定義については、Windows Azure SDK ドキュメントを参照してください。

5 キューの使用例

5.1 プロデューサー/コンシューマー シナリオ

以下の図は、Windows Azure キューのセマンティクスを明らかにするための例を示しています。

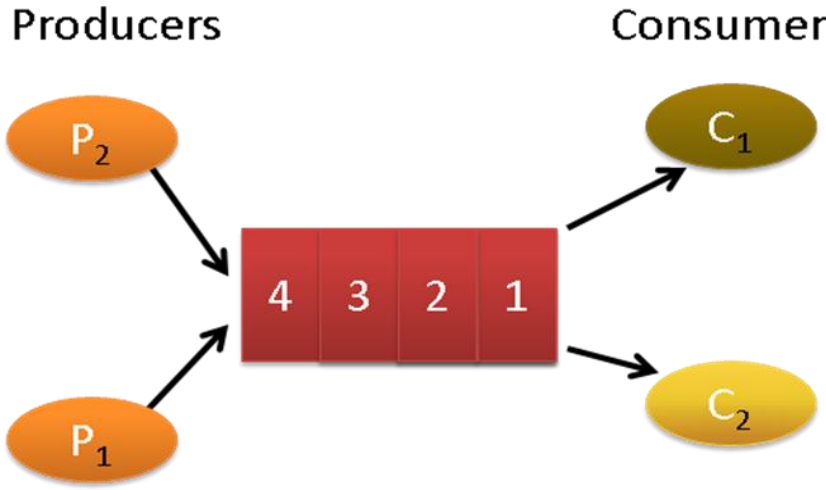

図 2: キューの使用例

この例では、上の図に示すように、プロデューサー (P1、P2) とコンシューマー (C1、C2) がメッセージ キューを通じて通信します。プロデューサーは作業項目を生成し、それをメッセージとしてキューに格納します。コンシューマーはメッセージ (作業項目) をキューから取り出し、処理します。プロデューサーもコンシューマーも、複数存在する場合があります。以下のような操作の流れを考えてみてください。

1. C_1 がキューからメッセージを取り出します。この操作を行うとメッセージ 1 が返され、メッセージ 1 は 30 秒間キュー内で見えない状態になります (この例では、既定の VisibilityTimeout である 30 秒が使用されていることを想定しています)。
2. 続いて、C_2 が別のメッセージをキューから取り出します。メッセージ 1 はまだ見えない状態なので、この操作ではメッセージ 1 は認識されず、メッセージ 2 が C_2 に返されます。
3. C_2 は、メッセージ 2 の処理を完了すると、メッセージ 2 をキューから削除するために Delete を呼び出します。
4. C_1 がクラッシュし、クラッシュ前にメッセージ 1 の処理を完了できなかったので、メッセージ 1 は C_1 によって削除されなかったとしましょう。
5. メッセージ 1 の VisibilityTimeout が経過したら、メッセージ 1 は再びキューに現れます。
6. メッセージ 1 が再びキューに現れた後で、C_2 がキューからメッセージを取り出すための呼び出しを行うと、メッセージ 1 を取得することができます。その後、C_2 はメッセージ 1 の処理を完了し、キューからメッセージ 1 を削除します。

この例で示したように、キュー API のセマンティクスは、キュー内のすべてのメッセージに、最低 1 回は処理が完了されるチャンスが与えられることを保証します。つまり、コンシューマーが、メッセージを取り出した後で、そのメッセージを削除する前にクラッシュした場合、そのメッセージは、VisibilityTimeout が経過したら再びキューに現れます。これにより、別のコンシューマーがそのメッセージの処理を完了することができるようになります。

5.2 REST 要求の例

このセクションでは、Windows Azure キュー によって使用される REST 要求を示します。ここで示す例では、キューは "myqueue" という名前で、"sally" というアカウントの下にあります。

5.2.1 REST の PutMessage

キューへの格納操作用の REST 要求の例を以下に示します。PUT という HTTP 動詞が使用されていることに注目してください。次に、省略可能な "messagettl" オプションが指定されています。このオプションでは、メッセージの有効期間 (秒単位) を指定します。許容される最大有効期間は 7 日間です。このパラメーターが省略されている場合、既定の値は 7 日です。この有効期間が経過すると、メッセージはシステムによってガベージ コレクトされます。Content-MD5 は、ネットワーク転送エラーを防ぎ整合性を確保するために指定することができます。この場合の Content-MD5 は、要求内のメッセージ データの MD5 チェックサムです。Content-Length では、メッセージ データ コンテンツのサイズを指定します。以下に示すように、HTTP 要求ヘッダー内には承認 (Authorization) ヘッダーもあります。メッセージ データは HTTP 要求の本文に含まれています。このメッセージ データはテキスト形式またはバイナリ形式の場合がありますが、メッセージを取得する際にはメッセージ データは base64 でエンコードされた形で返されます。

```
PUT http://sally.queue.core.windows.net/myqueue/messages
? messagettl=3600
HTTP/1.1 Content-Length: 3900
Content-MD5: HUXZLQLMuI/KZ5KDcJPcOA==
Authorization: SharedKey sally:       F5a+dUDvef+PfMb4T8Rc2jHcwfK58KecSZY+l2naIao=
x-ms-date: Mon, 27 Oct 2008 17:00:25 GMT
…… メッセージ データ コンテンツ ……
```

5.2.2 REST の GetMessages

キューからの取り出し操作用の REST 要求の例を以下に示します。GET という HTTP 動詞が使用されていることに注目してください。2 つの省略可能なパラメーターが指定されています。"numofmessages" では、キューから取得するメッセージの数 (最大 32 個) を指定します。既定では、1 つのメッセージがキューから取得されます。以下の例では、最大 2 個のメッセージが取得されます。"visibilitytimeout" では、メッセージの可視性のタイムアウト (秒単位) を指定します。メッセージは、このタイムアウト期間中はキュー内で見えない状態となり、可視性のタイムアウト期間が終わるまでに削除されなかった場合は再びキューに現れます。このタイムアウトの最大値は 2 時間で、既定値は 30 秒です。以下の例では、可視性のタイムアウトは 60 秒に設定されています。以下に示すように、HTTP 要求ヘッダー内には承認 (Authorization) ヘッダーもあります。応答は XML 形式であり、応答内のメッセージ データは base64 でエンコードされています (以下の例の <MessageText> タグと </MessageText> タグの間)。

```
GET http://sally.queue.core.windows.net/myqueue/messages
?numofmessages=2 &visibilitytimeout=60
HTTP/1.1
Authorization: SharedKey  sally:      QrmowAF72IsFEs0GaNCtRU143JpkfIlgRTcOdKZaYxw=
x-ms-date: Thu, 13 Nov 2008 21:37:56 GMT
```

この呼び出しに対しては、以下の例のような応答が返されます。

```
HTTP/1.1 200 OK
Transfer-Encoding: chunked
Content-Type: application/xml
Server: Queue Service Version 1.0 Microsoft-HTTPAPI/2.0
x-ms-request-id: 22fd6f9b-d638-4c30-b686-519af9c3d33d
Date: Thu, 13 Nov 2008 21:37:56 GMT

<?xml version="1.0" encoding="utf-8"?>
<QueueMessagesList>
  <QueueMessage>
```

<MessageId>6012a834-f3cf-410f-bddd-dc29ee36de2a</MessageId>

<InsertionTime>Thu, 13 Nov 2008 21:38:26 GMT</InsertionTime>

<ExpirationTime>Thu, 20 Nov 2008 21:36:40 GMT</ExpirationTime>

<PopReceipt>AAEAAAD/////AQAAAAAAAAAMAgAAAFxOZXBob3MuUXVldWUuU2VydmljZS5RdWV1ZU1hbmFn
ZXIuWEFDLCBWZXJzaW9uPTYuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49bnVsbAUBA
AAAVU1pY3Jvc29mdC5DaXMuU2VydmljZXMuTmVwaG9zLlF1ZXVlLlNlcnZpY2UuQXVldWVNYW5hZ2VyLlhBQy5
SZWFsUXVldWVNYW5hZ2VyK1JlY2VpcHQCAAAAFjxNc2dJZD5rX19CYWNraW5nRmllbGQgPFZpc2liaWxpdHlTdG
FydD5rX19CYWNraW5nRmllbGQDAAtTeXN0ZW0uR3VpZA0CAAAABP3///8LU3lzdGVtLkd1aWQLAAAAAl9hAl9i
Al9jAl9kAl9lAl9mAl9nAl9oAl9pAl9qAl9rAAAAAAAAAAAAAAAIBwcCAgICAgICAjSoEmDP8w9Bvd3cKe423ipfNapL
7xPLSAsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA=</PopReceipt>

<TimeNextVisible>Thu, 13 Nov 2008 21:38:26 GMT</TimeNextVisible>

<MessageText>.....</MessageText>

</QueueMessage>

<QueueMessage>

<MessageId>2ab3f8e5-b0f1-4641-be26-83514a4ef0a3</MessageId>

<InsertionTime>Thu, 13 Nov 2008 21:38:26 GMT</InsertionTime>

<ExpirationTime>Thu, 20 Nov 2008 21:36:40 GMT</ExpirationTime>

<PopReceipt>AAEAAAD/////AQAAAAAAAAAMAgAAAFxOZXBob3MuUXVldWUuU2VydmljZS5RdWV1ZU1hbmFn
ZXIuWEFDLCBWZXJzaW9uPTYuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibGljS2V5VG9rZW49bnVsbAUBA
AAAVU1pY3Jvc29mdC5DaXMuU2VydmljZXMuTmVwaG9zLlF1ZXVlLlNlcnZpY2UuQXVldWVNYW5hZ2VyLlhBQy5
SZWFsUXVldWVNYW5hZ2VyK1JlY2VpcHQCAAAAFjxNc2dJZD5rX19CYWNraW5nRmllbGQgPFZpc2liaWxpdHlTdG
FydD5rX19CYWNraW5nRmllbGQDAAtTeXN0ZW0uR3VpZA0CAAAABP3///8LU3lzdGVtLkd1aWQLAAAAAl9hAl9i
Al9jAl9kAl9lAl9mAl9nAl9oAl9pAl9qAl9rAAAAAAAAAAAAAAAIBwcCAgICAgICAuX4syrxsEFGviaDUUpO8KNfNapL
7xPLSAsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAA=</PopReceipt>

<TimeNextVisible>Thu, 13 Nov 2008 21:38:26 GMT</TimeNextVisible>

<MessageText>.....</MessageText>

</QueueMessage>
</QueueMessagesList>

5.2.3 REST の DeleteMessage

メッセージ削除 (DELETE) 操作用の REST 要求の例を以下に示します。今度は、DELETE という HTTP 動詞が使用されています。"popreceipt" パラメーターでは、削除するメッセージを指定します。先ほどの例で示したように、"popreceipt" は、キューからの前の取り出し操作から取得されます。

```
DELETE /sally/myqueue/messages/6012a834-f3cf-410f-bddd-
dc29ee36de2a?popreceipt=AAEAAAD%2f%2f%2f%2f%2fAQAAAAAAAAAMAgAAAFxOZXBob3MuUXVldWUuU2
VydmljZS5RdWV1ZU1hbmFnZXIuWEFDLCBWZXJzaW9uPTYuMC4wLjAsIEN1bHR1cmU9bmV1dHJhbCwgUHVibG
ljS2V5VG9rZW49bnVsbAUBAAAAVU1pY3Jvc29mdC5DaXMuU2VydmljZXMuTmVwaG9zLlF1ZXVlLlNlcnZpY2UuU
XVldWVNYW5hZ2VyLlhBQy5SZWFsUXVldWVNYW5hZ2VyK1JlY2VpcHQCAAAAFjxNc2dJZD5rX19CYWNraW5nR
mllbGQgPFZpc2liaWxpdHlTdGFydD5rX19CYWNraW5nRmllbGQDAAtTeXN0ZW0uR3VpZA0CAAAABP3%2f%2f%
2f8LU3lzdGVtLkd1aWQLAAAAAl9hAl9iAl9jAl9kAl9lAl9mAl9nAl9oAl9pAl9qAl9rAAAAAAAAAAAAAAAAIBwcCAgIC
AgICAjSoEmDP8w9Bvd3cKe423ipfNapL7xPLSAsAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA%3d&timeout=30
HTTP/1.1
Content-Type: binary/octet-stream
x-ms-date: Thu, 13 Nov 2008 21:37:56 GMT
Authorization: SharedKey sally:M/N65zg/5hjEuUS1YGCbVDHfGnI7aCAudkuTHpCDvZY=
```

6 ベスト プラクティス

Windows Azure ストレージで使用するアプリケーションを設計する際は、エラーを適切に処理することが重要です。このセクションでは、アプリケーションを設計する際に考慮する必要がある問題について説明します。

6.1 再試行のタイムアウトと "接続がホストによって閉じられた" というエラー

タイムアウトまたは "接続がホストによって閉じられた" という応答を受け取った要求は、Windows Azure ストレージによって処理されていない可能性があります。たとえば、PUT 要求がタイムアウトを返した場合、その後の GET では古い値または更新された値が取得される場合があります。タイムアウトまたは "接続がホストによって閉じられた" という応答が返された場合は、指数バックオフとしだいに長くなるタイムアウトを使用して、要求を再試行してください。

また、一部の削除操作 (キューのメッセージをクリアする操作など) はコストが高く、完了するのに時間がかかる場合があります。こうした操作がユーザー定義のタイムアウト期間内に完了しない場合、タイムアウト エラーがユーザーに返されることがあります。そのような場合、ユーザーは成功するまで要求を再試行する必要があります。

6.2 繰り返し発生するタイムアウト エラー向けにアプリケーションを調整する

アプリケーションとデータ センターとの間にネットワークの問題がある場合、タイムアウト エラーが発生することがあります。広域ネットワーク上では、1 つの大きな転送を一連のより小さな呼び出しに分割し、タイムアウト/エラーをこの単位で処理するようにアプリケーションを設計する (エラー発生後に再開して先に進み続けることができるようにするため) ことをお勧めします。

システムは、スケール変更でき大量のトラフィックを処理できるように設計されています。しかし、要求のペースが非常に速いと要求のタイムアウトが発生する場合があります。その場合、要求のペースを落とすとこの種のエラーが減少したり解消されたりすることがあります。一般に、ほとんどのユーザーは、このようなエラーにたびたびは遭遇しません。ですが、タイムアウト エラーがたびたび発生する場合や予期しないタイムアウト エラーが発生する場合は、MSDN の Windows Azure フォーラムを通じてお問い合わせいただければ、Windows Azure ストレージの使用を最適化し、アプリケーションでこの種のエラーが発生するのを防ぐ方法についてご説明します。

6.3 エラー処理とレポート

REST API は、標準的な HTTP サーバーのように機能し既存の HTTP クライアント (ブラウザー、HTTP クライアント ライブラリ、プロキシ、キャッシュなど) と対話するように設計されています。HTTP クライアントがエラーを適切に処理できるようにするには、各 Windows Azure ストレージ エラーを HTTP 状態コードにマップします。

HTTP 状態コードは Windows Azure ストレージ エラー コードよりも情報が少なく、エラーに関する情報が多くありません。しかし、クライアントは、Windows Azure ストレージのエラーは理解していませんが、HTTP のエラーは理解していますので、通常、エラーを適切に処理します。
したがって、エラーを処理する際や Windows Azure ストレージ エラーをエンド ユーザーにレポートする際は、HTTP 状態コードではなく Windows Azure ストレージ エラー コードを使用してください。Windows Azure ストレージ エラー コードには、エラーに関する最も多くの情報が含まれているからです。また、アプリケーションのデバッグを行う際は、人間が判読できる、XML 形式のエラー応答内にある <ExceptionDetails> 要素の内容も調べる必要があります。

6.4 GetMessage で使用する不可視期間を選択する

不可視期間の選択は、予想される処理時間とアプリケーションの回復にかかる時間との間のトレードオフです。

メッセージがキューから取り出されると、アプリケーションでは、そのキューからメッセージを取り出しているワーカーに当該のメッセージが見えないようにする時間の長さを指定します。この時間は、キューのメッセージによって指定された操作を完了するのに十分な長さである必要があります。

不可視期間が長すぎると、エラーが発生した場合、メッセージの処理を完了するのにかかる時間に影響があります。たとえば、不可視期間が 30 分に設定されていて、アプリケーションが 10 分後にクラッシュした場合、その後 20 分間は、そのメッセージの処理が再開されるチャンスはありません。

不可視期間が短すぎると、まだメッセージの処理が行われている間にそのメッセージが見える状態になってしまう可能性があります。したがって、複数のワーカーが同じメッセージを処理することになってしまう可能性があり、ワーカーはメッセージをキューから削除できない場合があります (次のセクションを参照してください)。アプリケーションは、以下のようにしてこれに対処することができます。

1. メッセージの処理にかかる時間が予測可能な場合は、不可視性のタイムアウトを、メッセージの処理が完了するのに十分な長さに設定します。
2. 処理時間は、メッセージの種類によって大幅に異なる場合があります。その場合は、メッセージの種類ごとに別々のキューを使用するとよいでしょう。こうすると、1 つのキュー内のメッセージは処理するのにかかる時間が同じくらいになります。その後、不可視性のタイムアウトの適切な値を各キューに設定します。
3. また、メッセージに対して実行される操作は、べき等であり再開可能であるようにしてください。効率を高めるためには、次のようなことを行うことができます。
    a. 重複作業を回避するため、不可視期間が終了する前に処理を中止する必要があります。
    b. メッセージの作業を小さなかたまりに分けて行い、不可視期間が短くて済むようにします。こうすると、再び見える状態になった作業が次にキューから取り出されたときに、その作業を続きから再開することができます。
4. 最後に、メッセージの不可視期間が短すぎて、キューから取り出されたあまりにも多くのメッセージが、削除できる前に再び見える状態になっている場合、アプリケーションでは、キューに格納される新しいメッセージ用に設定される不可視期間を動的に変更した方がよい場合があります。メッセージが見える状態になったことが原因でメッセージの削除が失敗している回数をワーカー ロールでカウントすることによって、これを検知することができます。その後、しきい値に基づいてそれをフロントエンド Web ロールに通知します。不可視期間を調整する必要がある場合に、フロントエンド Web ロールがキューに格納される新しいメッセージ用の不可視期間を長くすることができるようにするためです。

6.5 メッセージをキューから削除する

メッセージは、処理されたらキューから削除する必要があります。不可視期間内に処理が完了した場合、削除は成功します。

メッセージは、正常に処理された後でのみ削除する必要があります。それより前にメッセージを削除すると、アプリケーションがクラッシュした場合にメッセージの作業が完了されない可能性があります。

不可視期間が経過し、別のプロセスが同じメッセージをキューから取り出した場合、削除操作が失敗することがあります。この場合、アプリケーションは、不可視期間が経過したことを示すエラー コードを返します (これは HTTP 状態コード 400 です。また、拡張されたエラー コードは、"PopReceipt" の不一致に関するものです)。この古い PopReceipt を持つワーカー ロールは、このメッセージを無視して処理を続行します。このメッセージは再びキューから取り出されて処理されるからです。ここで重要なのは、不可視期間が終了する前にメッセージを削除する必要があるということです。

6.6 キューの長さに基づいてワーカー ロールを調整する

前述のとおり、キューの長さは、バックエンド処理ノードの処理能力が全体的なワークロードに対してどれくらい十分であるかを直接反映します。キューが長くなっていくということは、バックエンド サーバーが作業を処理するスピードが不十分であることを示しています。その場合、アプリケーションでは、作業がより迅速に処理されるようにするために、バックエンド ノードの数を増やす必要があるかもしれません。キューの長さが常にゼロに近い場合は、バックエンドの処理能力が、トラフィックが必要とする処理能力よりも大きいということです。その場合、アプリケーションでは、リソースを節約するためにバックエンド ノードの数を減らすことができます。キューの長さを監視し、それに応じてバックエンド ノードの数を調整することにより、アプリケーションはトラフィックの量に応じて効率的にスムーズなスケール変更を行うことができます。

6.7 メッセージにバイナリ形式を使用する

アプリケーションによっては、キューのメッセージにバイナリ データを格納する必要があるかもしれません。このようなアプリケーションでは、PutMessage API を呼び出して要求でバイナリ データを送信することができます。ただし、アプリケーションが "GetMessages" または "PeekMessages" を呼び出してメッセージを取得する際は、メッセージ データ、MessageID、および PopReceipt を含む応答は base64 でエンコードされています。したがって、アプリケーションでは、メッセージ データを使用する前に、GetMessages や PeekMessages から返された応答をデコードする必要があります。