修士輪講題目提案

05/14-藤居翔吾

1. データセンター・ネットワークにおけるEthenet Fablicに関する調査

Ethenet Fablic-スループットを最大化し, 広域化するための機能

・背景

現在, サーバの高密度化によって, 仮想サーバに対するIPトラフィックの増加している. 今後, ストレージの接続やライブマイグレーションなどのメモリー間のコピーのトラフィックなども乗り, 現状よりも大きいトラフィックに耐えられるような仕組みが必要.

だが, 既存のEthernetではスループットを伸ばしにくいという弱点がある. これは, Ethernetにおいて「スイッチは, 寿命のないEthernetフレームを, 単に転送するだけ」という仕組みが原因である. これによりループが存在するトポロジにおいては, Spanning Tree Protocol(STP)で, ブロック・ポートを設けることで, 対処しているが, これにより有用なパスが削られ, ボトルネックとなっている.

また, STPはリンクに異常が発生した場合, ツリーの再定義が必要となり, すべてのトラフィックが中断する.

さらにVMのマイグレーションや別のサーバへの移動の際, 従来のアクセス, アグリゲーション, コアの3 つの階層型のイーサネット・ツリートポロジにおいては, 制約が多い.

これらを解決するものが, Ethenet Fablic.

Ethenet Fablicでは, 低コストでシンプルなイーサネットのメリットを保持する一方で, ネットワークの構築に必要な機器の台数を減らし, 階層化されたネットワーク構造をシンプルにし, フラットで. 高速なネットワークを構築する.

物理サーバ1台で稼働するVM 数の増加に伴い, サーバあたりのトラフィック負荷が高まる

* サーバ間通信が増え, 3階層型の従来のネットワーク環境では性能が劣化
* VMモビリティはレイヤー2でのみサポートされることによる, VMの移動性の制限
* 3階層型アーキテクチャでの複雑性の増加と, STPの使用によるネットワーク・リソースの無駄

ISL Trunkingにより, 複数のリンクを論理的に1本としてみる. 10Gbpsを8本つなぎ80Gbpsにする

Fabric Shortest Path First（FSPF）と呼ぶルーティング・プロトコルを用いて, リンク・ステート・データベースを構築する. FSPFは, OSIルーティング技術の IS-ISやL3ルーティング・プロトコルの1つであるOSPFなどと同様に, Dijkstraアルゴリズムで経路表を構成する.

FSPFを使ったVCS(Virtual Cluster Switching)のルーティングでは, ループは起こらない. これは, RBridge IDと呼ぶ"L3"に相当するレイヤーが追加されているためである.

複数の同じコストのパスを同時に使用できれば, データセンターのスループット向上に大変役立つ. それでは, どのようなポリシーで, 複数のパスを切り分けるのだろうか. 現在のVCSでは, パスの割り当て基準は, トラフィック・フローのハッシュ値によって決定している.

WebSocket

近年, ブラウザ上では当たり前のように[Ajax](http://www.atmarkit.co.jp/aig/07wcr/ajax.html" \t "_blank)や[Comet](http://www.atmarkit.co.jp/aig/07wcr/comet.html" \t "_blank)を利用した, リアルタイムな表示の更新や, 通信が行われている.

Ajax

XmlHttpRequestオブジェクト（JavaScript）を利用した非同期通信処理のこと. Googleマップを筆頭とするWebアプリケーションで活用されている. ブラウザからサーバ側に非同期でリクエストを実施してデータを受け取り, JavaScriptにより部分的にページを更新することで, ページ全体を更新するよりも低負荷でサーバと通信できる. また, 通信中も操作を継続できるので, エンド・ユーザーの操作を妨げられることもなく, ユーザビリティに優れる.

デメリットとして, 基本的にブラウザからのリクエストで動作する仕組みなので, サーバ側から自動でデータを送信することはない.

これはAjaxがシングルスレッドで動くためである.

サーバに問い合わせる前に, 「問い合わせが帰ってきたときに呼ばれるコールバック関数」を定義しておき, 問い合わせをサーバに投げたら, その後は, 別の処理ができる.

Comet

Ajaxのデメリットである, ブラウザからのリクエストで動作する仕組みではなく, サーバ側からクライアント側にレスポンスを返すための仕組み. Ajaxの応用として利用されていたが, サーバ側で意図的にリクエストに対する保留状態を作り, サーバ側で何かしらの変更が発生した場合にレスポンスを返す仕組みのため, Ajaxに比べてリソース消費が大きいというデメリットもあった.

リアルタイム性をより重視するWebアプリケーションを開発しようとする場合にAjaxとComet, それぞれのデメリット部分がネックになってきていた. また, それぞれHTTPリクエストを利用しているため, 通信のたびにHTTPヘッダが付与されることでコネクション数に比例してトラフィックやリソー ス消費の増大も懸念されてきた.

そこでHTML5では, 「WebSocket」と呼ばれる新たな通信規格が追加された. WebSocketを利用することで, AjaxやComet の通信におけるデメリット部分を補い, より効率的にサーバとクライアント間の双方向通信ができるようになる. WebSocketの特長は, 以下の点である.

サーバとクライアント間は一度でも接続が確立すると, 明示的に切断しない限り通信手順を意識することなくデータのやり取りをソケット通信で実施できる

WebSocketで接続が確立しているサーバとすべてのクライアントは同じデータを共有し, リアルタイムで送受信できる

WebSocketは最初の接続時にそれ以降そのコネクションを使い続けるためにクライアントサイドからハンドシェイク要求を送る. サーバ側はハンドシェイク応答を返すことで1つのコネクションを使用し, 続ける仕組みになっている.

繰り返しになるが, WebSocketはハンドシェイクが成功した後は通信手順を意識することなくサーバとクライアントで双方向のデータの通信を実現できるので, 従来では工夫して実現していたリアルタイム性を求めるアプリケーションをより簡単に実現できる.

これまでWebSocketがもたらしてくれるステートフルの長所ばかりを取り上げていましたが, もちろん問題点もあります. それはスケールアウトが簡単にできない点です.

　これまでのWebサーバであれば, スケールアウトさせた場合は単純にWebサーバを何台も立ち上げれば良かったはずです（その分DBにアクセスが 集中するという副作用を生みがちですが）. 毎回のリクエストはステートレスであるため, どのサーバに接続しに行っても同じレスポンスが得られるはずです. サーバの前にロードバランサなどを置いておけば簡単に負荷分散が可能です.

　しかしながら, ステートフルなWebSocketサーバの場合はどうでしょう. 一度接続が確立すると, 同一のWebSocketサーバにつなぎっぱなしの状態になります. 最初1000ぐらいのアイドルなコネクションがつながっていて, 一気に大量のメッセージが送られて負荷が増えた際, ほかのサーバに負荷を切り替えようとすると, サーバの方で独自の仕組みが必要になってきます.

また, node-websocket-serverには, 現在サーバに接続しているすべてのメンバにメッセージをブロードキャストする機能があるが, すべての接続情報がメモリ上に保存されているため, 複数サーバ間のメンバにメッセージを送ることはできない. この場合はステートレスな既存のWebサーバと同じように, データベースや, 何らかのメッセージングシステムを経由してステートをシェアする必要が出てきます.

Websocketを用いたブラウザ間P2P通信

モニタリング

スケールアウト手法についての提案