5J-02

コンテナクラスタ向け分散ファイルシステムの開発と評価

平野　勇輝†　　田胡　和哉‡

東京工科大学バイオ・情報メディア研究科コンピュータサイエンス専攻†

東京工科大学コンピュータサイエンス学部‡

1. 背景

分散ファイルシステムは古くから使われてきた。近年はコンテナ技術の普及やクラウド化が進んでおり、分散システムや分散ストレージに対する要求も変化してきている。これに対応する試みの1つとして、筆者らは新しい分散ファイルシステムElton [1]を開発した。

Elton は可変データと不変データを分けて管理しており、この2つをUnion Mount してアプリケーションからは透過的に扱えるインターフェースを提供する。不変データの分散共有はキャッシュの一貫性維持が不要であるため、キャッシュシステムを簡素化できるなどのメリットがある。Elton はこのような性質を利用し、パフォーマンス向上と同期トラフィック削減の実現をする。本稿ではEltonの実用化を目的とし、これまでのEltonのプロトタイピングの経験を反映し、実運用で必要な機能の追加、および適した利用法の提案を行う。

2. 利用パターン

Elton を導入して効果が出る場面として、コンテナクラスタのストレージへの適用を提案する。

2.1. コンテナクラスタへの適用

コンテナとは、OS のリソース分離機能を使い、アプリケーションを軽量な隔離環境の中で実行する仕組みである。アプリケーションのデプロイに関する諸問題を解決出来るなどのメリットがあるため、コンテナの利用が増えている。これまでのコンテナの使い方は、アプリケーションをコンテナ内で起動させ、デーモンとして長時間動かし続けるのが主であった。そこでは、コンテナの寿命は長いのが一般的である。一方、最近は多様なコンテナを頻繁に起動し、処理完了後は速やかにコンテナを破棄する使い方が増えている。このような使い方をするサービスの代表例が、FaaS (Function as a Service)である。

また、継続的なアプリケーションの改善が求められていることにより、アプリケーションのデプロイ回数が増えている。このようなシステムでは、コンテナがすばやく起動し、速やかにアプリケーションが処理開始できることが不可欠である。しかし、条件次第ではコンテナの起動が遅くなるケースがある。コンテナの起動プロセスは、下記の3つの処理が行われる。

- Pull - アプリケーションの実行環境をダウンロード
- Create - コンテナを作成、ネームスペース作成やファイルシステムのマウントを行う
- Start - コンテナ内でアプリケーションを起動

通常は、Pullがスキップされ、CreateとStartが実行される。これらの2つの操作は短時間で実行可能であるため、短時間でコンテナが起動する。しかし、必要なイメージがノード内に存在しなかった場合はPull も実行される。この Pull 操作は時間がかかる処理であり、コンテナの起動が遅くなる原因である。

この場面にEltonを適用すると、Create操作中にEltonfsをマウントすることで、Pull操作を省略することが出来る。Elton は分散ファイルシステムであるため、ファイルシステムをマウントすると、即座にファイルシステム上に保存された全データがアクセスできる状態になる。Pull操作とは異なり、実際に必要となるファイルだけが取得されるため、Pull 操作に起因するコンテナ起動速度の低下が解消できる。

もう一つEltonの特徴的な点として、キャッシュの参照が高速なところである。ノード間で共有しているデータは不変であるため、キャッシュの一貫性維持のためのオーバーヘッドが生じない。このため、必要なデータがキャッシュされている状況下では、ローカルのファイルシステムと同等の性能でのアクセスが可能となる。

Development and evaluation of a distributed filesystem for container clusters

† Yuki HIRANO
School of Bionics, Computer and Media Sciences, Computer Science Program, Tokyo University of Technology Graduate School.
‡ Kazuya TAGO
School of Computer Science, Tokyo University of Technology.

3. 設計

3.1. 全体像

実用化において、信頼性、性能、スケーラビリティ、拡張性、保守性の5つが必要である。これらを満たせるよう注意して、設計を行った。

拡張性や保守性を高めるため、複数のサブシステムで構成した。サブシステムとその役割は、下記の3つである。

- Eltonfsサブシステム - ファイルシステムのインターフェースを提供する
- ストレージサブシステム - データの永続化と広域分散データ共有機構を提供する
- コントローラサブシステム - クラスタ管理やトランザクション制御などをする

各サブシステムの間は、RPC で接続されている。このような構成にすることで、システムのカスタマイズや拡張を行っても、システムに対する影響を局所的かつ最小限に留めることが期待できる。サブシステム毎に異なる実装に入れ替えることができるため、様々な要件にも対応しやすい。特にストレージに対しては、広域分散データ共有機構とバックエンドストレージの実装を分離し、容易に入れ替え可能にする。

信頼性や性能などは各サブシステム内で解決する設計にしたため、Elton のシステム構成を大きく変更すること無く、様々な要求に対応することが出来る。

3.2. 書き込み性能の向上

プロトタイプのEltonでは、読み出し性能の向上と引き換えに、同時書き込み不可の制約を課していた。これは一貫性を維持するための処置であるが、書き込み性能がスケールしないので利用できる場面が狭まってしまう。

同時書き込みの制約を課しているのは、ファイルシステムがファイルの内容を理解していないため、競合した書き込み操作を適切にマージできないからである。ファイルシステムに適切な対処法を指示する仕組みがあれば、この制約を緩めることが出来る。

Elton では、個々のファイルやディレクトリに対して、一貫性を維持しながらマージ可能な条件を指定できるようにした。トランザクションが競合した場合、全てのマージ可能条件を満たしているなら2つのトランザクションが自動的にマージされ、競合状態が解決できる。

マージ可能条件は、ファイルやディレクトリにuser.elton_merge_mode拡張属性を付与することで指定できる。

4. 実装

Eltonfs サブシステムは、カーネルモジュールとして実装した。read/write のハンドリングやキャッシュ参照など、頻繁に実行される処理をカーネル内で実行できるため、高速な動作が期待できる。通信やコミット関連の処理はユーザ空間で動作するプロセスで実行することにより、開発効率と性能のバランスを取った設計にした。

ストレージサブシステムは、ID の生成機構を工夫している。Eltonは、オブジェクトをIDにより一意に識別しているが、この ID が万が一衝突してしまうと不整合が発生してしまう。ID の衝突を検出する機構も実装されていないため、データ不整合による予期せぬ障害を引き起こす可能性がある。このようなトラブルを防ぐために、時刻ベースの分散 ID 生成アルゴリズムの SonyFlake [2]を採用した。

5. ベンチマーク

EltonfsとNFSv4.2とext4の3種類のファイルシステムで、Linux Kernel v5.4.8のソースコードの読み書きに要する時間を測定した。ベンチマーク結果を下図に示す。

	write	read
eltonfs	4.7	Cache hit: 5.4 Cache miss: 146.7
ext4	4.9	11.5
nfs	74.5	41.6

書き込みに関しては、ローカルファイルシステムと同程度の性能が出せている。読み込みは、ローカルキャッシュにヒットすれば高速である。キャッシュミスしたときの性能が悪いので、実装の改善が必要である。

6. 今後の展望

今後は安定性の改善、および Docker や Kubernetes との連携を取り、コンテナクラスタに適用可能な状態にする。

7. 参照文献

1. 水野拓. 粗結合マルチクラスタ向き分散ストレージ Elton の開発. 東京工科大学 修士論文 2015年度.
2. sony/sonyflake. GitHub, https://github.com/sony/sonyflake, 2020年01月07日.