多ノード間ファイル同期アルゴリズムの設計と実装

星野 喬 田浦 健次朗 近山 隆
東京大学 東京大学 東京大学

1 はじめに

複数のサブネットにまたがった、多数のノードで、大量のデータを同期させるシステムについて述べる。

本システムは、広域に分散した多数のノード間でプログラムやそれが用いるデータを、少ない手間で配布・更新するのに用いることができ、今後主流となる分散環境 [4] でのプログラム開発、インストール、計算機管理の効率化などに貢献する。

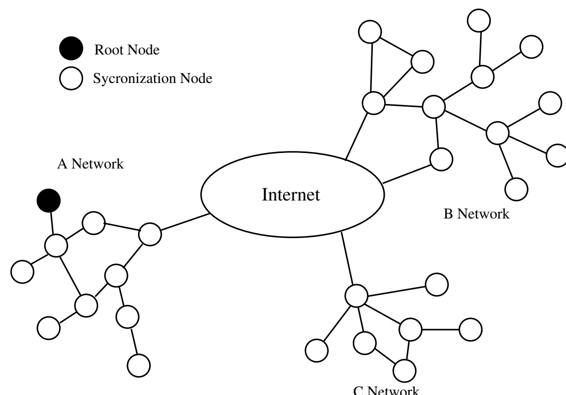

図 1: A-net と B-net と C-net を含むネットワーク

広域に分散した多数のノードで効率的にデータを同期するには、転送路の選択が重要である。

例として、図 1 のような状況では、サブネット間のデータ転送は最小限にして、あとは、サブネット内でファイル同期するのが明らかに効率的である。つまり、無駄なサブネット間のデータ転送を少なくできるようにしたい。

上記のような、効率的で信頼性のある、インターネットを経由する場合も含めて多ノードにファイルを同期するためのプログラムを設計・実装することが本研究の目的である。無駄なトラフィックが最小限になるようなシステムであると同時に動作開始から出来るだけ短時間で転送を完了させるシステムが理想である。

2 モデル化と問題点

本研究では、ネットワークトポロジーがはっきりとは分からないという前提に立つ。traceroute などのコマンドを使えばトポロジーを知ることはできるが、トラフィックが大きく、時間もかかる。また、セキュリティ上の理由から traceroute のパケットを通さないルータが増えてきている。

物理トポロジーが不明なので、大きなデータを同期するときは実際のネットワーク線を転送路が何回も往復してしまうことによりボトルネックが生じやすくなる。これを決定的に回避することは難しい。スループットを測定したりして最適ツリー構築アルゴリズムなどを使えばボトルネックを減らすことが出来るが、測定コストが大きいし、スループットは変動するのでやっかいである。

また、ノード数が多ければ多いほど、ネットワークや各ノードの可用性はコストの問題も含めて低くなってしまう。よってデータ転送中に一部で故障が発生しても全体の動作が停止してしまったりせず、復旧後も速やかに転送を再開してくれるようなやり方が望ましい。

3 多ノード間ファイル同期アルゴリズムの設計

本アルゴリズムはそのような要求に応えるために、次のような特徴がある。

- 耐故障性に優れており、長時間の転送を必要とする同期 (例えば大容量のデータの同期や細いリンクをまたいだ同期) を効率よく行える。途中でノードやネットワークに故障が発生しても正常なノード群は同期を続けることが出来るし、故障から復旧すれば、自動的に中断したところから同期を再開することが可能である。これは、後述するように、ノードが自己組織化によって自律的に転送路を構成することによって達成される。
- 多数のノードに対応できる。非常にスケーラブルなアルゴリズムである。
- 非常に単純なアルゴリズムであり、複雑になりやすく実装が難しい分散アルゴリズムの中で実装が比較的容易である。
- 各ノードがどのノードと繋がるか選択することによって、局所的ではあるが最適化を図ることが出来る。
- サブネットをまたがったノード間での同期ができる。多くのクラスタがそうであるように、NAT ルータの背後にあるノード間でも同期が可能である。

以下、アルゴリズムの詳細を説明する。

3.1 分散転送ツリー構築アルゴリズム

各ノードの動作を表す擬似コードを図 2 で示した。

同期命令を受けとったノードは、自分がそのデータをどれだけ持っているかを調べ (1–11 行目)、自分より

```
 1: parent = null;
 2: children.dellAll();
 3:
 4: // COMPLETE: データが完全にある。
 5: // SYNCING:  親がいて、データを同期中である。
 6: // ALONE:    親がおらず、データがない、または不完全。
 7: if (having complete data) {
 8:     offset = file_size; state = COMPLETE;
 9: } else {
10:     offset = get_now_size(file_name); state = ALONE;
11: }
12:
13: while (true) {
14:     if (state == ALONE) {
15:         dst = any process id; send(dst, ask(id, offset));
16:     }
17:
18:     // received(message(wid, ...)) =>
19:     // wid からメッセージを受けとったら以下の動作をする。
20:     received(ask(wid, woffset)) =>
21:         if ((offset > woffset)
22:         && (MAX_NODE > children.size)) {
23:             children.push(wid, woffset);
24:                send(wid, ok(id, offset));
25:         } else {
26:             send(wid, ng(id));
27:         }
28:     received(ng(wid)) =>
29:         state = ALONE; parent = null; continue;
30:     received(ok(wid, woffset)) =>
31:         parent = wid; state = SYNCING;
32:         send(parent, giveme(id, offset));
33:     received(data(parent, woffset, size, buf[]) =>
34:         save(file_name, woffset, size, buf[]);
35:         offset += woffset;
36:         if (offset == file_size) {
37:             state = COMPLETE;
38:                send(parent, complete(id)); parent = null;
39:         } else {
40:             send(parent, giveme(id, offset));
41:         }
42:     received(giveme(wid, woffset)) =>
43:         if (offset > woffset) {
44:             size = max(BLOCKSIZE, offset - woffset);
45:             load(file_name, woffset, size, buf[]);
46:             send(wid, data(id, woffset, size, buf[]));
47:         } else {
48:             send(wid, ng(id));
49:         }
50:     received(complete(child)) =>
51:         children.del(child);
52:
53:     if (error has occured(parent.died())) {
54:         state = ALONE; parent = null;
55:     }
56:     if (error has occured(child.died())) {
57:         children.del(child);
58:     }
59: }
```

図 2: 分散ツリー構築アルゴリズムの擬似コード

多くのデータを持っているノードを親候補として探す (14–16 行目)。どれだけデータを持っているかは、offset で表される。$0 \leq offset \leq file_size$ である。転送ツリーの親子関係は、必ず $child's\ offset \leq person's\ offset$ が成り立っている (21、43 行目)。

親子関係が成立したら、子は持っていないが親は持っているデータを親から子へ転送する (33–51 行目)。ここで注目すべきことは、あるノードは、同時に親と子になることが出来るということである。大きな視点から見れば、これはパイプライン転送と考えることが出来る。あるノードで故障が起こったら、その転送に関与していたノードは、そのノードを切り離すだけで良い (53–58 行目)。故障から復旧したノードは、このアルゴリズムに沿って動作し、中断したところから再び転送に参加することが出来る。

3.2 ボトルネックの回避

ボトルネックを事前に最小化した転送路を作ることは困難で、ボトルネックを減らすためにスループットを事前に測定するのはコストがかかる。よって、始めはランダムに木を作り転送路を確保した上で、転送中にスループットを測定し、その逆数を重みとするグラフ構造において転送ツリーの全体または一部を MST[1, 2] に近づけていくようにする方針である。これは局所的には各ノードがスループットが大きい親を選ぶことによって、トラフィックの無駄を最小限にしていくことである。

また、ノードの IPv4 アドレスの上位ビットを比較することにより、ある程度は物理的なネットワークトポロジーを推測出来ることを利用する。

4 アルゴリズムの検証と Java での実装

トラフィック削減、同期スピード向上、といった最適化の部分を抜きにして、耐故障性のある分散ツリー構築アルゴリズムを SPIN [5] という並列分散アルゴリズム検証ツールを使い、検証した。

その後、Java にてこのアルゴリズムを実装したプログラムを作成した (現在 3000 行程度)。現在、ランダムにツリーを作って、データを転送することが可能で、NAT 環境にも対応している。

このプログラムを実際に Solaris (sparc)、Linux (x86)、Windows (x86) そして Tru64Unix (Alpha) マシンで動かし、耐故障性があることを確認している。実際にデータを同期させることも出来た。最適化の部分は実装中である。

5 終わりに

多ノードで大容量データを、耐故障性を考慮して効率的に同期しようとする本アルゴリズムは、未だ実装中ではあるが、同期の自動化は既に出来るため、実用的に動作させることが可能である。今後、効率化のフィーチャーを組み込みさらに便利なものにできるだろう。

現在、セキュリティに関する配慮はこのアルゴリズム自体にはない。ただ、ssh や ssl を使用すれば、とりあえずの安全は得られると考える。

IPv4 の代わりに IPv6 [3] を使用したときのことを考えると、ネットワークトポロジーが IPv6 ではアドレスからほぼ推測できるので、もっと効率的な転送が可能になるのではないかと考えている。しかし、ネットワークが IPv4 と共存している間は、IPv6 はトンネリングなどのテクニックを使って実装されているためこのメリットはまだ生かせないだろう。

参考文献

[1] 石畑 清 “アルゴリズムとデータ構造” 岩波書店, pp. 223–296, 1989.
[2] 亀田 恒彦, 山下 雅史 “分散アルゴリズム” 近代科学社, pp. 51–77, 1994.
[3] 江崎 浩, 関谷 勇司, 吉藤 英明, 石原 知洋 “詳説図解 IPv6 エキスパートガイド” 秀和システム, 2002.
[4] Andrew S. Tanenbaum, Maarten van Steen “Distributed Systems: principles and paradigms” Prentice Hall, 2001.
[5] Gerard J. Holzmann “The Model Checker Spin” IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997, pp. 279–295.