Vol.2023-IOT-60 No.7
2023/3/15

共有リモートファイルシステムを用いたコンテナイメージ提供システムの実現とその性能評価

出口 弘大[1,a)]　桝田 秀夫[2]　森 真幸[2]　永井 孝幸[2]

概要：現在のコンテナ型仮想化技術の主な利用方法では，遠隔にあるレジストリ上のコンテナイメージを，ローカルのストレージ上に展開する所謂プルと呼ばれる操作を行う必要がある．しかしプル自体に時間がかかる点や，ストレージ容量の多くがイメージの保存に使われるにも関わらず，大部分は全くアクセスされない点が問題となる．本論文では，コンテナイメージを共有してプルの過程を一部省略できるように，ネットワーク経由で直接コンテナイメージを利用できる仕組みを提案する．コンテナランタイムに対して，複数のマシン間でメタデータを共有するためのプラグインを実装した上で，コンテナイメージの共有に適したリモートファイルシステムおよびこれを有するリレーサーバーの実装を行い，性能を評価した．

Implementation and Performance Evaluation of Container Image Provisioning System Using Shared Remote File System

1. はじめに

現在のコンテナ型仮想化技術の主な利用方法では，遠隔にあるレジストリ上のコンテナイメージを，ストレージ上にダウンロード・展開する所謂プルと呼ばれる操作を行った上で，ようやくコンテナを実行することができるようになる．しかしこの利用方法では，プル自体に時間がかかってしまう点や，ストレージ容量の多くがコンテナイメージの保存に使われるにも関わらず，大部分は全くアクセスされない点 [1] が問題となる．そこで本論文では，コンテナイメージを共有してプルの過程を一部省略できるよう，リモートファイルシステム上のコンテナイメージを直接利用できる仕組みを提案する．コンテナランタイムの 1 つである containerd に対して，複数のノード間でメタデータを共有するためのプラグインを実装した上で，コンテナイメージの共有に適したリモートファイルシステムおよびこれを有するリレーサーバーの実装を行い，性能を評価した．

2. 関連技術

本章では，本研究に関連する技術について述べる．

2.1 OverlayFS

現在コンテナ型仮想化技術に多く用いられているファイルシステム OverlayFS[2] は，上層側と下層側のディレクトリを透過的に重ね合わせる機能を持ったファイルシステムである．下層側のディレクトリは複数指定することができ，指定されたディレクトリの中にもさらに上下関係が存在する．例えば上層側のディレクトリに upper，下層側のディレクトリに lower1,lower2,...lowern を指定した場合，下層から順に lower1,lower2,...lowern,upper の順番で重ね合わせが起こる．また下層側に対してはファイルの読み込みに関する操作しか発生しないが，上層側では書き込みも発生しうる．

OverlayFS のマウントには下層側の指定は必須項目だが，上層側の指定は任意である．ただし上層側を指定しなかった場合には，読み込み専用のファイルシステムとしてマウントされることになる．

コンテナイメージのレイヤーはここで言うところの下層側に相当し，この下層側を読み込み専用のリモートファイルシステムとして読み込んでくるのが本研究の主旨である．

2.2 containerd

containerd[3] はコンテナランタイムの 1 つである．containerd はプラグインと呼ばれる単位で機能が分割されて

1　京都工芸繊維大学 工芸科学部 情報工学課程
2　京都工芸繊維大学 情報科学センター
a)　b9122034@edu.kit.ac.jp

ⓒ2023 Information Processing Society of Japan

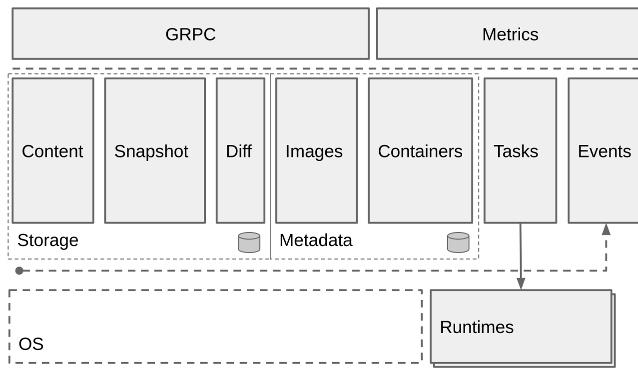


図 1: containerd のアーキテクチャ
Fig. 1 containerd architecture

おり，複数のプラグインによって構成されるアーキテクチャを採用している．アーキテクチャの概要を図 1[*1]に示す．また各プラグインに関して，図 1 にあるようにプラグインは Content,Snapshot,Metadata... というように複数の種類に別れている．以降プラグイン名を記述する際には "種類：実装" の書式で記述するものとする．

また containerd において Snapshot とは，コンテナのレイヤーに相当するものである．Snapshot には

- Committed
- View
- Active

の 3 つの種類が存在する．Committed Snapshot はコンテナイメージのレイヤー部分に相当し，OverlayFS ではマウント時に下層側として指定される．また Active Snapshot はコンテナの書き込み可能レイヤーに相当し，OverlayFS では上層側として指定される．

View Snapshot のみ特殊な Snapshot であり，Snapshot としてのメタデータしか存在せず，他とは違ってファイルとしての実体が存在しない．OverlayFS では先述した上層側の指定を省略した場合に相当するため，コンテナのファイルシステムは読み込み専用となる．

3. 関連研究

3.1 シーカブルな tar.gz アーカイブ

現在 Open Container Initiative(OCI) や Docker で定義される主流なコンテナイメージは，レイヤーのアーカイブ形式として tar.gz 形式を採用している．しかしこの tar.gz アーカイブは，1 つの tar アーカイブをそのまま gzip で圧縮しており，tar 中の各エントリをシークできない形式となっている．そのためレイヤーから 1 つのファイルを読み出すだけでも，レイヤーが含まれる tar.gz アーカイブ全体を取得する必要がある．Google 社が開発した crfs[4] は，シーカブルな tar.gz 形式として提案された stargz 形式を前提として実装された，コンテナレジストリそのものを読み込み専用の FUSE ファイルシステムとマウントできるファイルシステムである．

stargz 形式のコンテナイメージをコンテナランタイムの 1 つである containerd 上で動かすために，stargz-snapshotter[5] が実装されている．また stargz 形式の改良版として estargz と呼ばれるアーカイブ形式が実装されている．estargz 形式では Prioritized Files と呼ばれるアクセスされやすいファイルが，アーカイブの先頭に配置されるようになっている．

3.2 starlight

さらに stargz,estargz 形式の改良版として starlight 形式が提案 [6] されている．starlight 形式も似たようなアプローチを取っているが，次のような違いがある．

- ToC はアーカイブ中に含まれず，コンテナレジストリの側にあるデータベースに保管される．
- アーカイブ中の各エントリは，チャンク単位ではなくファイル単位で管理される．また各エントリから tar ヘッダと tar フッタが省略される．

さらに無駄なパケットのラウンドトリップを減らすため，starlight は Delta Bundle Protocol と呼ばれる HTTP ベースの独自のプロトコルを採用しており，1 度のリクエストでコンテナの起動に必要なデータがすべて取得できるよう実装されている．

4. 検討

4.1 コンテナイメージをネットワーク経由で利用する

コンテナイメージは，ネットワーク経由により全てのノード間で共有されているものとする．より具体的には，コンテナイメージを構成するファイルはリモートファイルシステムによって共有し，構成するメタデータは分散型 key-value ストアの 1 つである etcd[7] を用いて共有することにする．

またコンテナイメージ以外でコンテナ作成に必要なリソースは，非共有部としてノード間で共有せず，各ノード上で個別に作成されるものとする．非共有部のリソースには次が該当する．

- Active Snapshot
- コンテナそれ自体のメタデータ

これを実現するために，非共有部のメタデータ管理には，公式の containerd が採用している key-value ストアの bbolt[8] を用いることにする．

4.2 コンテナイメージに関する操作を行うためのリレーサーバー

多くのコンテナイメージは，主にサイズの小さいファイルによって構成されている．そのためリモートファイルシ

[*1] https://raw.githubusercontent.com/containerd/containerd/main/docs/historical/design/architecture.png

ⓒ2023 Information Processing Society of Japan

ステムとして，ファイル単位で共有されるファイルシステム（一例として Linux カーネルに実装されている NFS など）を採用した場合，大量の小さいファイルを書き込むプルには非常に時間がかかる．そこで別途，コンテナイメージに関する操作を行うためのリレーサーバーを提案する．リレーサーバーからは，コンテナイメージを保管するファイルシステムに対してブロック単位の高速な書き込みが必要であり，本論文ではリモートファイルシステムのサーバーが動いているノードと同一のノード上で動かすものとする．

実際にあるノードがコンテナイメージをプルするとき，リレーサーバーに対して

- コンテナイメージ名
- 名前空間
- プラットフォーム

といったコンテナイメージのプルに必要な情報をリクエストとして送る．リレーサーバーはリクエストを受けると，配下で動いている containerd のソケットに接続してプルを実行する．

またセキュリティの観点から見ても，リレーサーバーの実装は有用である．コンテナイメージのプルは，コンテナイメージを保管するファイルシステムに対する唯一の書き込み操作であるから，プルがリレーサーバーによって処理されるとなれば，リモートファイルシステムに対する書き込みを禁止して，読み込み専用として公開することができるようになる．これにより，コンテナイメージとは無関係なファイルの書き込みを防ぐことができるようになる．

4.3 インデックスの作成・読み込み

Linux ではパスを処理してファイルに紐づけられた dentry を取得するために，Lookup と呼ばれる操作を再帰的に行う [9]．このような動作は，レイテンシの高いネットワークにおいて大きな問題となり得る．これに加えて OverlayFS の透過的にディレクトリを重ね合わせるという性質上，大量の Lookup 操作がリクエストされることになる．

この大量の Lookup 操作のリクエストを，愚直にそのままリモートファイルシステムに問い合わせるのは，あまりに非効率である．そこでコンテナイメージをプルした際に，各レイヤーに対して 3.1 節にて説明した ToC を作成するように変更を加える．そして OverlayFS のマウント前に，作成した ToC を読み込んでファイルのメタデータをメモリ上に保存しておき，アンマウント後には読み込んだ ToC がメモリから解放されるようにしておく．

5. 実装

5.1 概要

全体のアーキテクチャ図 2 を参考に，コンテナを動かすワーカーノードが，実際にコンテナを作成・実行するまでの流れを説明していく．以降の説明に出てくる

- containerd-relay-server
- ctrnfs Service
- ctrnfsd

という用語は，本論文での実装の一つであり，詳細は 5.3 節,5.4 節にて記述する．

コンテナイメージのプルは以下のようにして行われる．

① ワーカーノードが containerd-relay-server に対してプルリクエストを送る．リクエストの内容は名前空間，イメージ名，プラットフォームから構成される．

②③ ctrnfs Service 上の containerd がプルを処理する．

④ containerd-relay-server からプルされたコンテナイメージに関するメタデータがレスポンスとして返ってくる．

またコンテナの実行は次のように行われる．

⑤ ワーカーノードが etcd からコンテナイメージのメタデータを取得してくる．

⑥ ワーカーノードのローカルストレージ上に OverlayFS の上層側となるディレクトリを作成し，作成されるコンテナのメタデータを bbolt に書き込む．

⑦⑧⑨ ctrnfsd に対して MountOverlay リクエストが行われる．リクエストには OverlayFS マウントを行うために必要なオプションに加えて，下層側に対応する ToC へのパスが，オプションとして指定される．OverlayFS の下層側には FUSE マウントされた ctrnfs 上のディレクトリが指定され，上層側には先程作成したローカルストレージ上のディレクトリが指定される．リクエストを受けると ctrnfsd は，指定された各 ToC の読み込みを行った後，OverlayFS のマウントを行う．最後にマウントポイントをコンテナのルートファイルシステムとして，containerd が通常通りコンテナの作成を行う．

5.2 containerd

公式の containerd が実装した Metadata:bolt プラグインとは別に，etcd を主な key-value ストアとして採用した Metadata:etcd プラグインを新たに実装した．具体的に格納するデータは

- Content
- Snapshot (Committed)
- Image
- Lease

に関するメタデータである．対して非共有部のメタデータ管理には通常通り bbolt を用いる．具体的に非共有部には

- Container
- Snapshot (Active,View)
- Lease
- Sandbox

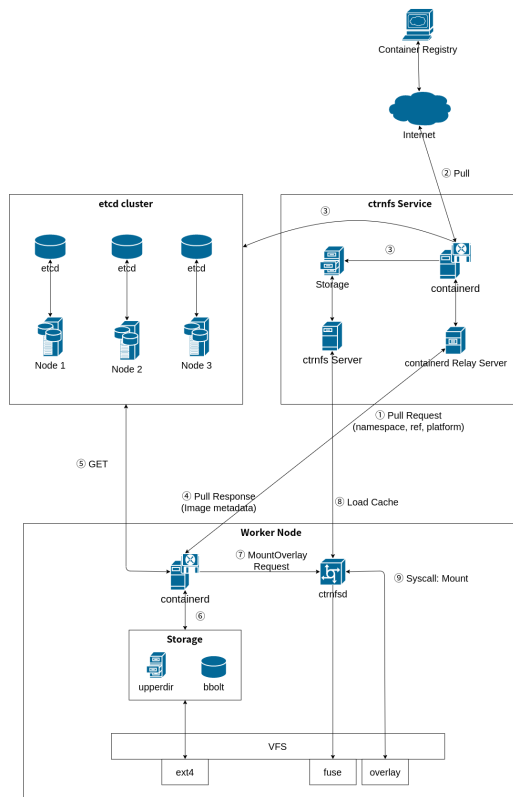


図 2: 全体のアーキテクチャ

が該当する．

containerd では Metadata プラグインが全てのメタデータを管理しているわけではない．Snapshot に関する詳細は Snapshot プラグインが管理している．ここでも共有部のメタデータの保存には etcd を，非共有部の保存には bbolt を用いることにする．より具体的には，Committed Snapshot のメタデータは etcd に保存され，Active または View Snapshot のメタデータは bbolt に保存されることになる．このように保存先の KVS を分岐させるために，Snapshot を識別する key の接頭辞に “upperdir” もしくは “lowerdir” を付与しておき，この接頭辞に応じて呼び出す処理を分岐させる．接頭辞が “upperdir” であれば Snapshot の種類は Active または View であり，接頭辞が “lowerdir” であれば種類が Committed であることを示唆している．

5.3 containerd-relay-server

containerd-relay-server はコンテナイメージに関する操作を行うためのシンプルなリレーサーバーであり，grpc を通信のプロトコルとして採用した．containerd-relay-server が動いているノード上には通常の containerd が動いており，リレーサーバーに送られてきたリクエストを処理している．

5.4 ctrnfs

ctrnfs は grpc を通信に用いた FUSE ファイルシステムである．grpc を経由して遠隔のノード上でシステムコールを呼び出すことで，遠隔のファイルシステムに対して操作を行うことができる．ctrnfs は一度取得した dentry をメモリ上にキャッシュしておくよう動作するが，このキャッシュを ToC としてストレージ上に保存することができる．また保存した ToC をキャッシュとして読み込むこともできる．

ctrnfs のクライアント上では ctrnfsd と呼ばれるデーモンが常に動作しており，Mount リクエストまたは MountOverlay リクエストを処理する．Mount リクエストは ctrnfs のマウントを要求するリクエストである．これに対して MountOverlay リクエストは OverlayFS のマウントを要求するリクエストであるが，リクエストを受けるとマウントの前後でキャッシュの読み込みまたは解放を行うラッパープログラムとして次のように動作する．

1. オプションで指定された ToC のパスを基に ToC を読み込む．
2. OverlayFS マウントを行う．
3. Linux の inotify API[10] を用いて，上で OverlayFS マウントされたファイルシステムのアンマウントを検知し，キャッシュの解放を行う．

表 1: 物理マシンのスペック

CPU	Intel Xeon E5-2670
メモリ	DDR3 256GB
ネットワーク	10GBASE-T
ストレージ	900GB SAS HDD

6. 評価

実験環境は，表 1 の物理マシン 3 台で構築された oVirt-4.4 の環境で複数台の仮想マシンを動かすことによって実験を行った．仮想マシンは，virtio ドライバを用いて，表 2 の通りのスペックのものを用意した．

ここではリモートファイルシステムとして, Linux カーネルに搭載される NFS を採用する．NFS サーバーを動かすノードと同一のノード上で, ctrnfs Server および containerd-relay-server も動いている．また etcd を動かすにあたってストレージのアクセス速度に問題があったため，データ領域を tmpfs 上に配置することにより実験を行った．

6.1 コンテナイメージのプル時間に関する評価

コンテナイメージのプル関して以下の 3 通りの方法を採用したとき，それぞれの方法についてかかる時間を計測した．プル対象のイメージには docker.io/library/nginx:1.23.1 を採用した．

- ローカルストレージ上にプルする場合

ⓒ2023 Information Processing Society of Japan

表 2: 仮想マシンの構成

NFS 構成	1 台
NFS 用ノードのスペック	64vCPU,8GB RAM
NFS 用ノードの OS	Ubuntu 22.04 LTS
etcd クラスタ構成	3 台
etcd ノードのスペック	1vCPU,4GB RAM
etcd ノードの OS	Ubuntu 22.04 LTS

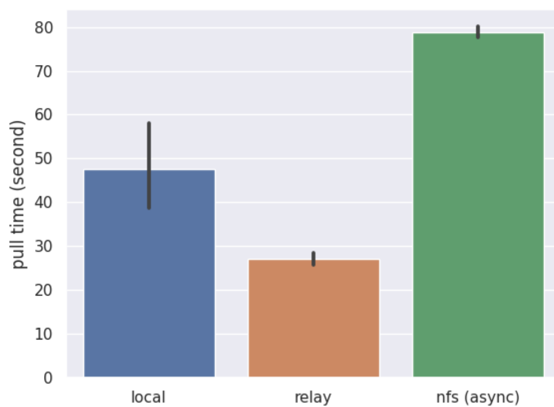


図 3: プルにかかる時間（N=10，95percentile）

表 3: kubernetes クラスタでの実験環境

kubernetes のバージョン	1.25.3
CNI	flannel
構成	master1 台＋ worker4 台
各ノードのスペック	64vCPU, 8GB RAM
各ノードの OS	Ubuntu 22.04 LTS

- NFS(async オプションによる非同期書き込み) 上にプルする場合
- containerd-relay-server 経由でプルする場合

測定結果は図 3 に示す通りであった．containerd-relay-server を経由することで，NFS 上にプルした場合よりも半分以上短い時間でプルが完了されることがわかった．

6.2 コンテナベースのクラスタを用いた性能評価

コンテナオーケストレーションツールの一つである kubernetes を用いてクラスタを構築した上で，Pod の数を 10,20,30,40,50 と変化させたとき

- 全ての Pod の状態が Running になるまでの時間（アップタイム）
- コンテナイメージが占有するストレージの使用量

を計測した．実験環境の詳細は表 3 に示す．

コンテナイメージの利用方法として，従来通り各ノードのローカルストレージ上にプルしてくる場合と，本論文で提案するアーキテクチャを用いて，NFS 上のコンテナイメージを共有する場合の 2 通りを採用した．実験結果は図

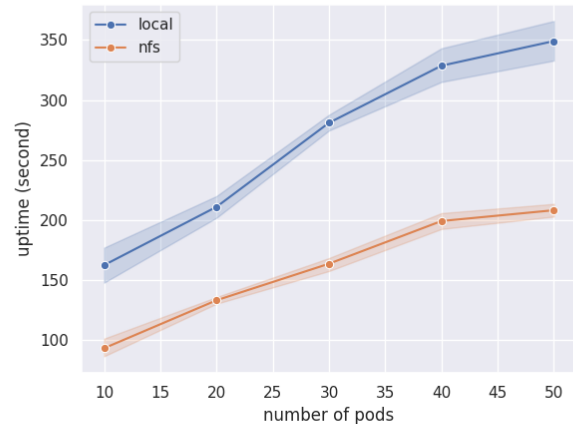


図 4: Pod の数を変化させたときのアップタイム（N=10，95percentile）

表 4: コンテナイメージが占有するストレージ使用量（MB）

	ローカル	NFS
master	724	
worker0	282	
worker1	282	
worker2	282	
worker3	282	
合計	1852	897

4 の通りであった．NFS によってコンテナイメージを共有することで，全ての場合においてアップタイムを縮めることができた．

またコンテナイメージが占有するストレージ使用量は表 4 の通りであった．NFS を用いた場合はファイルシステムが共有されているため，合計以外の欄は空白になっている．こちらも NFS によってコンテナイメージを共有することで，ストレージ使用量を $1852 - 897 = 955$MB 節約することができた．

6.3 地理的に離れた環境での性能評価

レイテンシを 0ms から 50ms の間で付与した時，

- コンテナの作成にかかる時間
- コンテナ内で指定されたタスクが開始されてから完了するまでの時間

を計測した．コンテナイメージには各ジャンル WEB,DB,Language,Distro の中から，代表となるものを選んで採用した．コンテナイメージの保存場所には，ローカルストレージ，NFS，ctrnfs の 3 つを採用した．

計測は containerd が出力するログを元に，Snapshot の stat リクエストから/tasks/start リクエストが送られるまでの時間を，コンテナ作成にかかる時間すなわち「作成時間」として計算した．同様に/tasks/start リクエストか

ら/tasks/exit リクエストが送られるまでの時間を「実行時間」として計算した．また作成時間と実行時間を足し合わせた合計時間を計算し，これら 3 つをそれぞれグラフとして描画した．

まず WEB コンテナに関する計測結果を図 5 に示す．代表イメージとして docker.io/library/nginx:1.23.1 を採用し，コンテナ内で実行するタスクには “nginx” コマンドを用いた．

次に DB コンテナに関する測定結果を図 6 に示す．代表イメージには docker.io/library/mysql:8.0.29 を利用し，タスクには “mysqld –daemonize=True” コマンドを用いた．

次に Language コンテナに関する測定結果を図 7 に示す．代表イメージには docker.io/library/python:3.11.0-slim を利用し，タスクには “python -c ’print(2**32-1)’ ” コマンドを用いた．

最後に Distro コンテナに関する測定結果を図 8 に示す．代表イメージには docker.io/library/ubuntu:22.04 を利用し，タスクには “echo helloworld” コマンドを用いた．

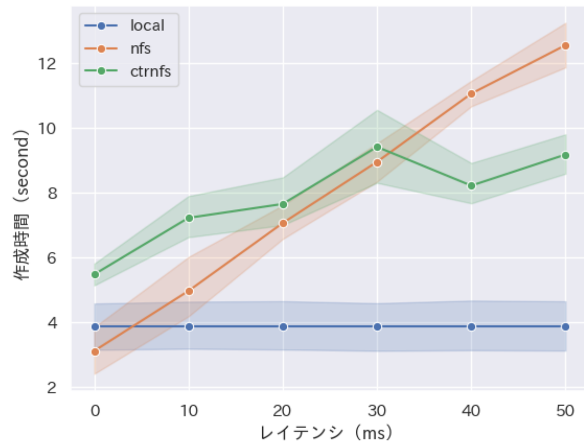
7. 考察

6.1 節にて行ったプル時間の比較では，containerd-relay-server を実装することでプル時間を大きく改善できたことがわかった．またコンテナイメージを共有して利用できるようになったことで，6.2 節にて評価した kubernetes クラスタを組んだときの Pod のアップタイムやストレージ使用量についても，大きく改善することができた．

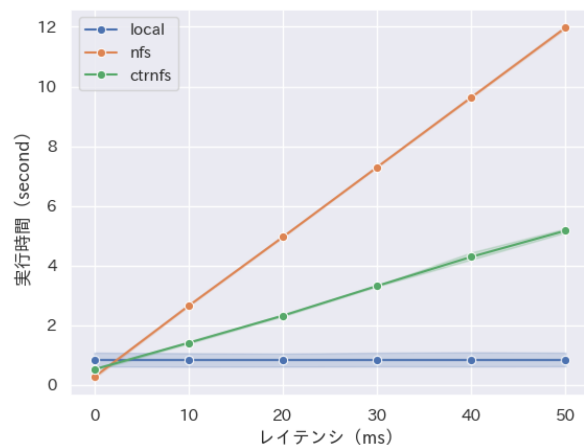
6.3 節の地理的に離れた環境での性能評価について図 5 から図 8 を見ながら考察していく．まず各図のコンテナの作成時間に関するグラフについて見ていく．ctrnfs を利用した際，遅延の低い環境ではかえって NFS よりもコンテナの作成に時間がかかってしまっている．ctrnfs ではマウント前に ToC を読み込むオーバーヘッドが発生する．また低遅延環境下ではサーバーに対する 1 つ 1 つのリクエストに時間がかからないため，ToC により読み込んだキャッシュから dentry を取得した場合と，サーバーに問い合わせて dentry を取得した場合で，かかる時間にそれほど差がないと考えられる．これらの要因により，ToC によるキャッシュが有効に働いていないのだと思われる．

続いてコンテナの実行時間について見ていく．こちらはほとんどの場合において ctrnfs の方が NFS よりも短い時間で済んでいる．実行時間はコンテナ内でプロセスが稼働している時間であり，リモートファイルシステムに対して最も多くのリクエストが送られる期間である．そのためここでは ToC によるキャッシュが有効に働いており，サーバーに対するリクエストを抑えることで，遅延が高くなればなるほど NFS との差が大きくなっているのがわかる．

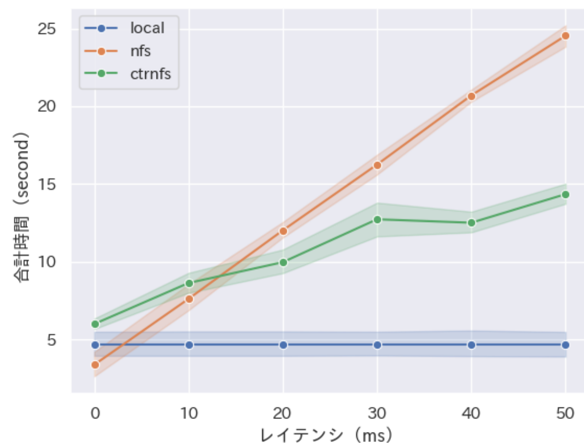
以上の考察により，ctrnfs はコンテナ作成に多少の時間がかかるものの，実行時間は短いという性質が判明した．これらの合計時間において ctrnfs が NFS よりも短くなるためには，実行時間における NFS との絶対的な差が一定



(a) 作成時間

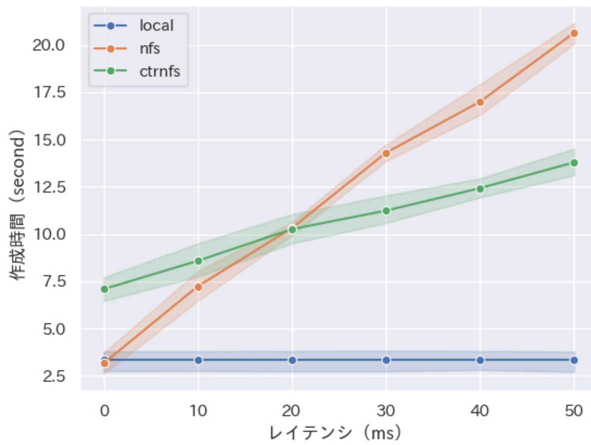


(b) 実行時間

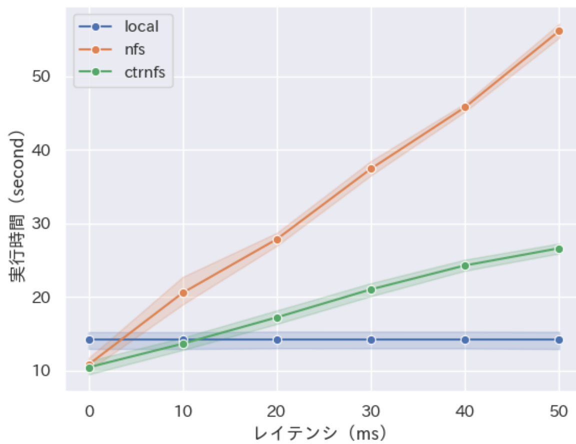


(c) 合計時間

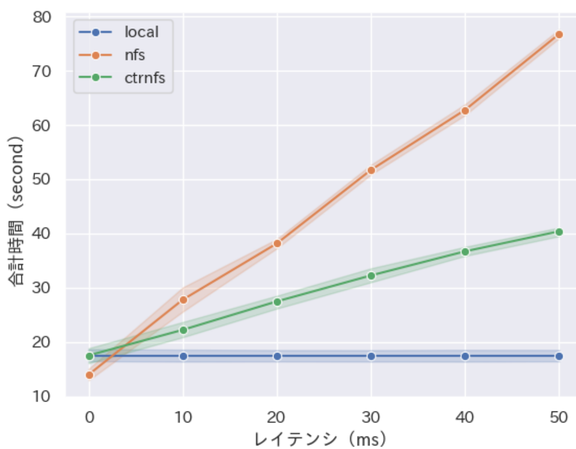
図 5: WEB コンテナの測定結果（N=10，95percentile）



(a) 作成時間

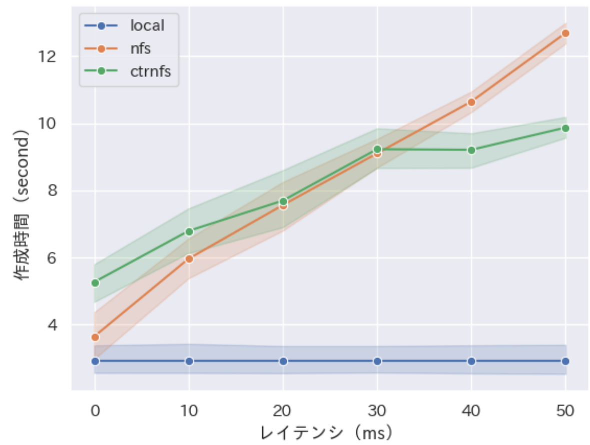


(b) 実行時間

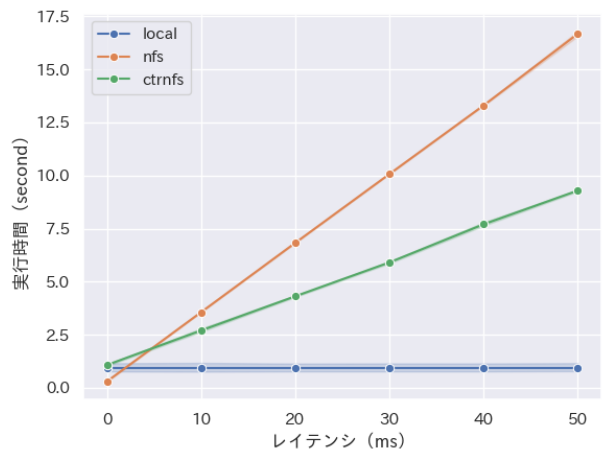


(c) 合計時間

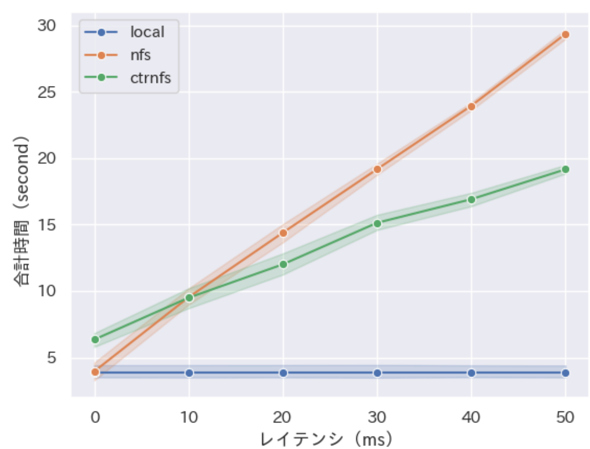
図 6: DB コンテナの測定結果（N=10，95percentile）



(a) 作成時間



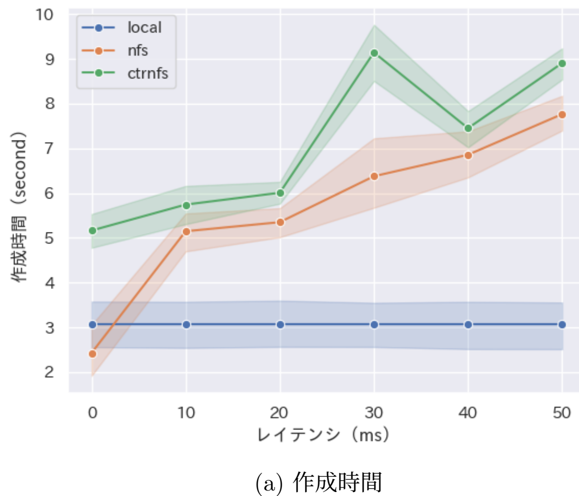
(b) 実行時間



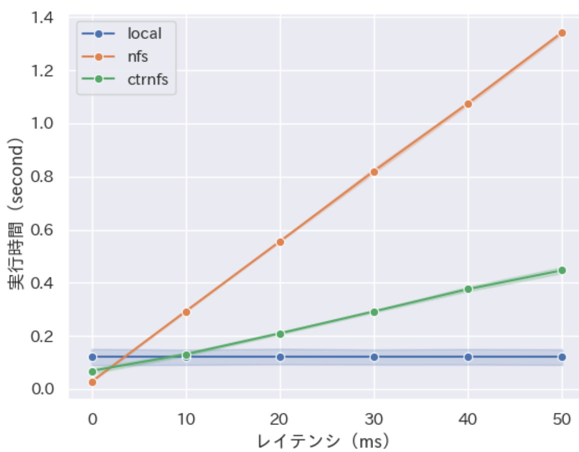
(c) 合計時間

図 7: Language コンテナの測定結果（N=10，95percentile）

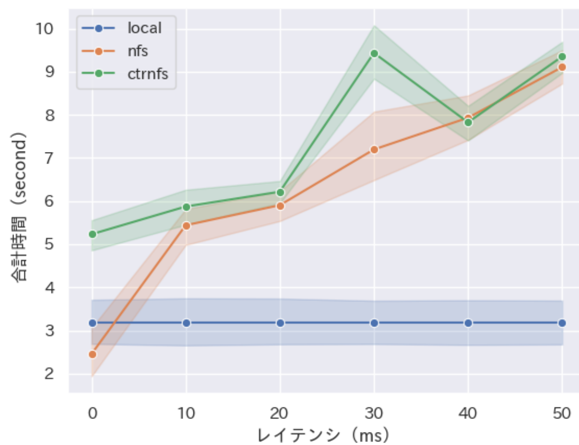
以上必要である．そして実行時間における NFS との絶対的な差は，コンテナ内のプロセスがリモートファイルシステムに対してどれほどのリクエストを送ったかに起因すると考えられる．例えば DB コンテナのような大きなプログラムを実行した場合には，リクエストが多く NFS との差が大きい（図 6：(c)）．対して Distro コンテナ内で echo



(a) 作成時間



(b) 実行時間



(c) 合計時間

図 8: Distro コンテナの測定結果（N=10，95percentile）

のような小さなプログラムを実行した場合，絶対的には差は小さく，合計時間は常に NFS の方が短い（図 8：(c)）．

8. まとめ

本研究では，主に共有リモートファイルシステムと分散型 key-value ストアを用いて，ネットワーク経由でコンテナイメージを共有して利用できるコンテナイメージ提供システムの試作を行った．これにより 1 度プルが完了したコンテナイメージであれば，全てのノードがプルの過程を省いてコンテナを実行できるようになった．またネットワーク環境のうち，リモートファイルシステムとの間の遅延が性能に与える影響を評価した．予め作成した ToC をコンテナ作成前に読み込むことで，リモートファイルシステムが行う通信の回数を減らすことができ，特に高遅延環境下においては遅延の影響を効果的に減らすことができることがわかった．

今後の課題として，プルそれ自体にかかる時間を減らすような仕組みが求められる．また本システムの応用として，円滑なコンテナのライブマイグレーションが可能なシステムの検討及び実装などが挙げられる．

参考文献

[1] : Slacker: Fast Distribution with Lazy Docker Containers — USENIX, https://www.usenix.org/conference/fast16/technical-sessions/presentation/harter.

[2] : Overlay Filesystem — The Linux Kernel documentation, https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html.

[3] : containerd/containerd: An open and reliable container runtime, https://github.com/containerd/containerd.

[4] : google/crfs: CRFS: Container Registry Filesystem, https://github.com/google/crfs.

[5] : containerd/stargz-snapshotter: Fast container image distribution plugin with lazy pulling, https://github.com/containerd/stargz-snapshotter.

[6] Chen, J. L., Liaqat, D., Gabel, M. and de Lara, E.: Starlight: Fast Container Provisioning on the Edge and over the WAN, *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, Renton, WA, USENIX Association, (online), available from ⟨https://www.usenix.org/conference/nsdi22/presentation/chen-jun-lin⟩ (2022).

[7] : etcd-io/etcd: Distributed reliable key-value store for the most critical data of a distributed system, https://github.com/etcd-io/etcd.

[8] : etcd-io/bbolt: An embedded key/value database for Go., https://github.com/etcd-io/bbolt.

[9] : Pathname lookup — The Linux Kernel documentation, https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html.

[10] : inotify(7) - Linux manual page, https://man7.org/linux/man-pages/man7/inotify.7.html.