大規模分散処理システムのシステム検証における テスト環境の開発アクティビティの実践

梅田 昌義† 吉田 悟†

†NTT ソフトウェアイノベーションセンタ 第二推進プロジェクト 〒180-8585 東京都武蔵野市緑町 3-9-11

E-mail: †umeda.masayoshi@lab.ntt.co.jp, †yoshida.satoru@lab.ntt.co.jp

あらまし 本稿では，筆者らが携わったプロジェクトで開発を進めた大規模分散処理システムである CBoC タイプ 2 のシステム検証を実施して得られた知見を元に，商用の大規模分散処理システムのシステム検証における課題を明確化し，どのような技術と対応が必要になるかを説明する．特に「テスト環境の開発」アクティビティにおける問題を「読み書きのデータ特性や集計を集中制御可能な TP 作成」，「ツールによる検証作業の効率化」，「NW（ネットワーク）／Disk（ディスク）環境のばらつきの回避」，「故障検知と故障対応」の技術的課題として整理し，それぞれに対して具体的な解決方法を提案した．また，これらを具体的な実例に適用した．結果，システム検証の遅延リスクを減少させた効率的な検証実施が可能となり，提案方法の有効性を確認することができた．

キーワード 大規模分散処理システム，システム検証，テスト環境，ビッグデータ

1. はじめに

近年，Google 検索や Amazon ネットショッピングでのレコメンドなど，ビジネスにおける BigData の有効活用が進んできており[1]，分散並列処理技術の発達により，BigData の管理と分析処理が可能で，スケールアウトが容易な大規模分散処理システムの重要性が増してきている．大規模分散処理システムを実現する代表的なソフトウェアとしては，Apache プロジェクトの Hadoop[2]があるが，筆者らが実際に携わったプロジェクトでは，NoSQL[3]による大量の非構造化データを対象としたバッチ処理系の大規模分散処理システムとして，CBoC（Common IT Base over Cloud Computing）タイプ 2[4]の開発を実施した．CBoC タイプ 2 は，「分散ロック」，「分散ファイル」，「分散テーブル」のプロダクトから構成され，Web コンテンツや多種多様なログデータ等の加工・分析を目的として数百台の PC サーバで連続に安定して大規模データを分散して蓄積・処理するソフトウェアプロダクトである．CBoC タイプ 2 は，検索サービスのバックエンドシステムとして数百台規模の Linux マシンで商用運用されている．具体的には， 図 **1** に示すようにクローラが Web サイトのデータを収集して CBoC タイプ 2 に書き込まれ，そのデータがデータ分析処理の際に読み出されて検索エンジンに必要なランキングのスコア等が作成される．このような商用システムのソフトウェア開発においては，商用運用前の最終確認となるシステム全体の検証（以降，システム検証と呼ぶ）が，事前に問題を発見・改修して商用では発生させないようにするという意味において非常に重要である．一方で，期間やリソースに制約のある商用の大規模分散処理システムに対するシステム検証では，検証の効率化を十分に図る必要があるという課題が発生する．

本稿では，商用の大規模分散処理システムに対するシステム検証における課題を抽出・整理し，具体的な解決策を導き出す．第 2 章では，大規模分散処理システムの関連研究を紹介する．第 3 章では，大規模分散処理システムに対するシステム検証の説明を行い，その課題を明確化する．第 4 章では，課題に対して実際のシステム検証において有効性を確認した実践的な解決方法を提案する．

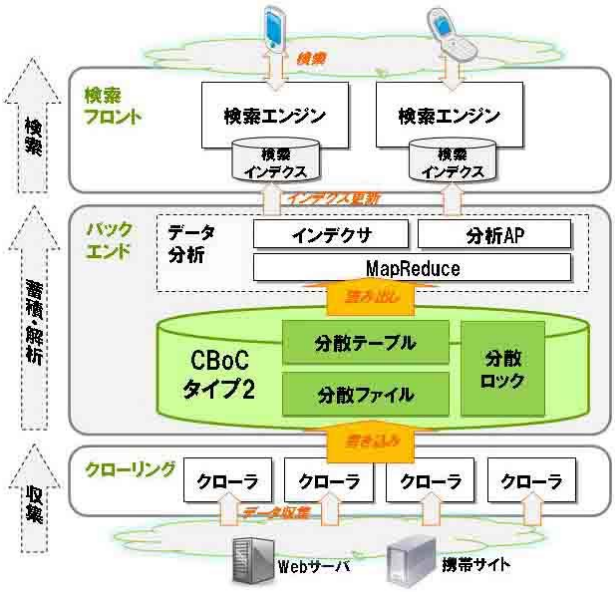

図 1. 検索サービスでの CBoC タイプ 2 適用イメージ

2. 関連研究

大規模分散処理システムに対するシステム検証効率化の課題に対して，既存の研究を下記(1)～(3)の観点で調査・分析した結果を示す．

(1) 大規模分散処理システムのシステム検証に関する

研究

大規模分散処理システムのシステム検証は，通常のマシン数台により構成されるシステム（以降，一般的な情報システムと呼ぶ）の検証とは異なり，数百台規模のマシンをリソースとして管理し検証を実施する必要がある．しかし，大規模分散処理システム自体のアーキテクチャが新しいことから，大規模分散処理システム固有の問題を考慮した検証の効率化に関する公知の文献は，これまでほとんどなかった．既存の論文[5][6][7]は，システム検証における性能の問題と項目作成の問題，あるいはクラウドシステムの検証におけるレガシーシステムの検証と比較した際の有効性，を扱うにとどまっている．また，Google のシステムに関する検証[8]や Microsoft のシステムに関する検証[9]は，ログの出力や API を利用したトラッキングやモニタリングによる動作の確認やボトルネックの確認方法であり，システム固有の確認方法であるため，他の類似した大規模分散処理システムでは確認ができない．

(2) 分散処理システムの検証に関する研究

分散処理システムの検証に関しては，論文[10][11]があるが，数百台規模のマシンを利用した検証効率化の観点はない．

(3) 一般的な情報システムの検証に関する研究

システム検証を一般的な情報システムのソフトウェア開発の検証については文献[12][13][14][15]が詳しい．しかし，検証の実施に際しては，検証の課題やその課題の対処方法の具体的な記述はない．例えば，今回の大規模分散処理システムに特有な，マシン台数も多く，プロダクトも複数あることから発生する検証実施に時間を要する問題や，故障検知の問題等に適用することができない．

これらの既存研究に対して，本稿では，大規模分散処理システムに対するシステム検証の課題について，数百台規模のマシンを利用した検証効率化の観点からの解決策を示す．

3. 大規模分散処理システムのシステム検証と課題

3.1. システム検証とアクティビティ

システム検証の目的は，開発工程の V 字モデルにおける最後の工程として，システム全体の検証を行い，要求仕様を満たしていることを確認することである．本稿では，検証における工程の分類をアクティビティの単位として説明する．具体的には，表 1 に示す IEEE で作成されたソフトウェアエンジニアリングの基礎知識体系の SWEBOK[16]に記述されている分類に基づいて説明を行う．

本分類における「テスト環境の開発」は，検証の準備に関してであり，効率的に実施可能な検証の環境を整える工程全体に関わる作業であるため，本論文では，「テスト環境の開発」のアクティビティに着目する．

表 1. アクティビティの分類

アクティビティ	内容
計画	人員調整，利用可能なテスト施設，装置に対するマネジメントおよび予想される不良な実行効果の対策
テストケース生成	実施されるテストレベルおよびテスト技法
テスト環境の開発	ソフトエンジニアリングツールとの互換性があり，制御が容易で結果を記録，再現可能なテスト環境
実行	文書化された手続きに沿って，テストの実施
テスト結果の評価	テストが成功したか，テスティングの結果の評価
問題報告/テストログ	テストログより，いつテストが実施されたか，だれが実施したか等関連する識別情報を残し，予期しない不正な結果は問題報告システムに記録し文書化
欠陥追跡	欠陥に対する分析を行い，原因等の決定

3.2. システム検証の課題

大規模分散処理システムのテスト環境は，マシン台数が多く検証対象のプロダクトが複数あるため，効率よくかつコントロールが可能なように，準備する必要がある．CBoC タイプ 2 のシステム検証における「テスト環境の開発」アクティビティでも，**表 2** に概要を示すような検証期間の短縮や検証作業の効率化が必要となる問題が実際に発生した．それぞれ，本アクティビティに関する数週間に及ぶスケジュールの遅延を発生したものであり，以下に詳細を説明する．なお，その他の問題については，数日のスケジュール遅延を発生させるものではなかった．

1 点目は，大規模分散処理システムの検証では，検証をする際に書き込みや読み出しの処理を擬似する必要があり，そのツールの準備が必要なことである．例えば，データの読み書きには，シーケンシャルリード，ランダムリード，ランダムライトと種類があり，さらにデータ自体もテキストや画像等のマルチメディアがあるため，これらに対する処理を実現すると共に，性能の基礎データを取得するに読み書きのスループットやデータ量の調整を細かく制御可能とするツールが必要である．

2 点目は，大規模分散処理システムにおいて，検証

の実施は，対象となるマシンが多いため操作が複雑となり，1 項目の実施に時間がかかることから，ツールを用いて効率的に実施することが必要なことである．例えば，検証は数百台のマシンを利用して検証項目を実施するため，1 台ずつを個別に操作して検証項目を実施すると 1 台 30 秒でも 100 台で 50 分かかってしまう．そのため，短時間で検証項目を実施できるように数百台のマシンを全てコントロールする必要がある．

3 点目は，大規模分散処理システムの検証では，環境を組み替える等して複数環境での効率的な実施が必須であることから，各環境に対する基本性能の確認が必要となることである．例えば，大規模分散処理システムは，ラックにマシンを設置し，スイッチを介して複数のラックを組み合わせてシステムを構成する．使われているマシンは全て同スペックであるため性能は均一と考えていたが，実際には性能値はばらつきがあった．このことを認識していないと，検証結果が想定と異なっていた場合に，結果の正しさが確認できなくなることがある．また，問題が発生した場合，プロダクトの解析だけでなく環境起因との切り分けが必要となることがある．

4 点目は，利用するマシン台数が多く故障が日々発生するため，検証への影響回避が必要となることである．例えば，故障したマシンでプロダクトを稼動させたままにすると，システム全体の性能や動作に影響が出る可能性があるため，マシンの切り離しをするなど検証への影響回避が必要である．

これらの実際に発生した問題から抽出した技術的課題を表 2 に示す．

表 2.「テスト環境の開発」アクティビティで発生した問題の概要と技術的課題

作業分類	発生した問題の概要	技術的課題（具体的内容）
検証実施	検証で利用する読み書きの処理が必要	読み書きのデータ特性や集計を集中制御可能な TP 作成（読み書きデータのスループットやデータ量の調整を細かく制御が可能な TP の作成し性能の基礎データを取得する）
	手作業の検証実施では検証期間をオーバ	ツールによる検証作業の効率化（OSS やスクリプトのツールを活用し検証を効率的に実施する）
環境確認	環境毎に測定結果が想定と異なる	NW/Disk 環境のばらつきの回避（複数環境において，各環境の基本性能を確認し検証結果の判断に利用する）
	マシン故障により機能確認や性能測定の結果が想定と異なる結果となる	故障検知と故障対応（マシン故障を速やかに検知し，検証に影響が出ないようにマシンの切り離しなどの対策をとる）

4. 検証作業の効率化

本章では，限られた期間内にリソースの制約条件の下に，システム検証を完了するための方法と適用事例を，3 章で抽出した 4 つの技術的課題毎に説明する．

4.1. 読み書きのデータ特性や集計を集中制御可能な TP 作成

大規模分散処理システムのシステム検証では，機能検証としてユースケースにおける書き込みや読み込みを模擬した処理が必要である．また，非機能検証として性能測定を実施する際には，測定用の読み込み/書き込みの負荷データを発生させるとともに，表 3 に示すような性能評価ができるログの出力を行うベンチマークツールや実アプリケーション（AP(Application Program)）を用意する必要がある．

1 点目の汎用ベンチマークツールは，OSS のベンチマークが主流であり，特定の条件の書き込みや読み出しの基本性能の把握が可能である．また，OSS 公開されているものは，利用者が多く，公開されている類似システムの結果と比較することが可能である．しかし，その結果は，特定条件の書き込みや読み出しの基本性能値しか取得できないため，実際のデータ特性を模擬した性能や，過負荷時や運用によるマシンメンテナンス作業など，商用での実利用を想定した性能を測定することができない．さらに，汎用ベンチマークツールは，商用で運用される AP の処理を模擬するために必要となる障害時の処理を機能として具備していない．

2 点目の独自ベンチマークツールは，商用で運用される AP の処理を模擬した TP(Test Program)であり，実利用を想定した性能用件や限界性能の特性を把握することが可能である．しかし，新たに TP を作成する必要があり，開発リソースが必要となってしまう．

3 点目の実アプリケーション(AP)は，商用で運用されるアプリケーションそのものであり，商用のユースケースを忠実に再現でき，機能や性能を正確に確認が可能である．しかし，利用に際してインターネット上のデータを取得するには，グローバルな IP アドレスを利用したインターネット環境が必要となるなどの制約がある場合がある．また，例えば書き込みデータ量が対象となるサーバ毎に異なるなど，検証に適したデー

タ量がコントロールできない場合がある．

一般的な情報システムのシステム検証では，主に汎用ベンチマークツールと実アプリケーションを利用することが多い．しかし，大規模分散処理システムのシステム検証においては，汎用ベンチマークツールでは商用の実利用を想定した性能が取得できないこと，実アプリケーションでは環境の制約やデータ量のコントロールができないことから，これらを解決する独自ベンチマークツールの作成が必要となる．本節では，商用のシステム検証で必要なベンチマークを明らかにするため，独自ベンチマークツールとして CBoC タイプ 2 の性能測定兼システム検証用に独自開発したツール（ランダムライト，ランダムリード，シーケンシャルリード）の機能について説明する．

表 3. 性能測定や負荷発生のためのツール

負荷発生方法	説明	主な用途
汎用ベンチマークツール	OSS 等のベンチマークツールで目的に合わせてツールを取捨選択（TestDFSIO[1]，TeraSort[2], MRBench[3], YCSB[4], TPC-H－Hive[5]など）	・類似システムとの性能比較
独自ベンチマークツール	実利用を模擬した負荷発生用のテストプログラム（TP）を独自に開発	・ユースケース性能要件達成確認 ・限界性能特性の把握
実アプリケーション	商用で運用される実アプリケーション（AP）そのものを用いる	・ユースケース性能要件達成確認

[1] Hadoop に付属の分散ファイルシステムベンチマークツール[17]

[2] Hadoop に付属の巨大ファイルソートプログラム[17],http://hadoop.apache.org/docs/current/api/org/apache/hadoop/examples/terasort/package-summary.html

[3] Hadoop に付属の MapReduce ベンチマークツール[17]

[4] Yahoo が公開している NoSQL 用ベンチマークツール[18]

[5] データウェアハウス向けベンチマークである TPC-H を Hive 上で動作可能としたツール，https://github.com/rxin/TPC-H-Hive/tree/master/TPC-H_on_Hive

4.1.1. 課題解決の考え方

独自ベンチマークツール(TP)は，AP の処理を模擬し性能を測定可能とするため，以下の機能を具備する．

(1) 作業効率化のために複数台の TP を集中制御可能であること

(2) 読み込み／書き込みのデータ内容や量をパラメータ等で柔軟に変更可能であること

(3) 実際の読み込み／書き込みに関する情報を計測できる仕組みがあること

1 点目は， TP 数が数百に及ぶ場合には一つ一つ手作業で起動や管理することは時間がかかるため，コントロールサーバの TP コントローラを起動することにより，各 TP を一括で管理でき，設定変更もコントロールサーバで実施できる．また，AP の処理と同じように，書き込みや読み出しの TP の起動や終了タイミングを合わせることで，AP の模擬が可能となる．

2 点目は，データの内容を固定ではなく変更可能とすることにより，データの内容によるシステムの性能への影響を確認できる．また，AP のデータ内容を模擬することで，実際のシステムを模擬した機能確認が可能となる．

3 点目は，各 TP のログ等から結果を集計することを，仕組みとして持つことにより，数百台規模のマシンから時刻を合わせて集計した結果を容易に確認することができる．

この考え方は，TP の作成というアプローチであり，一般の情報システムの場合でも同じである．そして，TP による読み書きがある場合，他の大規模分散処理システムでも適用可能である．一般の情報システムの場合，TP の作成は基本性能の確認が主であり，通常システム検証においては，AP を利用する．しかし，大規模分散処理システムのシステム検証では，AP の利用より，独自ベンチマークツールが有効であり，かつ上記 3 点で充分であることは，公知の文章では示されていない．

4.1.2. 具体的な解決方法

具体的な対応としては，ランダムライト／ランダムリードとシーケンシャルリードの独自ツールを TP(テストプログラム)として作成し，集中制御を可能とする．ランダムライト／ランダムリードについては，1 台のコントロールサーバと複数台のランダムライト処理をするサーバで構成される．オペレーションは全てコントロールサーバで行い，コントロールサーバから実処理を行う各サーバに対して起動／停止／性能データ収集等の制御を行う．また，登録するデータは，表 4 のようにツールにより自動生成する．読み込み／書き込みのデータにおいて，検索されるキーとなるデータの RowKey は，複数のサーバ／プロセス／スレッドから

読み込み／書き込みを実施することから，該当のサーバやプロセスの情報を用いて，ハッシュによる変換処理を用いて算出する．また，読み込み／書き込みをするデータ自体は，Rowkey で使用した該当のサーバやプロセスの情報からハッシュ算出するとともに，データ登録時の圧縮制御を加味し生成している．これにより，検証に利用するデータの特性を把握し管理可能とし，書き込みをするデータと書き込み後のデータの整合性をチェックすることを可能としている．シーケンシャルリードについては，ランダムライト/リードのツールと同様に，1 台のコントロールサーバと複数台のシーケンシャルリード処理をするサーバで構成する．オペレーションは全てコントロールサーバで行い，コントロールサーバから実処理を行う各サーバに対して起動／停止／性能データ収集等の制御を行う．

また，読み込み／書き込みのデータ生成 TP は，発生させる読み込み／書き込みのデータ量に応じてサーバ台数が増える．そのため，各サーバでのデータ生成 TP の起動／停止や性能結果の収集については，集中制御が可能なように構築することにより，作業の効率化を図る．以下に検証に利用するデータの特性を管理するために工夫した点と，計測の仕組みで工夫した点を示す．

(1)読み込み／書き込みのデータの生成におけるパラメータは，データの生成バリエーションを増やすため以下の工夫を実施した．
- ① スレッド数／プロセス数やデータの送信間隔で読み込み／書き込みデータの発生量を調整可能
- ② 登録するデータサイズは一様ではなく正規分布や F 分布やポアソン分布のようにデータサイズを分布に基づいて登録することを可能
- ③ 登録を中断した場合に中断点からの再開（レジューム）を可能

(2)データの読み込み／書き込み性能を計測する仕組みについては，読み込み／書き込みによる結果等の確認のためのログ出力をすることとし，下記に示す工夫を実施した．
- ① 各処理の平均処理時間のみならず，最大／最小／分散等を出力
- ② レスポンスタイムがあるしきい値以上のデータの個数を出力

表 4. ランダムライト/ランダムリードにおけるデータ生成方式

生成項目	内容
RowKey	実処理を行うサーバの IP アドレス、シーケンス番号，プロセス ID，スレッド ID からハッシュ算出
データ（テキスト）	実処理を行うサーバの IP アドレス、シーケンス番号，プロセス ID，スレッド ID、世代番号（パラメータ指定）からハッシュ算出 データ登録時の圧縮機能による圧縮率になるようにデータを作成．また，圧縮率はパラメータとして指定可能．
データ（画像）	実処理を行うサーバの IP アドレス、シーケンス番号，プロセス ID，スレッド ID，世代番号（パラメータ指定）からハッシュ算出

注：世代：データの版数

4.1.3. 適用結果

この方法を適用した具体的な実例と結果について示す．大規模な TP を効率よく実行するために，TP のコントロールサーバからの一括起動や設定変更等の一元管理をした．以下に具体的な TP の利用例を示す．

(1)図 **2** のように構成することにより，TP の集中制御を可能とした．

(2)データの読み込み／書き込みのパラメータは，ランダムライト／リード，シーケンシャルリードそれぞれ表 5，表 **6** のように指定可能とした．

(3)データの読み込み／書き込みした性能を計測する仕組みについては，以下の表 7，表 **8** のようなログ出力を可能とした．

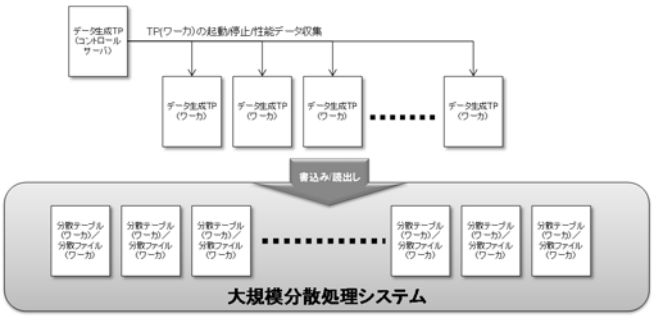

図 2. TP の構成と集中制御

表 5. 性能測定ツール（ランダムライト／リード）で指定可能な項目

指定可能項目	備考
読み込み/書き込みのデータ（スレッド数、インターバル、総件数）	総ライト件数は 1 ブロックあたりの件数×繰り返し回数で指定

RowKey サイズ	
データサイズ分布	正規分布／F 分布／ポアソン分布／特定ユースケースの分布(実ユースケースで取得したデータより分布を作ったもの)
データサイズ	データサイズの平均値
レジューム	指定した件数以降のデータ投入から開始／途中で終了した場合に処理中のブロックから開始
スループット計測周期	

表 6. 性能測定ツール（シーケンシャルリード）で指定可能な項目

指定可能事項
読み出しカラム
検索対象となるデータの timestamp
同時読出しプロセス数
読出しパターン（特定世代のみ，全世代）
フィルタリング指定

注：世代：データの版数

表 7. 性能測定ツール（ランダムライト/リード）の主な出力ログ

ログ出力項目	備考
ログ出力時刻	
アクセス数	前回ログ出力時刻から今回のログ出力時間までに性能測定ツールのプロセスが処理を行った数
平均処理時間	
処理時間の分散	
処理時間の最大値	
処理時間の最小値	
スループット	
処理時間が閾値以上の件数	閾値は指定する
読み出しデータ無しの件数	Rowkey に対してカラムに対するデータが１つも無い件数
読み出しデータの不一致の件数	RowKey に対するデータが想定データと違っていた件数

表 8. 性能測定ツール（シーケンシャルリード）の主な出力ログ

ログ出力項目
読出し開始/終了時刻
読出しデータ量
読出し件数
読出しスループット
読出し Area（水平分割されたテーブルの一部）数

以上の TP 作成により，TP での機能検証と非機能検証が可能となった．特に AP の利用による環境制約の考慮やデータ量コントロールの工夫が必要なくシステム検証が実施可能となった．

4.2. ツールによる検証作業の効率化

大規模分散処理システムでは，検証対象のマシンの数が数百台規模あり，手作業の構築では 1 週間はかかってしまう．また，検証実施において結果確認は，数百台規模のログ確認に時間を要する．このため，検証環境構築と検証実施／結果確認の効率化が必要となる．

一般的な情報システムの Web システム等では，検証実施は，自動検証が可能[19]であり，検証結果の確認も自動で可能である．しかし，大規模分散処理システムのシステム検証では，Hadoop のツールとして環境構築をサポートするツール[20]はあるが，検証実施と検証結果の確認については，どのツールが利用可能か公知となっていない．本節では，大規模分散処理システムのシステム検証における，検証環境構築と検証実施／結果確認の効率的な実施という課題に対し，どのようなツールが利用可能か明らかにする．

4.2.1. 課題解決の考え方

初期のシステム開発の場合には，ツールが整備されておらず全て手作業となるため，検証作業の効率化の 1 つとしてツールの利用による自動化がある．自動化の他に検証の手順を効率化するという方法もあるが，数百台規模のマシンを操作する手作業の手順に対する効率化では限界がある．大規模分散処理システムのシステム検証では，このツールによる効率化が有効と考

え，汎用的な作業についてはOSSの適用を行う．また，OSSが適用できない独自の作業については，検証環境構築と検証実施／結果確認それぞれにスクリプトのツールを用意するという方法で対応する．OSSが適用できる作業はOSのライブラリ等のインストールやOS／NWの状態確認であり，通常のマシン運用作業については適用ができ効率化を図ることができる． OSSが適用できない作業としては，独自開発したプロダクト等におけるインストール時に設定ファイルの設定，実行時にパラメータチューニング，検証時における読み書き同時の性能測定等，システム特有の作業である．これらは OSS での共通的な機能では対応できないため，独自のスクリプトツールを適用する．

この解決の考え方は，OSSの利用とスクリプトの作成というアプローチであり，多数のマシンに対する制御や監視が必要な場合，大規模分散処理システム一般に有効である．

4.2.2. 具体的な解決方法

OSSが適用できる作業については，表 **9** のように効率化を図る．独自のスクリプトのツールについては，表 **10** のように対応する．表の設計（環境構築）における RPM のインストールを例にとると，OSS ツールでも環境設定として，ライブラリのインストールができるが，プロダクトのインストールが図 3 のように設定リストを参照しながらインストールをする等，通常の作業とは異なるため，OSSツールでは対応ができない．このため，インストール作業での独自スクリプトツールとして，表 **11** の項番 5 のインストールツールを作成して効率化を図った．

表 9. OSS ツール（例）

分類	ツール名	作業内容
設計（環境構築）	Puppet	・必要な RPM のインストール/アップデート ・設定の配布/登録 ・データ初期化 ・ファイルの一斉配布
実行（検証準備）	iperf	NW の帯域測定
	bonnie	Disk の I/O 測定
実行（結果確認）	MRTG	リソース情報可視化
	Nagios,Crane	マシン監視

表 10. スクリプト（例）

分類	スクリプトツール例
設計(環境構築)	・サーバリストの作成 ・ssh 疎通確認 ・CBoC の RPM のインストール/アップデート/アンインストール ・設定の配布/登録 ・データ初期化 ・ファイルの一斉配布
実行(検証実施)	・CBoC の起動/停止 ・CBoC への書き込み/読み込み負荷ツール ・CBoC への書き込み/読み込み負荷ツールの起動/停止 ・指定プロセスの停止 ・リモートマシンへの JOB 登録/確認/削除 ・データのバックアップ/リストア
実行(結果確認)	・CBoC 稼働状態確認 ・CBoC,負荷ツールのプロセス存在確認 ・CBoC,負荷ツールのログチェック ・定期的な CBoC 統計情報取得/取得停止 ・定期的なサーバリソース情報取得/取得状況確認/取得停止 ・tcpdump 情報の取得開始/取得状況確認/取得停止（ローカル、リモート） ・複数リモートマシンにおける同一コマンドの実行 ・複数リモートマシンからの同名ファイル収集

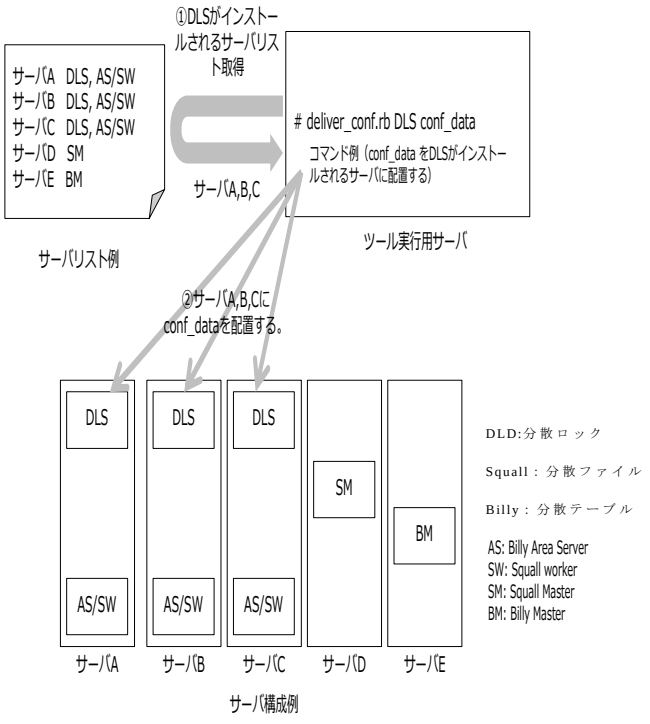

図 3. 環境構築ツールの動作イメージ

表 11. 環境構築時に用いた主な独自開発ツール

分類	スクリプトツール ファイル	機能概要
サーバリスト作成	generate_all_server_lists.sh	CBoC の環境構築および検証に必要な各種サーバリストファイルの生成
環境チェック	ssh_conn_check.rb	マシン間の ssh 接続可否確認
サーバ設定・配布	register_dls_cellname.sh	分散ロック セル名の /etc/hosts への登録
	install_pf_client.sh	検証用クライアントツールの配付およびインストール
	install_products.sh	CBoC 関連プロダクトの新規インストール
	deliver_conf.rb	CBoC 関連プロダクトの設定ファイル配布
	serve_files.rb	複数マシンへの同一ファイル群の配布
データ初期化	generate_squall_info.sh	分散ロック に登録する 分散ファイルの構成情報ファイルの生成
	register_squall_info.sh	分散ファイルの構成ファイルの 分散ロックへの登録
	initialize_cboc_data.sh	CBoC が保持する永続化情報の初期化
汎用	exec_ssh.sh	複数のリモートマシンにおける同一コマンドの実行

4.2.3. 適用結果

この方法を適用した具体的な実例と結果について示す．これらのツールの利用により，検証作業の効率化が図ることが可能となり，さらに人為的なミスも防ぐことができた．具体的には，従来の手作業では 1 週間かかっていた構築作業が，2 時間に短縮可能となった．また，検証実施と結果確認に 1 日かかっていた作業が，3 時間に短縮可能となった．

4.3. NW／Disk 環境のばらつきの回避

同一製造会社のカタログスペックが同じスイッチやケーブルで構成されたネットワーク（NW）では，どの末端から測定してもスループットは同じとなるはずである．また，マシン性能についても同様である．しかし，大規模分散処理システムの「分散かつ大規模」な検証環境で数百台のマシンの NW 環境とディスク（Disk）を調べると，実際には性能にばらつきが発生していることが判明した．この NW や Disk の性能のばらつきは，1 台 1 台では小さくても，数百台を組み合わせると性能測定や機能確認に影響を与えてしまう．したがって，大規模分散処理システムのシステム検証では，NW と Disk 環境のばらつきをどのように確認するかが技術的課題となる．

一般的な情報システムでは，マシン数台のみが NW で接続された環境での検証のため，1 台 1 台のばらつきの差異が小さい場合は，性能測定や機能確認に影響は与えない．本節では，大規模分散処理システムのシステム検証における，NW と Disk 環境のばらつきをどのように確認し解決するかについて説明する．

4.3.1. 課題解決の考え方

環境の確認を行うことで，性能が均一でない部分を見つけ対処をする．環境確認のポイントはマシン外側の NW 帯域とマシン内部の Disk I/O の 2 点であり，対処はパラメータチューニングである．実際に NW 帯域の確認において，複数のラックが論理的には同じ構成でも物理的には設定が異なっている場合がある．実際，図 4 のように中心的な NW 装置であるコアスイッチと末端のエッジスイッチ間で転送速度に約 1.5 倍の差が発生していた．また，Disk I/O の確認において，Disk ベンチマークツールの Bonnie++で測定した結果，図 **6** のように同一機種でも性能の良いものと悪いものとでは，約 2 倍の性能差が発生していた．これらのように，環境は，想定と異なり均一でない場合があり，事前に環境を確認することが重要となる．

この解決の考え方は，NW と Disk の性能確認というアプローチであり，検証環境が大規模で NW や Disk の性能のばらつきがある場合，大規模分散処理システム一般に有効である．

4.3.2. 具体的な解決方法

特に性能測定を実施する前に基本システム性能（Disk I/O スループット，NW 帯域）の測定を行い，マシン毎の性能状況や NW 帯域を確認して，必要に応じマシン交換や NW 設定の確認を行う．具体的な測定対象の環境と使用ツールを表 **12** に示す．NW 帯域の確認方法としては，図 **5** のようにコアスイッチとエッ

ジスイッチ間の転送速度を OSS の iperf で一方のラックから他方のラックへデータを送信することにより測定する.

4.3.3. 適用結果

この方法を適用した具体的な実例と結果として，CBoC タイプ 2 の検証環境では環境の影響を考慮した測定結果の解析が可能となり，解析の精度が向上したため再測定の必要がなくなった.

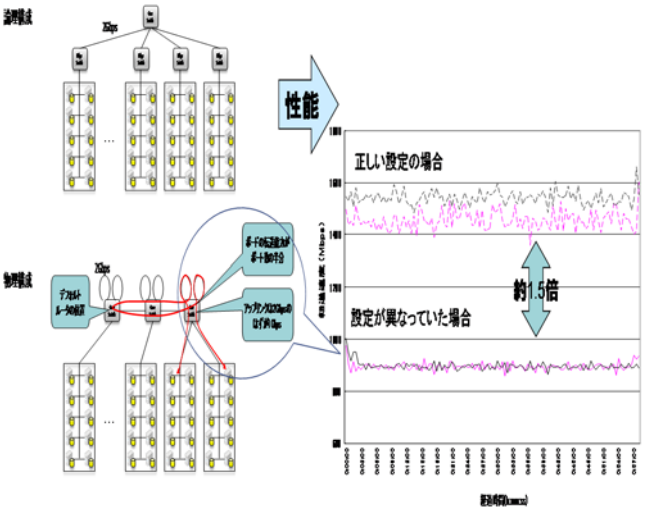

図 4. NW 帯域の違い

表 12. 測定対象の環境と使用ツール

測定対象	使用ツール	備考
マシンの Disk I/O	bonnie++	集計例を図 **6** に示す
マシンのメモリ・Disk I/O	hdparm	
NW 帯域	iPerf	集計例を図 **5** に示す

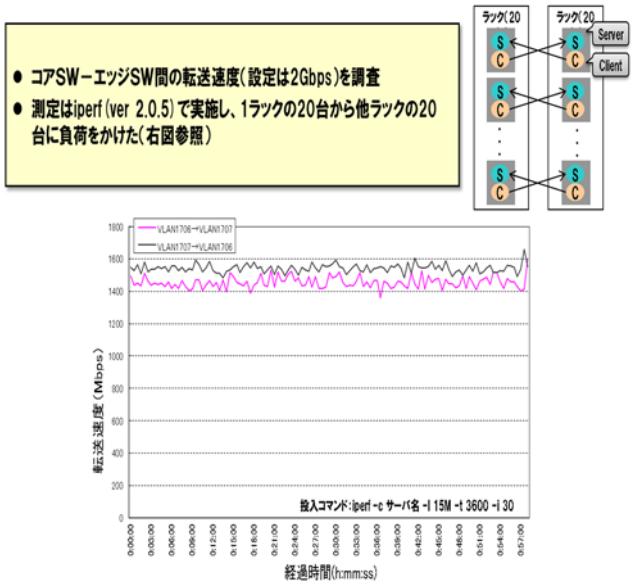

図 5. NW 帯域の確認方法

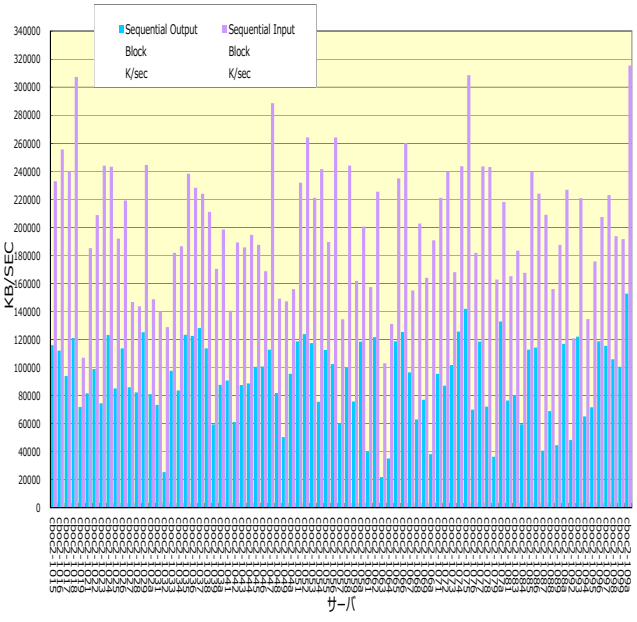

図 6. Bonnie++による測定結果

4.4. 故障検知と故障対応

大規模分散処理システムでは，利用するマシンの台数が数百台規模であるため，マシンあたり 1 年に 1 度の故障率でも日々マシンの故障が発生する状況となる. これを放置した場合，故障したマシン台数分の処理性能が落ちるだけでなく，そのマシンの処理がボトルネックとなり，機能に影響を与えることもある. そのため，マシンの故障を検証前に検知して交換等の対応をすることにより,検証への影響を回避する必要がある. ただし，マシン台数が多いため，効率的に実施しない

と検証開始が遅れることになってしまう．

一般的な情報システムでは，マシン数台のみのため，1 年に数度の故障発生であり，検証開始前に手作業で事前確認することが可能である．本節では，大規模分散処理システムのシステム検証における，マシン故障に対する検証への影響回避について説明する．

4.4.1. 課題解決の考え方

解決の考え方としては，監視ツールによる故障監視と予備マシンの用意で対応する．故障率はある程度範囲がわかるため，それを元に想定した故障マシン数と同程度の予備マシンを準備しておくことにより，検証に影響がないよう速やかに故障マシンを切り離し予備マシンに入れ替えることが可能となる．また，速やかな故障発見のために，通常のマシン監視に加え，スクリプトや Cron による定期的 SSH 接続による通信経路の確認やログ監視の自動化により，問題のリアルタイムでのエスカレーションを実現する．

この解決の考え方は，故障の即時検知というアプローチであり，マシン台数が数百台規模の場合，大規模分散処理システム一般に有効である．

4.4.2. 具体的な解決方法

4.2 の表 **9** に示した監視 OSS である MRTG と Nagios や Crane により故障を検知する方法を取った．MRTG では，Disk I/O／容量や NW の不具合等を数値の異常により確認する．Nagios や Crane では，syslog 等のシステムログに出力されたエラーメッセージだけでなく，追加の設定により大規模分散処理システムのエラーメッセージも逐次確認を可能とした．また，故障率とラック搭載可能マシン数を考慮し，予備マシン数は全体台数の 25%とした．

4.4.3. 適用結果

表 **13** に 2 年間で発生した故障率の実例を示す．故障の発生には，当初は「ある検証を実施すると発生する」,「あるプロダクトのマシンは発生する」等の仮説を立てたが，特定の作業による故障の傾向は見られなかった．一方で，故障したマシンは再度故障するという傾向は見られた．また，SSH で接続できないマシンがある状態は，ネットワーク層を利用する大規模分散処理システムでは，ネットワーク層の通信障害を意味し重大な故障であるが，通常の監視システムでは監視対象にならず発見が遅れることが多い．予備マシンが無い時は，マシン調整の実施から行い，復旧時期は不明であったが，提案の解決方法を故障率などが上記のような，CBoC タイプ 2 の検証環境において実施した結果，予備マシンが準備済みであることにより，多くても半日の作業で復旧が可能となった．

表 13. 故障率（例）

期間	故障件数	故障率(全体の台数に対する割合)	参考 SSH 接続不可件数
2010 年 10 月～3 月	27 件	9％（半年），0.05%（1 日）	13 件
2011 年 4 月～9 月	38 件	12％（半年），0.07%（1 日）	8 件
2011 年 10 月～3 月	32 件	10％（半年），0.05%（1 日）	16 件
2012 年 4 月～9 月	8 件	2％（半年），0.01%（1 日）	5 件

5. まとめと今後の課題

大規模分散処理システムのシステム検証の「テスト環境の開発」アクティビティにおける課題を 4 つの技術的課題として整理し，効率的に実施可能な解決方法を提案・実践して，その有効性を確認した．

提案方法は，TP の集中制御，ツールの利用，NW／Disk の確認，故障検知と故障マシン入れ替えというアプローチであり，汎用の PC サーバで構成された大規模分散処理システム一般に適用可能である．ただし，まだ課題は残っており，TP のリアルタイムの設定変更，ツールの集約，NW/Disk の自動確認，故障の傾向の分析ができておらず，これらは今後の課題である．

参　考　文　献

[1] ビッグデータ競争元年,ハーバード・ビジネス・レビュー,No.293(2013/2),ダイヤモンド社.

[2] 西田　圭介: Google を支える技術～巨大システムの内側の世界,技術評論社(2008).

[3] Junichi Niino: 「NoSQL」は「Not Only SQL」である，と定着するか？, http://www.publickey1.jp/blog/09/nosqlnot_only_sql.html, (2009).

[4] 鷲坂　光一,中村　英児,高倉　健,吉田　悟,冨田清次:大量データ分析のための大規模分散処理基盤の開発,NTT 技術ジャーナル,Vol.23,No.10,pp.22-25(2011）.

[5] Jerry Gao, Xiaoying Bai, Wi-Tek Tsai: Cloud Testing – Issues, Challenges, Needs and Practice, SOFTWARE ENGINEERING, An International Journal, Vol1, No.1,(2011).

[6] 坂井 俊之, 梅田 昌義, 中村 英児, 本庄 利森: 大規模分散処理システムのソフトウェア試験とその実践,情報処理学会デジタルプラクティス,Vol4,No.1(2013).

[7] 廣川 裕, 林 孝志, 山中 章裕, 吉田 悟: 商用サービス適用のための大規模分散処理システムの性能評価, Vol4,No.1(2013).

[8] Benjamin H. Sigelman, Luiz Andr´e Barroso, Mike

Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, Chandan Shanbhag: Dapper, a Large-Scale Distributed Systems Tracing Infrastructure, Google Technical Report,(2010).

[9] LIU, X., GUO, Z., WANG, X., CHEN, F., LIAN, X., TANG, J.WU, M., KAASHOEK, M. F., AND ZHANG, Z. D3S: Debugging Deployed Distributed Systems. In Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation, NSDI'08, USENIX Association, pp. 423–437(2008).

[10] 山根 智: 時間オートマトンによる分散システムの仕様記述と検証の方式の提案, 電子情報処理学会論文誌, Vol.J79-D-I No.8, pp.511-521(1996-8).

[11] 田村 慶信, 内田 雅也, 山田 茂, 木村 光宏: 分散開発環境に対する確立微分方程式に基づくソフトウェア信頼度成長モデルの一般化, 電子情報処理学会 信学技報, R2002-54(2002-11).

[12] Cem Kaner, Jack Falk, Hung Quoc Nguyen:Testing Computer Software 2nd edition, John Wiley & Sons, Inc(1999).

[13] Glenford J. Myers: ソフトウェア・テストの技法 第 2 版, 近代科学社(2012).

[14] ボーリス・バイサー: ソフトウェアテスト技法, 日経 PB マーケティング(2011)

[15] 長尾 真(監訳), 松尾正信(訳):ソフトウェア・テストの技法 第 2 版,近代科学社(2012).

[16] 松本吉弘(監訳):ソフトウェアエンジニアリング基礎知識体系 SWEBOK, オーム社(2004).

[17] Tom White, 玉川 竜司（訳）: Hadoop 第 2 版, オライリー・ジャパン(2011)).

[18] B.Cooper, A.Silberstein, E.Tam, R.Ramakrishnan, R.Sears: Benchmarking Cloud Serving Systems with YCSB, Proceedings of the 1st ACM symposium on Cloud computing, New York(2010).

[19] 大田尾一作: HTML5 時代のツール「Selenium2」で Web システムのテストを自動化, http://codezine.jp/article/detail/7427, CodeZine 2013.

[20] Savanna Project: https://savanna.readthedocs.org/en/0.3/, OpenStack Foundation 2013.