Vol.2010-HPC-126 No.23
2010/8/4

XQueryによる柔軟な問い合わせが可能な大規模分散環境モニタリングフレームワーク

加辺友也[†1]　田浦健次朗[†1]

本研究の目的は, 分散環境から得られる情報を蓄積し, それに対して問い合わせるという形式のインタフェースを提供することにより, システム管理者の管理負荷を軽減することである. 既存のモニタリングシステムでは主として障害の発生を検知し, システム管理者にその事実のみを知らせることを目的としており, その原因が何であるかまでを突き止めるのには不十分である. 大規模な分散環境においては障害の発生件数も増え, 原因の調査に要する時間も無視できなくなる. 本研究では情報をXMLで表現し, それに対してXQueryで問い合わせるというインタフェースを提供することで, システム管理者が障害の原因を特定するのに要する時間を削減するためのフレームワークを実装した. また実際に利用されるようなケースを想定した実験を行い, 10秒程度で問い合わせが完了することを確認した.

Monitoring Framework for Large-Scale and Distributed Environments that is Flexibly Queryable with XQuery

Tomoya Kabe[†1] and Kenjiro Taura[†1]

We targeted at reducing administration time for systems administrators by gathering information from distributed environments and providing an interface to query the information. Previous monitoring systems are insufficient to tell them the cause of problems since their targets are to detect the occurrences of problems and tell them the facts of the occurrences. It is hard to ignore the time to investigate the cause of problems due to the increasing number of occurrences of anomalies in large scale and distributed environments. We implemented a framework to reduce the time for systems administrators to identify the cause of the problems by providing an interface that they query with XQuery. Experiments showed the framework can finish querying in about ten seconds.

1. はじめに

近年, InTrigger(日本)[3], Grid'5000(フランス)[2] など大規模な分散環境の普及によって, 各コンピュータの状況把握の要求が高まっている. 数十 ~ 数百以上のコンピュータ群を結合した大規模な分散環境においてこれらを管理維持していくためには, システム管理者は多くの時間を割かなければならない. コンピュータの数が増えれば増えるほど故障率が増大してしまい, 問題が発生すれば管理者は問題を認識し, 原因を突き止め, 解決しなければならない.

システム管理者の手間を削減するため, 様々なモニタリングシステムが開発されており, 広く用いられている. しかし, これらのモニタリングシステムの多くは問題を発見することに役立つ一方で, その原因が何であったのかを知るための作業は従来通り必要となってしまっている.

本研究ではマルチクラスタ環境を想定し, 問題の原因を突き止める部分の手間を削減することを目指す. コンピュータから得た時間的に変化する情報をデータベースに蓄え, システム管理者が必要な情報を問い合わせることにより, 有用な情報のみを提示する. マルチクラスタ環境では物理的に近いものと遠いものが明確に分かれているため, 処理を効率的に行うために木構造が有利である. そこで本研究では取得する情報をユーザが明示的に記述し, データ表現形式としてXMLを用い, 問い合わせ言語としてXQueryを用いたモニタリングフレームワークを提案する.

2. 関連研究

Ganglia[1)11)] は各コンピュータの情報を相互に交換して保持しておき, 最終的にはWebブラウザなどから各ノードの各データについての, 時間的変化を時系列グラフを用いて監視することができるモニタリングシステムである. また複数のクラスタの情報を1箇所で監視することができる. しかし, Gangliaの元々の目的は残っているメモリ, ディスク容量などの資源を監視するためのものであるため, 取得した情報そのものを表示するという機能しか持たない. したがって, 保持している情報に対してユーザが自由に情報のクエリを行うためには, 別途Gangliaが内部で利用しているデータベースであるRRDtool[4] に対して問い合わせを行わなければならないが, 問い合わせは独自形式であり, システム管理者の学習コストが高く

†1 東京大学大学院 情報理工学系研究科
Graduate School of Information Science and Technology, The University of Tokyo

なってしまう.

Nagios[9] はユーザが設定したプラグインを用いて情報を取得し, 異常を検知したとき電子メールで報告する機能をもつ障害監視ソフトウェアである. 異常を検知するためのプラグインはユーザが自由に記述することができ, 状態を正常, 注意, 異常の 3 段階に分けて監視することができる. しかし, Nagios は問題の発生を一早くシステム管理者に知らせることを目的としており, 情報をクエリする機能をユーザに提供していないため, システム管理者が問題の原因を特定するのに用いるのには不適である.

UbiCore[10] はセンサネットワークからのデータを XML ストリームへと変換し, これに対してユーザが記述した XQuery を用いて問い合わせることによってデータを加工し, 必要なデータのみをバックエンドアプリケーションへと送信するミドルウェアである. UbiCore では情報源毎に過去に取得したデータもある程度保持しており, 時間的な変化を検出することができるようになっている. UbiCore で想定している情報源は複数の物理的なセンサであり, データを 1 箇所に集約して処理するという動作モデルを採用している.

3. 提 案 手 法

3.1 動 作 環 境

本フレームワークは Debian GNU/Linux Kernel 2.6.18 および 2.6.26(32/64 bit), Python 2.5.2 で動作する. また XML データベースとして Sedna Version 3.2.91[5] を利用している.

3.2 ユーザインターフェース

ユーザ (システム管理者) は情報取得モジュール (3.2.1), 問い合わせクエリ (3.2.2) の 2 つの部分を実装することによって, システムを利用することができる.

提案手法ではコンピュータから取得した情報を表現する手段として XML[7] を用い, XML データを XML データベースに格納する方式を選択した. これは XML のスキーマが木構造であり, レコード構造をもった関係データベースに比して自由であること, 木構造であることでユーザがデータを扱いやすいこと (3.4 で述べる), そして現在利用対象としているマルチクラスタ環境との相性が良いことによる. この XML データベースに対し, クエリ言語として XQuery[6] を用いている. これは XML と同じく W3C により標準化されており, 木構造である XML に対して強力な表現力を持っているためである.

3.2.1 情報取得モジュール

情報取得モジュールはコンピュータから情報を取得し, それを本フレームワークでのデータ表現形式である XML に変換するモジュールである. 本フレームワークは Python スクリプトで書かれているため, ユーザは Python のクラス定義を行うモジュールを作成する. このクラスオブジェクトは情報更新を行う `update()` および 1 つの XML 要素からなる XML 文字列を返す `xml()` メソッドを持つ (Listing 1). 複数のモジュールおよびクラスを定義しておき, 内部でそのクラスを呼び出すこと, ライブラリの本体を C などの言語で記述し, Python でラップすることなどは自由に行える. このモジュールとクラスの情報は情報取得間隔と情報保持数と共に設定する.

Listing 1: Python モジュールに要請されるインタフェース

```
class Klass(object):
    def __init__(self,):
        Constructor(optional)
    def update(self,):
        Update internal information
    def xml(self,):
        Return XML string
```

3.2.2 問い合わせクエリ

問い合わせクエリは各情報源から情報を取得するための XQuery である. ユーザはクエリにおいて各ノードで実行される関数 (nodefunc), Head ノードで実行される関数 (clusterfunc), Root ノードで実行される関数 (rootfunc) の 3 つを定義する必要がある. これを Root ノードに渡すことによって, 後述する手順で処理結果が返される.

3.3 動作モデル

各クラスタから 1 台を選出し, それをクラスタの Head ノードとする. また, 分散環境に存在している全コンピュータの中から 1 台を選出し, それを Root ノードとする. 現在の実装では各クラスタ毎にデータベースサーバを 1 台置くか, 各コンピュータがデータベースサーバになるか設定することができる (図 1).

本システムは動作開始時に情報取得モジュール (3.2.1) で記述されたクラスのインスタンスを生成する. その後, 設定に従って定期的に `update()` メソッドを呼び出し, 続いて `xml()` メソッドを呼び出す. ここでの出力が XML データベースへと格納される. 過去の情報を含む情報の検索を可能にするため, データベースへはその時刻毎のデータを複数蓄積しておくことができる. 蓄積する上限は情報源ごとに設定可能である.

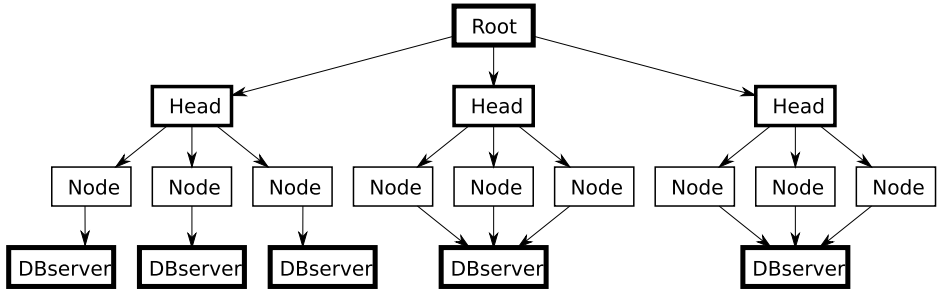

図 1: システムコンポーネントのモデル
Fig. 1: System component model

Root ノードは XQuery を受け取るとそれを各クラスタの Head ノードへ転送する. Head ノードはそれをクラスタ内のコンピュータへ転送する. 各コンピュータは XQuery 中で nodefunc を実行し, 結果を Head ノードへ返す. Head ノードは各コンピュータからの結果を集め, それに対して clusterfunc を実行し, その結果を Root ノードへ返す. Root ノードはまた同様に, Head ノードの実行結果を集め, それに対して rootfunc を実行し, 結果を呼び出し元へと返す.

3.4 XML スキーマ

本フレームワークではコンピュータから取得した全情報を 1 つの XML 木として扱う. 全情報が 1 つになっていることで, 複数の情報源からの情報を同時に扱うことができる. 各コンピュータが取得した情報はそのコンピュータ上にのみ, あるいはクラスタに 1 つ設定されたデータベースサーバに保持されているため, 特定の 1 ノードにシステム全体のデータを格納しなければならないほど巨大なストレージが必要となることはない. ユーザはあたかも<root>, <cluster>, <node> で階層化され, <node> 内では情報源毎に複数のスナップショット格納されている Listing 2 のような XML データ構造があるかのように XQuery を記述することができる.

Listing 2: XML 構造

```
<root>
  <cluster>
    <node>
      <builtin>
        <source1>
          <snapshot>
            <time> ... </time>
            xml() の出力
          </snapshot>
          <snapshot> ...
        </source1>
      </builtin>
      <userdefined>
        ...
      </userdefined>
    </node>
    <node>
      ...
    </node>
    ...
  </cluster>
  <cluster> ...
</root>
```

<root> 要素以下はシステム全体, <cluster> 要素以下は各クラスタ, <node> 要素以下は各ノード内の情報をそれぞれ表している. <builtin> の下にある情報はシステムに組み込まれた情報源（主に/proc ファイルシステム[8] から取得される情報を想定）を表し, <userdefined> の下にある情報はユーザが自由に設定した情報源であることを示す. いずれも xml() の出力にデータ取得時刻として<time> 要素が追加されて格納される.

木構造をもつ XML を利用することにより, ユーザはフレームワークの動作モデルに関してほとんど考慮しなくてよい. また, 主に我々が情報源として想定している /proc ファイルシステムから得られる情報も多くが木構造として自然に表現可能である一方, それぞれのデータスキーマは別々であり, 関係データベースへ格納するより木構造のデータ構造とした方がユーザにとって直感的であると考えられる.

表 1: 設定した情報源とそのパラメタ
Table 1: Information sources and parameters

情報源	ディスク使用量	ユーザ情報	CPU 情報	CPU 負荷	SSH 接続可否
取得間隔（秒）	3600	60	3600	60	300
最大保持数	5	2	5	60	30
情報取得方法	/proc/mounts, statvfs(2)	/etc/passwd, NIS	/proc/cpuinfo	/proc/loadavg	SSH 接続試行

4. 評　　価

本フレームワークにより目指している時間的, 空間的に広がりを持った情報に対する処理が行えるようになったかどうか, 3 種類の評価実験を行った.

評価環境として InTrigger プラットフォーム[3] および本研究室で所有する 3 クラスタ 96 ノードを用いた. 各情報源および取得間隔, データの最大保持数, データ取得方法を 表 1 に示す.

4.1 ディスク使用量の監視

時間的にも空間的にも一切広がりのない単純な例として, `/proc/mounts` から得られるコンピュータ上のディスクマウント情報と statvfs(2) システムコールによって得られる利用可能ブロック数, 総ブロック数の情報を利用して, 「使用量が 80%を越えているディスクとコンピュータを列挙する」ことが可能となった.

システムからディスク使用量を取り出す部分（データの取得には`/proc/mounts` および `statvfs(2)` システムコールを利用している）を Listing 3 に, 実際に問い合わせに利用した XQuery コードのうち, 全ノードで実行される部分を Listing 4 に示す.

Listing 3: ディスク使用量を知るための xml() が出力する XML データの例

```
<data>
  <fs>
    <fs_file>/var</fs_file>
    <fs_spec>/dev/sda5</fs_spec>
    <f_blocks>2579457</f_blocks>
    <f_bavail>1986674</f_bavail>
    ...
  </fs>
  <fs>
    ...
</data>
```

Listing 4: ディスク使用量が多いものを返す XQuery

```
declare function local:nodefunc($node) {
  let $nodename := $node/node/nodeinfo/nodename
  let $snapshot := $node/node/builtin/statvfs/snapshot[1]/data
  let $u :=
    for $fs in $snapshot/fs
        let $fsname := $fs/fs_file
        let $fsspec := $fs/fs_spec
        let $usage := 100 * ($fs/f_blocks - $fs/f_bavail)
                        div $fs/f_blocks
        where
          $usage > 80
          and not(contains($fsspec, ":")) (: exclude shared FS :)
        return <fs>
                 { $fsname, $fsspec }
                 <usage>{ $usage }</usage>
               </fs>
  where not(empty($u))
  return <node>{$nodename, $u}</node>
};
```

4.2 負荷変動の検知

時間的な変化を適切に扱うため, `/proc/loadavg` を読むことによって得られる CPU 負荷から, 負荷変動を検出することを考える. この情報を 1 分おきに記録しておくことによって, 「3 分前と比較し, 負荷が 0.5 以上上昇した計算機を列挙する」ことが可能となった. これは過去の情報を保持しておき, それを利用するインターフェースが用意されているからこそ可

能な処理である. XQuery の例として各ノードで実行される関数を Listing 5 に示す.

Listing 5: 負荷が上昇したノードを列挙する XQuery

```
declare function local:nodefunc($node) {
  let $cur_loadavg_one :=
    $node/node/builtin/loadavg/snapshot[1]/data/load_one
  let $old_loadavg_one :=
    $node/node/builtin/loadavg/snapshot[4]/data/load_one
  where $cur_loadavg_one - $old_loadavg_one > 0.5
  return
  <node>
  {
      $node/node/nodeinfo/nodename,
      $cur_loadavg_one,
      $old_loadavg_one
  }
  </node>
};
```

4.3 障害検知

SSH 接続の可否によって障害があるかどうかを判別する. これを利用して「3 台以上の正常な計算機から『障害がある』と報告されたノードを, その報告をしたノードの一覧とともに列挙する」ということが可能となった. 「3 台以上」という制約によって局所的ですぐに復旧するようなネットワーク不通などは対象外とすることができる. これは複数のノードからの情報を処理するインタフェースを提供しているからこそ可能な処理である. 障害検知のために利用した環境を図 2 に示す. システム中の各ノードはクラスタ内で全対全での SSH 接続を試行し, 各クラスタの Head ノードは Head ノード同士での全対全での試行を行う. 実験環境では 3 つのクラスタ環境を用いたため, どのクラスタにも属しておらず, 現在稼働していないノード (oooka-mini-charlie) を別途 Head ノードとして扱うように追加して実験を行った. 結果として返された XML を Listing 6 に示す. この結果, sheep クラスタ内で故障しているノードおよび, 意図的に追加した稼働していないノードが他の Head ノードにより検出されていることがわかる.

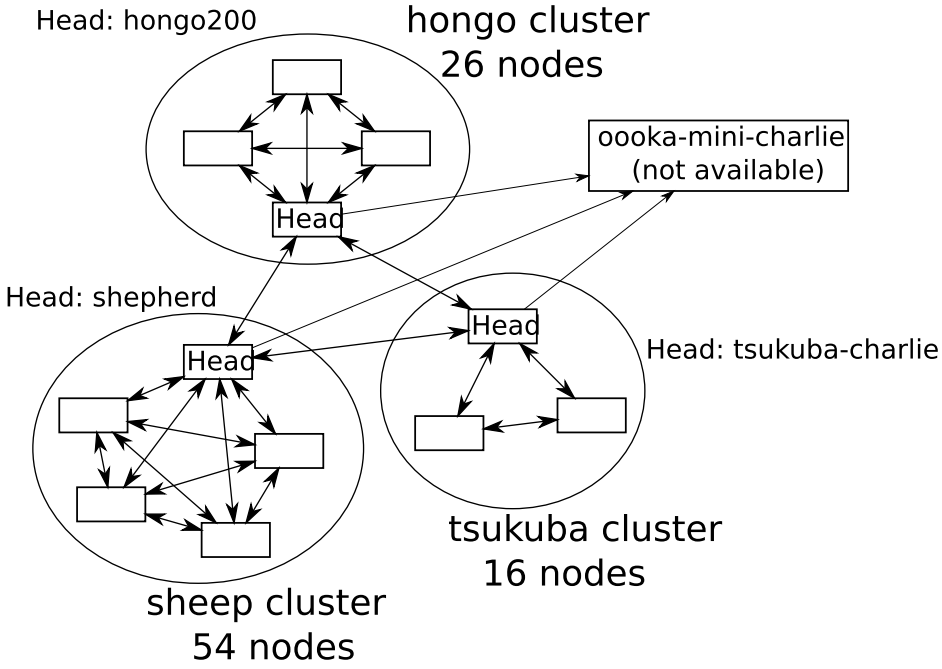

図 2: 障害検知において用いた環境
Fig. 2: Environment used for anomaly detection test

Listing 6: SSH による接続試験の結果

```
<ret>
 <fail>
  <node>sheep46</node>
  <from>
   <nodename>sheep64</nodename>
   <nodename>sheep59</nodename>
   ...
```

```
    <nodename>shepherd</nodename>
   </from>
  </fail>
  <fail>
   <node>sheep02</node>
   <from>
    <nodename>sheep64</nodename>
    <nodename>sheep59</nodename>
     ...
    <nodename>shepherd</nodename>
   </from>
  </fail>
  ... 複数台分上記と同じような結果が続く ...
  <fail>
   <node>oooka-mini-charlie.m.gsic.titech.ac.jp</node>
   <from>
    <nodename>shepherd</nodename>
    <nodename>tsukuba-charlie</nodename>
    <nodename>hongo200</nodename>
   </from>
  </fail>
</ret>
```

なお, このフレームワークそのものがシステムに与える負荷は, 1 回の XQuery 問い合わせで各ノードで約 0.04 秒の CPU 時間を消費する程度である. それ以外の CPU 時間はほぼ全てが XML データベースに格納するコストである. その他遅延やソフトウェアオーバヘッドを含め, 全体として 1 回の問い合わせには約 10 秒を要した.

5. おわりに

本稿では, XML と XQuery を利用し, 少ないユーザ操作で表現力の高いモニタリングを行うためのフレームワークを提案した. 実装を行い, 従来では困難だった時間的な変化, 空間的に広がっている情報を複合的に利用しなければ実行できないような, 有用な目的のモニタリングに利用可能であること, そして動作時に過大な負荷をかけないことを示した.

しかし, 現在の実装では Root ノード, Head ノード, 計算ノードの 3 段階の構成を前提としており, どのようなクラスタ環境にも適用できるものではない. また, 現在の構成では単一故障点が発生しやすくなってしまっているため, 冗長化が可能なよりよいモデルが必要である. 今後は対象となる環境のモデルを再考し, より広く適用可能となるよう改良を進める.

謝辞 本研究の一部は文部科学省科学研究費補助金特定領域研究「情報爆発に対応する新 IT 基盤研究プラットフォームの構築」の助成を得て行われた.

参 考 文 献

1) : Ganglia. http://ganglia.info/.
2) : Grid'5000. http://www.grid5000.fr/.
3) : InTrigger Platform. http://www.intrigger.jp/.
4) : RRDtool. http://oss.oetiker.ch/rrdtool/.
5) : Sedna XML Database. http://modis.ispras.ru/sedna/.
6) : XQuery 1.0: An XML query language. W3C Recommendation.
7) Bray, T., Paoli, J., Sperberg-McQueen, C., Maler, E. and Yergeau, F.: Extensible markup language (XML) 1.0, *W3C recommendation*, Vol.6 (2000).
8) Faulkner, R. and Gomes, R.: The Process File System and Process Model in UNIX System V, *USENIX*, pp.243–252 (1991).
9) Harlan, R.: Network management with Nagios, *Linux Journal*, Vol. 2003, No. 111, p. 3 (2003).
10) Lee, H.S. and Jin, S.I.: An Effective XML-Based Sensor Data Stream Processing Middleware for Ubiquitous Service, *Computational Science and Its Applications - ICCSA 2007*, pp. 844–857 (2007).
11) Massie, M.L., Chun, B.N. and Culler, D.E.: The ganglia distributed monitoring system: design, implementation, and experience, *Parallel Computing 30*, pp.817–840 (2004).