社団法人 情報処理学会 研究報告
IPSJ SIG Technical Report

2004－HPC－99 (37)
2004／8／1

ポータビリティの高いジョブスケジューリングシステム設計と実装

町田悠哉[†] 中田秀基[††,†] 松岡聡[†,†††]

グリッド環境に存在する不安定性と非均質性という２つの特徴に対応したジョブスケジューリングシステム Jay について述べる．Jay はウィスコンシン大学で開発されたスケジューリングシステム Condor の構造をベースとし，ポータビリティを高めるため Java で実装を行った．ここで Java にはプロセスのユーザ ID を安全に変更する手法がないという問題が生じたが，JNI をサポートしていない Java 環境においても稼働するようなポータブルな C++デーモンを開発することでこの問題に対処した．小規模環境における評価実験により，本システムが耐故障性と高いポータビリティを備えていることが示された．

Design and Implementation of a Highly Portable Job Scheduling System

YUYA MACHIDA,[†] HIDEMOTO NAKADA[††,†] and SATOSHI MATSUOKA[†,†††]

We present a job-scheduling system for the Grid, Jay. Jay handles two difficulties inherent in the Grid: namely heterogeneity and instability. Jay is based on the techniques of Condor, which was developed at the University of Wisconsin, and has been implemented in Java for better portability. Although Java does not have a secure way of changing user IDs of an arbitrary process, we resolved the problem in Jay by developing a highly-portable C++ daemon that achieves this property and can run in Java environments that does not support JNI. The results of small-scale experiments show its fault-tolerance and high portability.

1. はじめに

広域ネットワーク上に分散した多数の計算資源を統合し，仮想的に高性能な計算機を構築するグリッドと呼ばれるシステムが普及しつつある．このようなグリッドシステムは，ユーザの大規模な計算要求を満たす技術であり，概して長期間に渡る計算を必要とするようなジョブを対象としている．そのため，サブミットされたジョブが終了するまで計算に関わるマシンやネットワークなどすべてのエンティティが障害を起こさず，正常に動作し続けるという保証はなく，実行時間が長くなるほどそのような障害が発生する確率も高くなる．また，多数の計算資源が統合されたグリッド環境では，ソフトウェア的にもハードウェア的にもすべての計算資源を統一することは不可能であり，非均質な環境と考えなければならない．したがってグリッド環境においてユーザに有用なジョブスケジューリングシステムを構築するためには耐故障性と高いポータビリティは欠くことのできない要素である．

耐故障性を備えたシステムの例として Condor[1)~3)] がある．Condor にはチェックポイントやプロセスマイグレーションなどの機能を備えており，広く普及しているシステムである．しかし，Condor では，特定のプラットフォームしかサポートされておらず，ソースが非公開であるため，ジョブスケジューリングシステムの研究基盤としては利用することができない．

そこで，本稿では，耐故障性を備えた Condor の構造をベースとし，ポータビリティの高いジョブスケジューリングシステム Jay の設計・実装について述べる．本システムはポータビリティを高めるため Java で実装を行った．しかし，Java ではプロセスのユーザ ID を変更することが不可能になるという弊害が生じた．そこで C++で記述されたデーモンを用意することでこの問題に対処した．Jay の有効性を示すため，長時間の計算を必要とするジョブをサブミットし，計算途中でリブートなどの外乱を与えたが，プログラムは無事終了し，小規模環境における耐故障性が示された．また，多少のコード修正で異なるアーキテクチャ上でもシステムが稼働することを確認し，ポータビリティの高さが示された．

† 東京工業大学
Tokyo Institute of Technology
†† 産業技術総合研究所
National Institute of Advanced Industorial Science and Technology
††† 国立情報学研究所
National Institute of Infomatics

2. Condorの概要

Condorはウィスコンシン大学で開発されたジョブスケジューリングシステムである．Condorでは，ハイスループットコンピューティングの実現を目指しており，計算資源の遊休時間をいかに効率的に利用できるかが鍵となる．

Condorではサブミットされたジョブを実行するマシンの集合をCondorプールと呼ぶ．Condorプールにはセントラルマネージャーと呼ばれるマシンが1台存在し，実際のスケジューリングを担当する．ユーザがCondorプールにジョブをサブミットするとジョブの情報はセントラルマネージャーに送信される．また，Condorプール内の各マシンもマシンのモニタリング情報をセントラルマネージャーに送信する．セントラルマネージャーに送信されるジョブやマシンの情報はClassAd[4),5)]と呼ばれるデータとして扱われる．セントラルマネージャーは回収したClassAdに対してマッチメイキングと呼ばれる操作を行い，各ジョブに適したマシンが割り当てられる．

またサブミットするプログラムをCondorシステムコールライブラリとリンクさせておくことにより，ジョブは計算の実行中に定期的にチェックポイントが取られ，チェックポイントファイルがクライアントのローカルディスクもしくは専用のチェックポイントサーバーに保存される．実行マシンがクラッシュした場合は，他のマシンでチェックポイントファイルからジョブを再開することが可能で，それまでの計算が無駄になるのを防ぐことが可能である．また，プライオリティの高いジョブがサブミットされた際にプライオリティの低いジョブのチェックポイントを取り，他のマシンへマイグレーションさせることも可能である．サブミットするプログラムは必ずしもCondorシステムコールライブラリとリンクさせる必要はないが，その場合にはチェックポイントは取られず，マシンのクラッシュ時などには最初から計算が実行される．

3. Jayの設計

3.1 システムの要件と設計方針

グリッド環境には不安定性・非均質性という2つの特徴がある．そこで，これらの特徴に対処するためシステムを設計する際の要件として耐故障性と高いポータビリティが挙げられる．

まず，耐故障性については，マシンのクラッシュなどの障害により，サブミットされたジョブの情報が失われることを避けなければならない．これが保証されていないシステムでは，ユーザは常に自分がサブミットしたジョブのモニタリングを続け，計算結果を受け取る前に障害が発生した場合には，再びジョブをサブミットしなければならない．これを避けるためにサブミットされたジョブの情報はファイルなどの永続性がある資源に(コンシステントな)状態を常に保持しておき，障害が起きたとしてもそのファイルから障害発生前の状態に復旧できるようにしておく必要がある．また，ハードウェア的には正常に動作しているにも関わらず，ソフトウェア的な障害によりデーモンが落ちてしまい，システムが稼働しなくなるという事態も避けなければならない．このような障害を防ぐため，デーモン群を監視するデーモンを用意することにより，耐故障性の向上をねらう．また，ポータビリティに関しては，プラットフォームに依存しないようにJavaで実装を行い，高移植性の実現を目指す．

3.2 システムの概要

図1に本システムの概要を示した．本システムの基本的な構造はCondorを参考として設計されている．各マシンは以下に示す3種類の役割を担う．ただし，単一のマシンが複数の役割を担うこともある．

- サブミットマシン
- 実行マシン
- セントラルマネージャー

サブミットマシンはサブミットされたジョブの情報の管理を行い，セントラルマネージャーから通知された実行マシンにジョブの実行を依頼する．ジョブがサブミットされるとサブミットマシンはジョブ情報を取り出し，このマシンが管理するキューに格納する．キュー内のジョブ情報は定期的にセントラルマネージャーに送信される．また，キューに格納されたジョブ情報はファイルに書き出され，ジョブ情報に変更があった場合には，その変更情報がファイルに追記される．そしてジョブが終了するとそのジョブ情報がキューから削除され，ファイルにも終了したことが追記される．これにより，ユーザはつねにマシンやジョブ状態の監視を続けなければいけないという状態から解放される．サブミットマシンはクラッシュなどが発生し，キュー内の情報を失ったとしても書き出しておいたファイルを解析し，クラッシュ前のキュー状態に復元することが可能となる．

実行マシンはマシン情報の管理を行い，サブミットマシンから依頼されたジョブを実行し，終了するとその結果を返す．マシン情報にはこのマシンのアーキテクチャやOS，メモリなどの情報の他にマシンの所有者により定義された利用ポリシーも含まれており，利用ポリシーを満たしたジョブのみがこの実行マシンで実行される．このマシン情報は定期的にセントラルマネージャーに送信される．

セントラルマネージャーはマシン情報やジョブ情報を基にスケジューリングを行うマシンであり，必ず1台のマシンをセントラルマネージャーとして用意する必要がある．セントラルマネージャーはマシン情報やジョブ情報を基に各ジョブを実行するのに適当なマシンを割り当てる作業(マッチメイキング[4),5)])を定期的

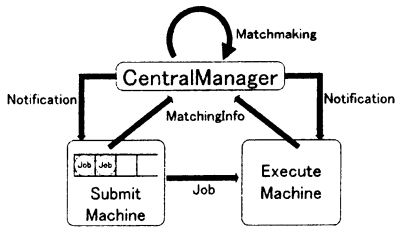

図 1 Jay の概要

に行う．Jay システムでは，ユーザごとにプライオリティが管理されており，プライオリティに応じて公平にマシンが割り当てられる．

セントラルマネージャーはステートレスな情報管理を行っており，サブミットマシンや実行マシンは，セントラルマネージャーの状態に依存せず，ジョブやマシン情報を定期的に送信する．これにより，セントラルマネージャーがクラッシュした場合などに再起動後，迅速にクラッシュ前のシステム状態に戻すことが可能となる．また，すべてのマシンにおいて稼働デーモンを監視するデーモンが用意されているため，可能な限りシステムが自律的に復旧できるようになっている．

4. Jay の実装

ここでは，Jay の実装について述べる．

4.1 デーモン

機能に応じて以下の 5 つのデーモンに分類する．

Master Master デーモンはすべてのマシンにおいてマシン起動時に稼働を開始し，他の必要なデーモンを起動する．また，定期的に他のデーモンが稼働しているかチェックし，止まっている場合には，再起動を行う．

Collector Collector デーモンはセントラルマネージャーで稼働しており，ジョブやマシンの情報を回収し，キューに格納する．Negotiator からリクエストが来ると，キュー内の情報を Negotiator に送信する．

Negotiator Negotiator デーモンはセントラルマネージャーで稼働しており，Collector から情報を受け取り，それをもとにマッチメイキングを行う．マッチメイキングが完了すると該当するマシンの Schedd と Startd に通知を行う．それ以降のジョブ実行には関わることはない．

Schedd Schedd デーモンはサブミットマシンや実行マシンで稼働しており，ジョブがサブミットされるとキューに格納し，各ジョブの情報をファイルに書き出す．定期的にキュー内のジョブの情報を Collector に送信する．Negotiator から通知が来ると Shadow を起動する．Shadow は Negotiator から通知されたマシンの Startd にジョブの実行を依頼する．ジョブの実行が終了するとそのジョブの情報をファイルとキューから削除する．

Startd Startd デーモンは実行マシンで稼働しており，定期的にマシンの状態を調べ，マシンオーナーの設定ポリシーとともにマシン情報を Collector に定期的に送信する．Shadow からジョブの実行を依頼されるとそのジョブが実行できる状態にあるかを調べ，起動できる場合には Starter を起動する．Starter は Shadow から依頼されたジョブを実行し，結果を返す．

これらのデーモンはセキュリティ基盤として Globus Toolkit[6] の GSI(Grid Security Infrastructure)[7),8)] を Java Cog Kit 1.1[9] 経由で利用している．それに対して Condor のデーモンは基本的に IP アドレスをベースとして相互認証を行っている．しかし，IP アドレスは詐称可能であり，セキュリティ基盤としては不十分である．GSI を導入した本システムは，よりグリッド環境に適したセキュアなシステムであると言える．

```
MyType        =  "Job"
TargetType    =  "Machine"
Cmd           =  "sim"
Out           =  "out"
Err           =  "err"
Arch          =  "INTEL"
OpSys         =  "LINUX"
Requirements  =  (Target.Arch == "INTEL")
                 && (Target.OpSys == "LINUX")
```
図 2 ジョブの MatchingInfo の例

```
MyType        =  "Machine"
TargetType    =  "Job"
Memory        =  256
Out           =  "output"
Args          =  100000
Owner         =  "smith"
Requirements  =  (Target.Arch == "INTEL")
                 && (Target.OpSys == "LINUX")
Rank          =  Target.Memory
```
図 3 マシンの MatchingInfo の例

4.2 MatchingInfo

ジョブやマシンに関する情報は MatchingInfo として送受信を行う．MatchingInfo は，属性名と属性値のリストであり，Condor における ClassAd に相当するものである．図 2 にジョブの，図 3 にマシンの MatchingInfo の例を示す．`Requirements` や `Rank` 属性により，柔軟なスケジューリングが可能となる．

4.3 実装における問題点とその対処法

ここでは，Java で実装を行うことにより生じる問題点とその対処法について述べる．

4.3.1 問 題 点

Java を使うことによる弊害は，Java がプラットフォーム間の完全な互換性を保つため一部の OS 機能をサポートしていないことである．その中には setuid/seteuid によるユーザ ID の変更機能が含まれている．これにより，ジョブはサブミットしたユーザとは異なるユーザ ID で実行されることになり，ファイルへのアクセスが制限されてしまう．

また，このユーザ ID の変更は慎重に行わなければ，任意のユーザを詐称される危険性を含んでおり，セキュリティの確保が必須の要素である．

4.3.2 対 処 法

Java ではセキュアかつポータブルにユーザ ID を変更できないという問題に対処するため，本システムでは，C++で記述されたデーモンを用意することにより対処する．このデーモンは，すべてのマシンで稼働しており，任意のユーザ ID でジョブを起動し，監視するという作業を担当する．

Kickd デーモンを稼働させるため，本システムでは，システム稼働用の特別なユーザアカウント (jay とする) を用意しなければならない．Jay のデーモン群はこの jay アカウントで起動される．そして Kickd デーモンは jay からの要求のみ受け付けるようにする．Kickd デーモンは root アカウントで起動されるが，起動直後に seteuid システムコールを用いて実効 ユーザ ID を jay にして稼働する．プロセスの起動要求が来ると，認証完了後，まず seteuid システムコールで 実効ユーザ ID を root に戻し，その状態でプロセスを fork，直後に 実効ユーザ ID を jay に再セットする．子プロセス側では setuid システムコールで 実ユーザ ID を要求されたユーザ ID にセットして実行する．

Kickd デーモンでは図 4 のように認証を行う．まず，プロセスの起動要求が来るとある名前のファイルを作るように指示する．要求側は指示通りにファイルを作り，OK メッセージを送る．Kickd デーモンは OK メッセージを受け取ると作成するように指示したファイルの所有者を調べ，所有者が jay の場合のみ要求通りにプロセスの起動を行う．このように Kickd デーモンでは，ポータビリティと簡便性のために，ファイルベースでの認証作業を行っているが，将来的には GSI による認証も導入する予定である．

4.3.3 課 題

Kickd デーモンの導入により Java 実装による問題点は解決されるが，Kickd デーモンは root 権限で起動しなければならないので Master デーモンの管理下にはない．そのため Kickd デーモンが落ちた場合にはそのマシンではジョブが実行されなくなるという問題がある．この点については今後の課題として挙げられる．また，Java Native Interface(JNI) を用いて Java で setuid を行うという代替案も考えられる．

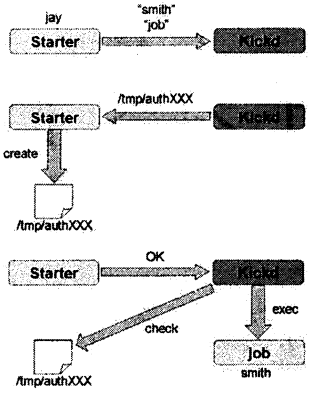

図 4 Kickd デーモンにおける認証

これにより容易にユーザ ID の変更が可能となるが，ユーザ ID 変更部分を切り出し，認証を行うなどの手法により慎重にセキュリティを確保する必要がある．

4.4 ジョブ実行の流れ

ジョブがサブミットされてからジョブが終了するまでの流れを説明する (図 5)．実際にジョブをサブミットする際には図 6 のようなサブミットファイルを用意する必要がある．

(1) Startd がマシンの MatchingInfo を Collector に送信する
(2) ジョブがサブミットされる
(3) Schedd がサブミットファイルを解析し，ジョブの MatchingInfo を作り，キューに格納するとともにジョブ情報をファイルに書き出す．
(4) Schedd がジョブの MatchingInfo を Collector に送信する
(5) Negotiator が Collector にリクエストを送信する
(6) Collector が Negotiator に MatchingInfo を送信する
(7) Negotiator は受け取った MatchingInfo をもとにマッチメイキングを行う
(8) マッチメイキングによりペアになったマシンに通知を送る
(9) Schedd は Negotiator から通知が来ると Shadow を起動する
(10) Shadow は Startd にジョブが実行できるかを問い合わせる
(11) Startd は Shadow からの依頼に返信する
(12) Startd は Shadow からの依頼を受け付ける場合，Starter を起動する
(13) Shadow は Startd から ''OK'' メッセージを受け取るとジョブの起動に必要な情報を Starter に渡す
(14) Starter がジョブを実行する
(15) ジョブの実行が終了すると Shadow に実行結果

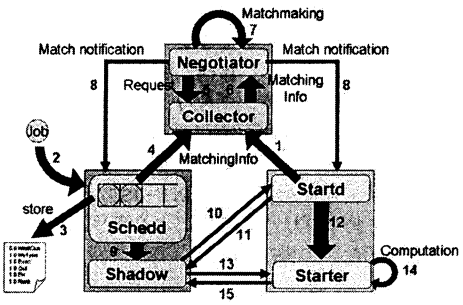

図 5　ジョブ実行の流れ

```
executable      =  sim
arguments       =  100
output          =  out.$(Process)
error           =  err.$(Process)
rank            =  Memory
requirements    =  (Arch == ''INTEL'')
                   && (OpSys == ''LINUX'')
queue 100
```

図 6　サブミットファイルの例

表 1　実験環境

CPU	Athlon MP 1900+ × 2
Memory	768MB DDR
Network	100Base-T
OS	Linux 2.4.18
Java	Sun JDK 1.4.2_03

を返す

本システムは現時点ではまだ機能も限定されているという理由もあるが，約 1 万行程度のコードで非常にコンパクトに構成されており，改変も容易である．

5. 評価実験

5.1 耐故障性

まず本システムの耐故障性について評価を行う．

5.1.1 評価環境

実験には東工大松岡研究室のクラスタ PrestoIII の 10 ノードを使用した．ノードのスペックを表 1 に示す．1 ノードをセントラルマネージャーとし，残り 9 ノードを実行マシンとした．

5.1.2 手　順

長時間の計算が必要なジョブとしてモンテカルロ法を用いて円周率を求めるプログラムを使用した．このプログラムは正方形に乱数で 2×10^9 個の点を生成し，それが正方形に内接する円に含まれるかどうか調べ，生成した点の数と円内に含まれる点の数との比から円の面積を求め，その値から円周率を計算する．表 1 のマシンでは約 1 時間の実行時間を要する．このようなジョブを実行マシンの 1 台から 50 個サブミットした．

長時間の実行中に以下のような外乱を故意に与えることにより，本システムの耐故障性を検証した．

（1）　開始約５０分後実行マシンの１つをリブート
（2）　開始約９０分後１つの実行マシンの Startd プロセスを停止
（3）　開始約１３０分後セントラルマネージャーをリブート
（4）　開始約１７０分後実行マシンの１つをリブート
（5）　開始約２１０分後１つの実行マシンの Startd プロセスを停止
（6）　開始約２７５分後サブミットマシンをリブート

5.1.3 評価結果

図 7 に実行中のジョブの数の推移を示す．約 1 時間ごとにジョブが終了し，次のジョブがスケジューリングされていることがわかる．また，外乱を与えた時間にジョブが停止し，その後，再実行されていることもわかる．サブミットされたジョブは約 6 時間後にすべて終了し，本システムが耐故障性を備えていることが確認された．より大規模で実際のアプリケーションでの実験については今後の課題とする．

5.2 ポータビリティ

本システムは x86 Linux 2.4.19 上で開発を行ったが，ポータビリティの検証のため，SPARC SunOS 5.8 に移行を行った．本システムが稼働するために必要な修正点は Master デーモンでデーモンの稼働状況を確認する際に使用する ps コマンドのオプションとマシンのメモリ情報を確認する際に使用する free コマンドだけである．このようなマシン依存部分が今後増えないという保証はないが，プロセスの確認部分やマシン情報のモニタリング部分というように簡単に切り分けが可能で容易に移行が可能であると考えられる．また，C++で記述された Kickd は，プロセスの実効ユーザ ID を変更するという必要最低限の機能のみを提供するコンパクトなプログラムなのでプラットフォームに依存しておらず，ソースを改変せずにコンパイルをすればよい．以上より，本システムのポータビリティの高さが確認されたが，今後，他のさまざまなプラットフォームにおける検証も必要である．

6. おわりに

6.1 まとめ

本稿では，グリッド環境に適応するため，耐故障性およびポータビリティを備え，研究基盤として利用可能なスケジューリングシステム Jay の設計と実装について述べた．本システムは Condor を規範とし，ジョブ情報のファイルへの書き出しやデーモンを監視する Master デーモンにより耐故障性を提供している．また，ポータビリティを高めるため，Java で実装を行い，それにより生じる問題点についても C++で記述された Kickd デーモンを用意することで解決した．

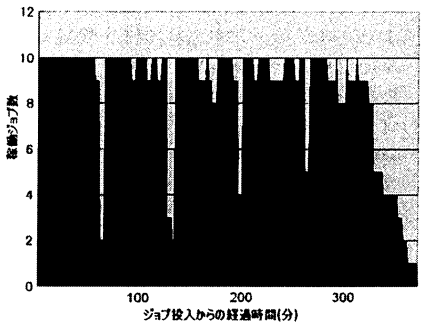

図 7 稼働ジョブ数の推移

本システムの耐故障性を確認するため，長時間の計算を必要とするジョブをサブミットし，故意にマシンをリブートするなどの外乱を与えた．その結果，すべてのジョブが無事終了し，耐故障性が備わっていることを確認した．また，本システムを異なるプラットフォームに移行を行い，ポータビリティの高さについても確認できた．

6.2 今後の課題

今後の課題として以下が挙げられる．

- Condor との比較
  ポータビリティや認証，互換性などを含めて Condor と比較を行う必要がある．
- 大規模環境での実験
  セントラルマネージャーにおけるマッチメイキングや認証などがボトルネックとなることはないか，より大規模なグリッド環境での実験を行い，システムのスケーラビリティを確認する必要がある．
- 複数のチェックポインタのポータブルな導入フレームワーク
  実行マシンがクラッシュした場合などは，それまでの計算結果が失われてしまい，一から計算をやり直さなければならないので，既存のチェックポインタを利用してそれに対応する必要がある．これによりプロセスマイグレーションも可能となる．この際に，実行プラットフォームによってチェックポインタが用意されている場合は，それを直接利用するようなポータビリティの高いフレームワークが必要である．
- プライベートアドレスへの対応
  現在，本システムはプライベートアドレス空間には対応していないので，UPnP を含み，さまざまな手法に対処できる必要がある．
- データインテンシブなアプリケーションへの対応
  入出力データのサイズが大きい場合，実行マシンは計算以外のデータ転送に時間を浪費してしまう．Condor では，Stork[10] というデータスケジューリングシステムを用いて対応している．しかし，Condor と Stork は，独立にスケジューリングを行っており，最適なスケジューリングを行っているとは言えない．そこで本システムでは，複製管理システムと連携し，複製に関する情報も回収した上でマッチメイキングを行うことでより最適なスケジューリングが可能になると考える．

謝辞 本研究は，科学技術振興機構・計算科学技術活用型特定研究開発推進事業（ACT-JST）研究開発課題「コモディティグリッド技術によるテラスケール大規模数理最適化」の援助による．

参 考 文 献

1) Condor Project Homepage, http://www.cs.wisc.edu/condor/.
2) Litzkow, M., Livny, M., and Mutka, M.: Condor - A Hunter of Idle Workstations, Proceedings of 8th International Conference of Distributed Computing Systems, pp. 104-111 (1988).
3) Epema, D., Livny, M., Dantzig, R., Evers, X. and Pruyne, J.: A Worldwide Flock of Condors: Load Sharing among Workstation Clusters, Journal on Future Generations of Computer Systems, Vol. 12 (1996).
4) Livny, M., Raman, R. and Tannenbaum, T.: Mechanisms for High Throughput Computing, SPEEDUP Journal, Vol. 11, No. 1 (1997).
5) Raman, R., Livny, M. and Solomon, M.: Matchmaking: Distributed Resource Management for High Throughput, Proceedings of 7th IEEE International Synposium on High Performance Distributed Computing, July 28-31 (1998).
6) Foster, I. and Kesselman, C.: Globus: A Metacomputing Infrastructure Toolkit, Internatinal Journal of Supercomputer Applications, 11(2):115-128 (1997).
7) Foster, I., Kesselman, C., Tsudik, G. and Tuecke S.: A Security Architecture for Computational Grids, Proceedings of 5th ACM Conference on Computer and Communications Security Conference, pp. 83-92 (1998).
8) Welch, V., Foster, I., Kesselman, C., Mulmo, O., Pearlman, L., Tuecke, S., Gawor, J., Meder, S. and Siebenlist, F.: X.509 Proxy Certificates for Dynamic Delegation, 3rd Annual PKI R&D Workshop (2004).
9) Java Cog Kit 1.1, http://www.globus.org/cog/java/1.1/.
10) Kosar, T. and Livny, M.: Stork: Making Data Placement a First Class Citizen in the Grid, Proceedings of 24th IEEE International Conference on Distributed Computing Systems (ICDCS2004), Tokyo, Japan.