分散ファイル群高度管理のためのファイル関連の発見エンジンの開発

三森祐一郎† 森嶋 厚行†,††

† 筑波大学大学院 図書館情報メディア研究科 〒 305–8550 茨城県つくば市春日 1–2
†† 筑波大学 知的コミュニティ基盤研究センター 〒 305–8550 茨城県つくば市春日 1–2
E-mail: †{mitsu,mori}@slis.tsukuba.ac.jp

あらまし 本論文では，ファイルの内容やファイル操作ログなどからファイル間の多様な関連を発見し，利用するための汎用のファイル関連発見エンジンの開発について述べる．近年，ファイル間の関連を利用した研究が活発に行われているが，これらで開発されたアプリケーションではファイル間の関連発見部分がそれぞれ個別に実装されていた．したがって，毎回大きな実装コストが必要であり，しかも関連発見部分を他のアプリケーションで再利用することも困難であった．これまで，我々はこのような関連発見エンジンを実現するための要素技術を開発してきたが，本論文では，これらの要素技術をどう組み合わせて，分散環境で動作する関連発見エンジン全体を実現するのかを説明し，実装したエンジンの評価を行う．

キーワード ファイル管理，メタデータ

Development of a File Relationship Discovery Engine for Advanced Management of Distributed Files

Yuichiro MITSUMORI† and Atsuyuki MORISHIMA†,††

† Grad. Sch. of Library, Information and Media Studaies, Univ. of Tsukuba, 1–2 Kasuga, Tsukuba, Ibaraki, 305–8550 Japan
†† Research Center for Knowledge Communities, Univ. of Tsukuba, 1–2 Kasuga, Tsukuba, Ibaraki, 305–8550 Japan
E-mail: †{mitsu,mori}@slis.tsukuba.ac.jp

Abstract This paper explains the development of a general-purpose engine for the discovery and utilization of various relationships among files. Recently, there have been many researches dealing with relationships among files, but each researcher implements his own module for the relationship discovery. Therefore, the huge development cost is required each time and it is difficult to reuse the modules. So far, we developed key components required to develop a general-purpose engine for the discovery and utilization of file relationships. This paper mainly explains how the components are combined in order to make our engine work in distributed environments and then evaluates the implemented engine.

Key words File management, Metadata

1. はじめに

近年，計算機による情報処理が日常のものになり，計算機に格納されるデータ量は飛躍的に増大している．また，コンピュータネットワークや各種デバイスの発達により，これらのデータは複数の機械に分散して格納されていることが一般的である．例えば，大学の小規模な研究室のファイルサーバに格納されているファイルであっても数十万あることは珍しくない．また，それらのファイルのコピーや関連ファイルはファイルサーバ内に留まらず，研究室の構成員のノート PC をはじめとして様々な機器に分散して格納されている．その結果，あるプロジェクトに関連するファイルはどこに散らばっているのか，あるファイルの最新バージョンはどれか，バックアップの管理はどうすればよいのか，といったようなことの把握が難しくなり，計算機に格納されているファイル群の管理はますます困難になっている．

このように分散しているファイルの間には，実際には既存のファイルシステムでは明示的に扱われていない，ファイルのコピーや派生，参照関係などといったファイル間の関連や，ユーザとの関連が存在している．このような関連は，ファイル群を

管理する上でとても大きな手がかりとなる．ファイルの複製やバージョン違い，変換による同じコンテンツの別形式などといった関連を利用したい状況は多く存在する．しかし，現在のファイルシステムではこのような関連に関する情報は明示的に扱われていないため，利用することができない．そのため，ファイル間の関連を利用した既存のアプリケーションでは，特定の関連を計算・利用するための機能を個別に実現しているのが現状である．関連を利用したこのようなアプリケーションの開発コストは一般的に大きく，既存の関連を利用したアプリケーションの上に構築するという方法も，それら既存のアプリケーションの多くが関連発見とその応用を一体としているため困難なことが多い．

そこで我々は，ファイル間の関連を容易に利用可能とするための汎用ソフトウェアエンジンの開発に取り組んできた [1]．具体的には，この関連発見エンジンを実現するための各種モジュール (拡張可能な関連発見モジュールや関連発見に利用する情報を収集するためのクライアントソフトウェア，関連を格納するためのグラフデータベース) の提案・開発を行ってきた．特に，論文 [1] では様々な種類の関連を発見できるよう，Candidate Generator と Inspector にという 2 つのファンクションインターフェースを用いたエンジンの拡張フレームワークを提案した．

本論文では，これらの構成要素をどう組み合わせて，複数の計算機に格納されている分散ファイル環境において動作するような汎用の関連発見エンジンを実現するのかを説明する．この実現においては，関連発見の機能が Candidate Generator と Inspector の 2 つのファンクションとして与えられる事を利用して，分散環境でのデータ転送量を削減することを図る．次に，本論文では，次の観点から実装したエンジンの評価を行う．(1) 本エンジンの利用により，ファイル関連を利用するアプリケーションを開発するためのプログラムのコード量をどの程度低減できるか．(2) Candidate Generator と Inspector を利用することにより，分散計算機環境で本エンジンを実行するときに必要な計算機間のデータ転送量を実際にどの程度削減できるか．

本論文の構成は次の通りである．2 章では，関連研究について紹介する．3 章では，現在我々が取り組んでいるコミュニティ情報空間ガバナンスプロジェクトについて説明する．4 章で，関連発見エンジンの概要と設計について説明する．5 章で，エンジンの評価結果を示す．6 章は，まとめと今後の課題である．

2. 関 連 研 究

ファイル間の関連を発見し，応用しようという試みは多くあり，それらのほとんどは情報検索の文脈で行われている．

渡部らは，ファイルサーバのアクセスログからファイル間関連度を検索に利用する FRIDAL(File Retrieval by Inter-file relationship Derived from Access Log) [2] の研究を行っている．FRIDAL では，プロセスで使用されているファイルの共起時間の累計，共起回数，間隔等を利用して，共起検索時にファイル間に重みづけをすることで，キーワードを含まない関連ファイルの検索の実現を可能としている．また，Craig らは read や write などのシステムコールの履歴を検索に利用する Connections [3] の研究を行っている．

図 1　コミュニティ情報空間 (CIS) の例

我々のプロジェクトの特徴は，このようなファイル間の関連を統合的に発見・利用可能なエンジンを提供することで，情報検索を含めた様々な応用アプリケーションの開発を容易にすることである．本エンジンでは，既に提案された様々な関連発見アルゴリズムも組み込み可能な，汎用的な基盤を目指している．

3. コミュニティ情報空間ガバナンスプロジェクト

我々は，複数の構成員と計算機を含むような知的活動を行うコミュニティの情報空間を対象とした管理システム InfoSpace Governor の研究を進めている [1] [4] [5] [6] [7] [8]．本システムでは，既存のファイルシステムでは明示的に扱うことのできない，ファイルのコピーや派生，参照関係などといったファイル間の関連やユーザとの関係を収集・利用することで，情報空間の効果的な管理を目指している．

本研究の関連発見エンジンは，このプロジェクトに属し，中核として利用されている．本エンジンが目指すのは汎用の関連発見エンジンであり，特定のシステムを対象としたものではないが，利用例の 1 つとしてここで簡単に紹介する．エンジンについての詳細は 4 章で述べる．

3.1 InfoSpace Governor の特徴

InfoSpace Governor には，大きく分けて 2 つの特徴がある．

第一の特徴は，個々の計算機のファイルシステムではなく，**コミュニティ情報空間** (Community Information Spaces, CIS) と呼ぶ複数の構成員とファイルシステムを含む空間を管理の範囲としていることである．図 1 はある大学の研究室における CIS の例である．一般に，CIS には複数の計算機が含まれており，それぞれが情報を分散して管理している．例えば，この研究室で行われているプロジェクトに関連する情報は，ファイルサーバと参加メンバの PC に分散されて管理されている．それらの PC に管理されている情報は，お互い関連しているもののシステムレベルでは明示的には関連づけられていない．例えば，あるプロジェクトに関連するファイルの最新バージョンはどれか，といった情報や，そのファイルが参照するファイルはどれか，といった情報をシステムはもっていない．したがって，これらの問にシステムが答えることは出来ない．ネットワークを通じてこれらの情報源が物理的には接続されているにもかかわらず，先に述べたような関連のレベルでは実は接続されていないのである．

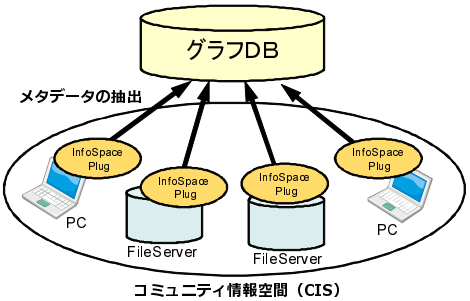

図 2　InfoSpace Governor のアーキテクチャ

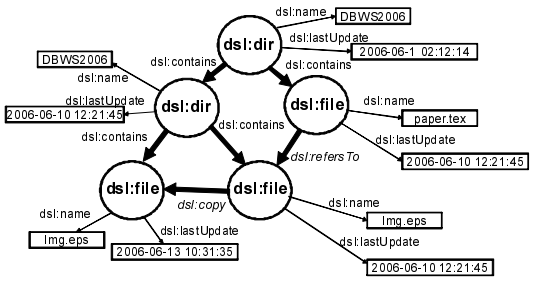

図 3　情報空間メタデータの例

第二の特徴は，これらの関連を明示的に表すメタデータを保持することにより，情報空間の管理を行う事である．具体的には，各クライアント PC 上で InfoSpace Plug と呼ばれるソフトウェアモジュールを用いて，今説明したような情報資源間の関連を，グラフ DB に明示的に保持する (図 2)．図 3 はその内容の一部を示したものである．グラフ DB 中では，ファイル，ディレクトリ，プロセス，ユーザといった要素がグラフノードとして表現しており，ファイル間だけでなく，ユーザとファイルの関連や，プロセスとユーザの関連といった情報も利用することができるが，本稿ではファイル間の関連に絞って説明する．

3.2　ファイル間の関連

InfoSpace Governor が標準的に扱うファイル間の関連のうち，主要なものを次に説明する．これらの関連のいくつかは図 3 に現れている．

Copy: 同じコンテンツを持つファイル間に成立する関連．

Move: 削除されたファイルを表す特別なノード「ファントム」と，移動先のファイルを表すノード間に成立する関連．

RefersTo: ファイルの参照関係．例えば，tex ファイルや html ファイルから別のファイルを参照している場合に成立する．

Version: バージョン違いのファイルの間の関連．ファイルのコピー後に編集した場合の．コピー元と編集後のファイルの間に成立する．

Derive: あるファイルとそのファイルから生成されたファイル間の関連．例えば，tex ファイルとそのコンパイル結果の dvi ファイルとの間に成立する．Version の関連を持つファイルはデータ形式が同じだが，Drive は変換を伴う結果なのでデータ形式が異なる．

後述するように，本エンジンは外部プログラムに対する API を持つが，そこでは，ファイル関連のやりとりを行う際に次のデータクラスを利用する．

```
class Triple{
  String resource,  String subject,  String property;
}
```

ここで，resource と subject は各々に対応するファイルの識別名が保持され，property にはその 2 つの関係を表すラベル名が入る．

3.3　アプリケーション例

ファイル関連発見エンジンを用いた InfoSpace Govenor のアプリケーション例をいくつか紹介する．より詳細な例は [7] にある．(1) 関連ファイル，コピーファイル，最新バージョンなどの発見：格納された関連情報を用いて，通常の検索やブラウジングでは発見できない関連ファイルの発見支援を行う．(2) 問合せに結びついた仮想フォルダの作成: 例えば，ある特定のファイルの更新履歴を追っていき，常に最新バージョンのファイルにアクセス可能な仮想フォルダを実現する．(3) バックアップの支援: ファイルのバックアップはディレクトリ単位で行われることが通常であるが，あるタスクやアプリケーションに関係するファイルは必ずしも特定のディレクトリの下にあるとは限らない．これらのファイルを漏れなくバックアップする．(4) ファイルの関連に基づく警告：例えば，あるファイルから参照されている別のファイルを削除しようとしたときに警告を出す．(5) ファイル整理支援：あるファイルをこの PC から消しても問題ないか (例えばファイルサーバに同じものがあるなど) の決断を支援する．

4.　関連発見エンジンの構成要素

本章では，これまでに開発してきた関連発見エンジンの構成要素（図 5）について説明する．

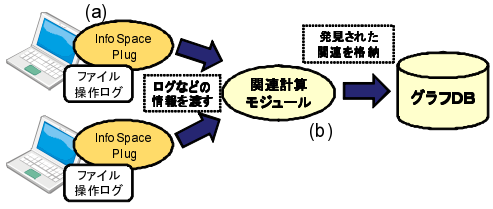

図 5　関連発見エンジンの構成要素

4.1　InfoSpace Plug

関連を発見するためには，そのための情報が必要となる．本エンジンでは，CIS 中の各クライアント PC に，InfoSpace Plug (図 5 (a)) というソフトウェアモジュールを組み込む．このモジュールは次の 3 つの機能を持つ．(1) 操作ログ抽出機能: 利用者の操作ログを記録する．(2) ファイルディレクトリクローリング機能: 各ファイルシステムのコンテンツにアクセスし，必要な情報を入手する．(3) 収集した情報の送信機能: (1)，(2) で得られた情報を関連計算モジュールへと送信する (図 5)．ただし，これらの情報の収集はクライアント PC を使用するユーザの許可した範囲のファイル群に対してのみ行われる．

抽象クラス名	メソッド名	説明
CandidateGenerator	Triple[] generate (String[] nodes)	ノード名の集合 nodes から，特定の関連を持つノードの組み合わせを RDF トリプルの集合として返すメソッド.
Inspector	bool inspect (Triple triple)	nodeA が nodeB に label の関係を持つかどうかを検証し，持っていれば True を返すメソッド.

図 4　実装される抽象クラス

4.2　関連計算モジュール

これは本エンジンの中心となるモジュールである（図 5 (b)）．3.2 節で説明した種類の関連をはじめ，様々な種類の関連の発見・利用をサポートできるよう，拡張可能なインターフェースを用意している．特徴は，関連発見のためのアルゴリズムを追加するために Candidate Generator と Inspector という二つのファンクションインターフェースを用意していることである．利用者は，これらのインターフェースを満たすファンクションをエンジンに登録することによって，新たな種類の関連の発見機能を追加することができる．また，次に説明するように，これらの 2 種類のファンクションを用意することにより，効率良い関連発見処理を実現する．

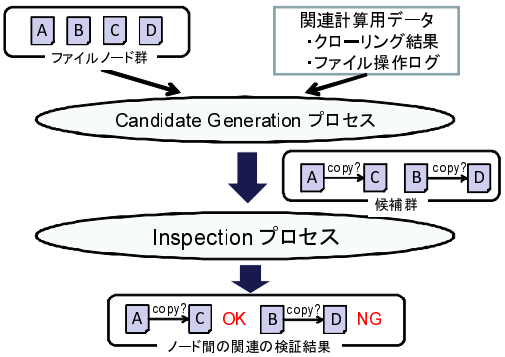

図 6　関連発見処理の流れ

関連発見のプロセス．図 6 は，どのように関連の発見を行うかの流れを示したものである．次に，それぞれのプロセスを説明する．

CandidateGeneration プロセス．InfoSpace Plug によって収集した情報から，CIS に存在する関連の候補を発見するためのプロセス．厳密な検証は行わず，比較的低いコストで候補の絞り込みを行う．このプロセスは，エンジンに登録された，CandidateGenerator インターフェースを持つファンクションを実行することにより実行される．CandidateGenerator の実装パターンとしては，主にログのパターンから発見するものと，ディレクトリの中をクローリングして候補を見つけるものが考えられる．一つの種類の関連に関して複数の CandidateGenerator を登録することが許されるため，一つの CandidateGenerator で必ずしももれなく候補を列挙する必要はない．

Inspection プロセス．CandidateGeneration プロセスで生成された関連の候補を実際に検証するプロセス．検証を通った候補は，実際にメタデータ DB に反映される．厳密な検証が難しい場合は，正しい可能性が高いかを検証する．このプロセスは，エンジンに登録された，Inspector インターフェースを持つファンクションを実行することにより実行される．

このように，最初のプロセスで簡易なチェックを行うことで関連候補を絞り込み，その後，次のプロセスで厳密な検証を行うことで，関連発見にかかる計算コストの削減を図っている．

実際には，CandidaterGenerator と Inspector インターフェースは図 4 の抽象クラスとそのメソッドとして提供される．追加したい関連の種類ごとに，このこれらの抽象クラスを継承するクラスを定義し，そのオブジェクトをエンジンに登録する．

5.　関連発見エンジンの全体設計

本章では，分散環境において動作する関連発見エンジンの全体設計について説明する．

5.1　もっとも単純な設計

関連計算のプロセスを実現するもっとも単純な設計は，図 7 に示す関連発見モジュールの discovery メソッドをサーバで定期的 (例えば 1 時間に一回) に実行することである (図 8)．その際，InfoSpace Plug は，更新された全てのファイルと前回の Discovery メソッド呼び出し以降のログの全てをサーバに転送する．InfoSpaceStorage は，InfoSpace Plug から転送されたファイルやログを一時的に保存するための記憶領域である．

図 7 のコードを説明する．RelationshipDiscovery クラスのメンバ変数 gens と inspectors (3-5 行目) には，それぞれ事前に登録された，抽象クラス CandidateGenerator と Inspector の継承クラスのインスタンスが格納されているものとする．CandidateGeneration クラスの 12-20 行目で，登録されたすべての candidateGenerator を実行し，ラベル毎に候補を保存する．その後，22-28 行目で保存した候補群を，inspect メソッド (32-43 行目) によって検証する．inspect メソッドでは，34 行目において関連に対応した Inspector を取得し，37-42 行目で検証を行う．検証を通過した関連はデータベースへと格納される．

5.2　解決すべき課題とそれを考慮した設計

もっとも単純な設計における二つの問題と，それに対する対応を説明する．

(1) 関連発見に必要な処理のタイミング．最も単純な設計では，定期的に Discovery メソッドが呼び出されるタイミングで，関連に関する全ての CandidateGeneration プロセスと Inspector プロセスが起動する．しかし，これは必ずしも望ましくない．例えば，ログを用いた CandidateGenerator は短期間毎に実行することが望ましいし，クローリングを行う CandidateGenerator は実行コストが高いため，実行間隔を長く取ることが適切な場合もある．したがって，これらの一律に同じタイミングで実行することは望ましくない．したがって，CandidateGenerator 毎にタイミングを指定できる用にしたい．

メソッド名	説明
Triple[] storeFile (String path, String[][] metadata, byte[] content)	ファイルパス path のファイルを，メタデータ metadata，内容 content を持つファイルとして，InfoSpace Storage に格納する．
boolean storeLog (String domain, byte[] log)	ドメイン domain の新規ログデータを，InfoSpace Storage に格納する．
String[] getRequest(String domain)	ドメイン名 domain である計算機に、サーバが要求するファイルパスのリストを取得する．要求がない場合，null を返す．

図 9　サーバが提供するメソッド

```
01 /* 関連計算モジュールクラス */
02 class RelationshipDiscovery {
03   CandidateGenerator[] gens;//登録された CandidateGenerator 群
04   //キーを関連名，値を対応する Inspector とするハッシュ
05   Hashtable inspectors;
06
07   // 関連を発見する
08   void discovery(){
09     //キーが関連名，値が対応する関連の候補リストのハッシュ
10     Hashtable   candidates;
11
12     // 登録された CandidateGenerator から候補生成
13     for (i=0; i < gens.size(); i++){
14       // 候補を生成
15       Triple[] result = gens[i].generate();
16
17       // ラベル毎にリストに保存
18       List l = candidates.getValue(gens[i].LABEL);
19       l.addAll(result)
20     }
21
22     // ラベル毎に候補を Inspection で検証する．
23     Set<String> set = candidates.keySet();
24     foreach(k ∈ candidates.keys){
25       List cs = candidates.get(k);
26       // ラベル k の候補群を Inspection に渡す．
27       ins.inspect(cs.toArray(), k);
28     }
29   }
30
31   //受け取った label の関連候補 candidates を検証するメソッド
32   void inspect(Triple[] candidates, String label){
33     // 関連 label の Inspector を取得
34     Inspector ins = inspectors.getValue(label);
35
36     // 候補の検証．通った関連を DB に登録
37     for (i=0; candidates.size(); i++){
38       if (ins.inspect(candidates[i])){
39         /*** データベースに candidates[i] の関連を格納 ***/
40         …
41       }
42     }
43   }
44 }
```

図 7　関連発見エンジンの動作

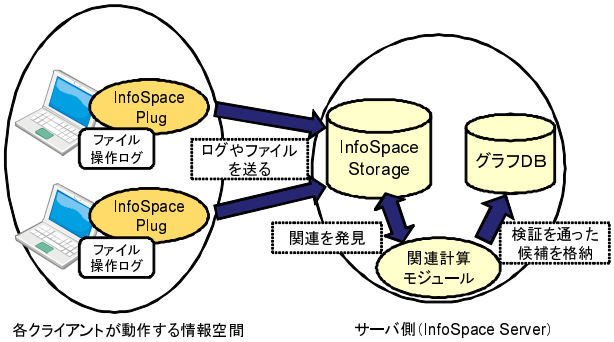

図 8　ファイル関連発見エンジンのアーキテクチャ

(2) Inspection プロセスのためのファイルの転送．

最も単純な設計では，Discovery メソッドを呼び出すタイミングで，更新のあったファイルを全てサーバに転送する．しかし，これらが必ずしもすべて Inspection プロセスで参照されるとは限らない．したがって，無駄な転送になる可能性がある．したがって，CandidateGenerator をまず先に実行し，その後で必要なファイルだけをサーバに転送するようにしたい．

以上の問題を考慮した全体設計を説明する．

次に，これらを考慮した全体設計における，サーバと InfoSpace Plug の動作についてそれぞれ説明する．

5.2.1　サーバの動作

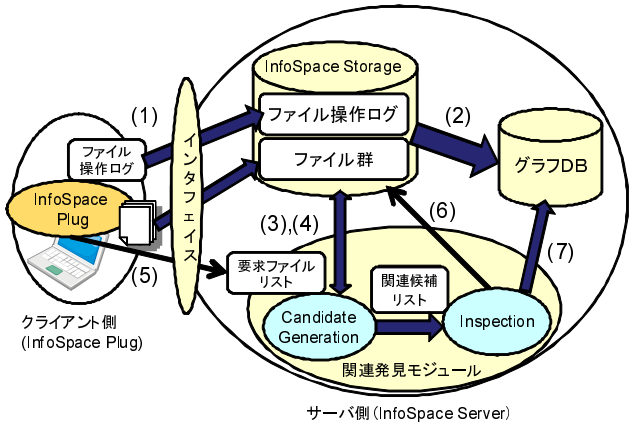

図 10　サーバの動作

サーバの動作の主な流れは図 10 の通りである．(1) InfoSpace Plug からの操作ログを InfoSpace Storage に定期的に格納．(2) InfoSpace Storage にログが格納されると，そこからファイルのメタデータをグラフ DB に格納．(3) CandidateGeneration プロセスが，指定された周期で各 CandidateGenerator を実行．(4) 3 で生成された関連候補から，各クライアント PC に要求するファイルのリストを作成．(5) 4 の要求ファイルリストを元に InfoSpace Plug が送ってきたファイルを InfoSpace Storage に格納．(6) Inspection プロセスは，検証に必要なファイルが揃った関連候補から検証する．(7) 6 で検証を通った関連をグラフ DB に格納．

サーバの疑似コードを図 11 に示す．サーバは，図 9 のメソッドを提供している (7-25 行目)．InfoSpace Plug はこれらを呼び出すことで，InfoSpace Storage へのログやファイルの格納や，ファイルの要求の確認を行う．また，CanadidateGeneration プロセスと Inspection プロセスを実行している (27-34 行目)．

CandidateGeneration と Inspection の疑似コードは図 12 である．CandidateGeneration クラスでは，setSchedule メソッド (7-10 行目) によって各 CandidateGenerator の実行をスケジューリングする．スケジューリングされた CandidateGenerator は，19-30 行目の実行メソッドにおいて定期的に実行される．CandidateGeneration で関連候補が生成されると，その関連候補を Inspection で検証するのに必要なファイルの，計

```
01 /* サーバのインタフェース部分 */
02 class InfoSpaceServer {
03   InfoSpaceStorage storage;
04   GraphDbms gdbms;  //グラフ DBMS
05   File requests; //要求ファイルリスト
06
07   //// InfoSpace Plug から呼び出されるメソッド群
08   // クライアントのファイルを格納するメソッド
09   Triple[] storeFile(String path, String[][] metadata,
                                              byte[] content){
10     // ストレージにファイルを格納
11     storage.storeFile(filepath, metadata, content);
12   }
13   // クライアントのログを格納するメソッド
14   boolean storeLog(String domain, byte[] log) {
15     storage.storeLog(domain, log);  //ストレージにログを格納
16   }
17   // クライアントに要求するファイルがあれば，そのリストを返すメソッド
18   String[] getRequest(String domain){
19     //要求ファイルのうち，domain を含むものを返す.
20     List rq;
21     while (String r = read(requests))
22       if (r.contains(domain)) rq.add(r);
23
24     return rq.toArray();
25   }
26
27   // サーバの処理
28   static void main(String args){
29     //CandidateGeneration と Inspection を実行
30     CandidateGeneration gen = new CandidateGeneration();
31     gen.run();
32     Inspection ins = new Inspection();
33     ins.run();
34   }
35 }
```

図 11　サーバ側の疑似コード

算機のドメインとファイルパスを要求リストに追加する (28 行目)．前述したように，InfoSpace Plug はこの要求リストを addRequest メソッドを呼び出すことで定期的に取得し，ファイルを InfoSpace Storage へと格納する．Inspection の実行メソッドでは，現在存在する関連候補の中で InfoSpace Storage にファイルが格納されているものすべてを返す hasCandidates メソッドを呼び出して検証可能な候補群を取得し (46 行目)，それを inspect メソッドによって検証する (47-50 行目)．inspect メソッドでは関連候補を対応する Inspector によって検証し，通過した関連をグラフ DB に格納している．

5.2.2　InfoSpace Plug の動作

4.1 節でも述べたように，InfoSpace Plug はファイル操作のログを記録し，定期的にサーバへと送信する機能を持っているが，これに次の機能を追加する．すなわち，サーバの動作で説明したように，ログを元に関連候補がサーバ側で生成されると，その関連候補の検証に必要なファイルの一覧がサーバの要求リストに追加されるので，InfoSpace Plug はこのリストを定期的に問い合わせて，要求されたファイルが存在すればそのファイルを送る．これらの処理は，サーバ側が提供する図 9 のメソッドを利用して行う．InfoSpace Plug の関連発見部分の疑似コードを図 13 に示す．InfoSpaceClient クラスでは，図 9 のメソッドをサーバオブジェクト obj (4 行目) から取得し，ログの送信 (32-35 行目) とサーバの要求ファイルの送信 (6-20 行) を定期的に実行している．

6. 評　　価

6.1 概　　要

設計したエンジンのプロトタイプを実装し，その評価を行った．評価するのは，以下の 2 点である．(1) 本エンジンにより，

```
01 /* CandidateGeneration クラス */
02 class CandidateGeneration extens Thread{
03   CandidateGenerator[] gens;
04   File requests; //要求ファイルリスト
05
06   // makeCandidate をラベル毎にスケジュール設定するメソッド
07   private void SetSchedule() {
08     /*** 各 CandidateGenerator を実行周期毎にスケジューリング ***/
09     …
10   }
11
12   // 関連候補の検証に必要なファイルを要求リストに追加
13   void addRequest(Triple[] candidates){
14     // 要求リストに必要なファイルを追加
15     for (int i=0; i<candidates.length; i++)
16       write(candidates[i] to requests);
17   }
18
19   //CandidateGeneration 実行メソッド
20   void run(){
21     Triple[] candidates; //生成する関連候補群
22     //関連の候補発見を行う
23     while(true){
24       /*** スケジュールに従って指定の CandidateGenerator を実行 ***/
25       …
26
27       //要求ファイルリストに追加
28       addRequest(candidates);
29     }
30   }
31 }
32 /*-----------------------------------------------------------*/
33 /* Inspection クラス */
34 class Inspection {
35   Hashtable inspectors;
36
37   // 受け取った候補を検証するメソッド
38   void inspect(Triple candidate){
39     /*** 図 7 の inspect の処理を単一のトリプルで行う ***/
40     …
41   }
42
43   //Inspection 実行メソッド
44   void run() {
45     while(true){
46       Triple[] candidates = hasCandidates();
47       if (candidates != null)
48         foreach(c ∈ candidates){
49           inspect(c);
50         }
51       }
52     }
53   }
54 }
```

図 12　CandidateGeneration と Inspection の疑似コード

ファイル間の関連を利用したアプリケーションの開発に必要なコード量をどれだけ低減できるか．(2) エンジン設計におけるファイル転送の遅延によって，必要なファイル転送量は実際にどれだけ削減されるか．

6.2 関連を発見するために必要なコード記述量

本エンジンの利用により，分散ファイル環境における関連発見を利用したアプリケーションの開発コストがどの程度低減できるかを調べるために，関連発見の実装に必要なコード量の調査を行った．

対象としたのは，コピー関連の発見を行うために必要となるコードの量である．本エンジンを利用してアプリケーションを開発した場合に必要となるコード量は，コピーに関する CandidateGenerator クラスと Inspector クラスのコード量とした．CandidateGenerator はログベースのものである．本エンジンを利用しなかった場合のコード量は，次の 4 つの行数の合計として計算した．(1) コピーに関する CandidateGenerator クラスと Inspector クラスの行数．(2) 1 のコードで利用されているライブラリの行数．(3) 2 のライブラリで利用する情報を収集する InfoSpace Plug の行数．(4) 2 の関数で利用するファイルを格納する InfoSpace Storage の行数．

```
01 /* InfoSpace Plug の関連発見関連部分 */
02 class InfoSpaceClient extends Thread{
03   // サーバのメソッドを呼ぶためのオブジェクトが入っている
04   InfoSpaceServer obj;
05
06   // ファイルの要求があるまで待つ
07   public void run (String label){
08     // サーバがファイルを要求するまで待機
09     while(true){
10       String[] request = obj.getRequest();
11       if (request != null){
12         //要求されたファイルをサーバに送信
13         for (int i=0; i<request.length; i++) {
14           String path = request[i];
15           if (path のファイルが存在するなら)
16             sendFile(path, getMetadata(path), getContent(path));
17           }
18         }
19       }
20     }
21
22     static void main(String[] args) {
23       //要求待ち受けプロセスの開始
24       InfoSpaceClient client = new InfoSpaceClient(LOG_NAME);
25       client.run();
26
27       //ロギングプロセスの開始
28       InfoSpaceLogger logger = new InfoSpaceLogger();
29       logger.run();
30
31       // 定期的に新規ログを送信
32       while (true){
33         obj.storeLog(クライアントのドメイン，前回からのログの差分);
34         sleep(300000);
35       }
36     }
37   }
```

図 13　InfoSpace Plug の関連発見部分の疑似コード

これらの行数は関連発見に必要なコード行数だけを数えており，InfoSpace Plug のインタフェース部分やグラフ DB などの関連発見と関係のないコードは数えないため，比較として適切であると考える (すなわち，エンジンを利用しない場合に開発者が必ず書かなければならないコードだけを数えている)．
図 14 はその結果である．

本ミドルウェアの利用	行数（行）
なし	4257
あり	282

図 14　分散ファイル環境でのコピー関連発見実装に必要なコード行数

図 14 から，本エンジンを利用しなかった場合を比較して，利用した場合には関連発見のために記述するコード量が大きく削減できていることが分かる．したがって，本エンジンの利用により，関連を利用したアプリケーションの開発コストを削減できると考えられる．

6.3　エンジンのデータ転送量

本エンジンの設計では，データ転送前に CandidateGeneration プロセスを実行して，サーバに転送するファイルの絞り込みを行っている．これが実際の環境においてどの程度，ファイルの転送量の削減につながるかを調べた．

実験方法．典型的なファイル操作のシナリオを実行し，更新ファイルを全てサーバに転送した場合 (A) と，CandidateGenerator が出力した候補に関係するファイルだけを転送した場合 (B)，のデータ転送量の比較を行った．ここでのデータ転送量とは，コピー関連発見のためのログベースの CandidateGenerator と Inspector を対象とし，この処理に必要なデータ転送量の事である．図 16 は，実験に使用した計算機の詳細である．

tex ファイルサイズ	34.6 KByte
画像ファイル総サイズ	8.332 MByte

図 15　実験対象データ

クライアント	CPU: Core2Duo 2.0 GHz　メモリ: 2.5 GB OS: Windows Vista
サーバ	CPU: Pentium4 3.4 GHz　メモリ: 1.99 GB OS: Windows XP

図 16　実験環境

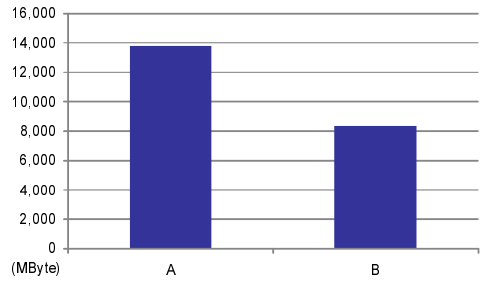

図 17　実験結果

実験で使用したシナリオは，tex ファイルの編集である．次のような手順の操作を想定した．(1) ある tex ファイルとそこで使う画像ファイルが入ったフォルダを別の場所にコピー．(2) コピーしたフォルダの tex ファイルの内容を編集．(3) 編集したファイルから dvi ファイルを同じフォルダに生成．(4)dvi から pdf ファイルを同じフォルダに出力．

対象となるフォルダのデータは，図 15 の通りである．

実験結果．図 17 に結果を示す．今回のシナリオでは，本エンジンの設計にしたがって CandidateGenerator で転送するファイルを絞った場合のファイル転送量は，すべてのファイルを転送した場合と比べて転送量を約 43 %削減されていた．ただし，この結果は CandidateGenerator の性能に依存することに注意しなければならない．なぜなら，候補をあまり絞り込めない CandidateGenerator が存在した場合はほとんどのファイルを転送することになってしまうためである．したがって，多くの関連候補を生成する可能性があるクローリングベースの CandidateGenerator の設計には十分注意する必要がある．

7.　まとめと今後の課題

本稿では，ファイル間の関連抽出のための汎用的な関連発見エンジンの開発について述べた．本フレームワークは，既存のファイルシステムでは明示的に扱うことのできない，ファイル間の関連の発見・利用を行うための機能を提供することで，これらを利用したアプリケーションの開発を容易とすることを目指している．本論文では，実装したエンジンの評価として，ソフトウェア開発コストの削減と実利用におけるパフォーマンスについての評価を行った．今後の課題としては，より大規模な実用実験と，その知見に基づくエンジンの改良がある．

謝辞　ゼミなどでコメントいただきました筑波大学大学院図書館情報メディア研究科の杉本重雄教授，阪口哲男准教授，永森光晴講師に感謝いたします．本研究の一部は科学研究費補助

金若手研究 (B)(#20800076) による．

文　　献

[1] 三森祐一郎，森嶋厚行：分散ファイル群高度管理を目的としたファイル関連の発見支援機構，情報処理学会研究報告 vol.2008, No.146, (2008-DBS-146) , pp.319-324. 2008 年 9 月.

[2] 渡部徹太郎，小林隆志，横田治夫：キーワード非含有ファイルを検索可能とするファイル間関連度を用いた検索手法の評価，電子情報通信学会 第 19 回データ工学ワークショップ (DEWS2008) 論文集 E10-6, Mar 9-11, 2008.

[3] Soules, C. A. N., Ganger, G. R. : Connections: using context to enhance file search. In Proceedings of the 20th ACM Symposium on Operating Systems Principles, pp.119-132. 2005.

[4] 石川憲一，森嶋厚行, 田島敬史： メタデータ DB のための到達ノード問合せと更新処理の効率化，電子情報通信学会第 18 回データ工学ワークショップ (DEWS2007), 2007 年 3 月.

[5] 望月祥司, 児玉麻莉子, 石川憲一，森嶋厚行, 田島敬史： 大規模ディレクトリ空間視覚化のための論理構造抽出, 電子情報通信学会第 18 回データ工学ワークショップ (DEWS2007), 2007 年 3 月.

[6] 石川憲一，森嶋厚行, 田島敬史：大規模ドキュメント空間管理のための意味ファイルシステムの構築, 電子情報通信学会技術研究報告, Vol.106, No.150, DE2006-115, pp. 139-144, 2006 年 7 月.

[7] 三森祐一郎，森嶋厚行：メタデータを利用した高度ファイル操作のためのミドルウェアの提案，情報処理学会研究報告 Vol.2007, No.65 (2007-DBS-143), pp. 180-194. 電子情報通信学会技術研究報告 Vol.107, No.131, DE2007-53, pp. 189-194, 2007 年 7 月.

[8] 三森祐一郎，森嶋厚行：コミュニティ情報空間管理のための制約記述と処理手法の提案，電子情報通信学会第 19 回データ工学ワークショップ (DEWS2008), 2008 年 3 月.

[9] W3C. Resource Description Framework (RDF). http://www. w3.org/RDF/.