Towards Self-organizing Service Communities

Hye-young Paik[1], Boualem Benatallah[2] and Farouk Toumani[3]

[1] School of Information Systems, Queensland University of Technology, Australia, h.paik@qut.edu.au
[2] School of Computer Science and Engineering, University of New South Wales, Australia, boualem@cse.unsw.edu.au
[3] LIMOS, ISIMA, University Blaise Pascal, France, ftoumani@isima.fr

***Abstract*— This paper discusses a framework in which catalog service communities are built, linked for interaction, and constantly monitored and adapted over time. A catalog service community (represented as a peer node in a P2P network) in our system can be viewed as domain specific data integration mediators representing the domain knowledge and the registry information. The query routing among communities is performed to identify a set of data sources that are relevant to answering a given query. The system monitors the interactions between the communities to discover patterns that may lead to restructuring of the network (eg., irrelevant peers removed, new relationships created, etc).**

***Index Terms*— e-Catalogs, Catalog portals, P2P, Self-adaptivity**

I. Introduction

CATALOG[1] portals, such as Amazon.com or Expedia.com, are becoming a prominent feature of the World Wide Web. However, the technology to organize, search, integrate and evolve these portals has not kept pace with the rapid growth of the available information space. Currently, users need to access a number of information portals separately, using search engines, in order to collect relevant information. For example, in a computer market scenario, users need access to several Web portals (ones that are specialized in computer retail) separately, manually filter and organize the search results to obtain the complete picture of the products on offer. Clearly, to effectively realize the potential of Web-based information access, there is a need for facilitating the *integrated* access to *relevant* Web information portals.

Another aspect to be noted in current portals is that their centralized and static categorization often results in a rigid structure that is difficult to change or maintain. The categorization is designed to reflect the system designer's point of view, who has a priori expectations for how catalogs will be explored by customers. It is necessary to take into consideration how the catalogs are actually utilized by the customers as well as the portal itself. This may continuously minimize the gap between expectations of the system designer and customers. For example, if observations show that a catalog on `Laptops` and a catalog on `Handheld Computers` are visited together most of the time, the system designer may decide to put them in the same category rather than in separate categories.

In the project WS-CatalogNet [1], [2], we proposed a platform through which catalog portals are built and interact with one another. This enabled a potentially large number of catalogs to act as one catalog to serve customers' queries. The approach is based on a hybrid of P2P data management paradigm and Web services architecture. It uses the notion of *catalog service communities*[2] where catalogs catering for similar customer needs are grouped together [3], [4] and form a single *community* (ie., a peer node). Catalogs *register* themselves to a catalog service community (hereafter community) as *members* and the communities themselves interact with other peers. Each peer (ie., the community) in WS-CatalogNet can be viewed as a domain specific data integration mediator which holds metadata and registry information about its members (ie., catalogs). Queries are processed by the members of a peer, but routing of the queries is a responsibility of the peers. It should be noted that the purpose of the query routing in our system is to identify a set of members that, when put together, can satisfy *all* constraints required by a query. Hence, a routing takes place before the actual query process. Once a set of members (not necessarily from the same peer) are identified, queries are sent to each member in the set for processing. The results are combined by the original community.

WS-CatalogNet work, so far, has focused on creating, linking and querying the community network. In this paper, we investigate the ability to evolve from the initial design of the system. The evolution may involve capabilities like "discovering" members (ie., catalogs) or other communities (ie., peers), splitting a community, re-forming the relationships, etc. As communities are developed and relationships are formed, the interactions between communities can be monitored. We analyze the monitoring result to draw *interaction patterns* that may trigger restructuring of the network. We also offer a set of operations for restructuring community networks.

The paper proceeds as follows. Section II looks at the related work. In section III, we present the overview of WS-CatalogNet. Section IV discusses the monitoring/adapting techniques for restructuring of the community network. Sections V and VI describe the implementation and evaluation results, followed by concluding remarks in VII.

II. Related Work

The catalog portal framework proposed in WS-CatalogNet combines the Web Services architecture and P2P data management paradigm. Instead of developing a single global system or registry, we explore the possibility of using "P2P"-ed Web

[1]We use catalogs and e-catalogs interchangeably.

[2]The term service community comes from the fact that the community is implemented as a Web service.

services as a flexible and scalable framework for sharing data across a large number of online sources.

Early P2P systems [5], [6] are based on the common objective; retrieval of a document or a file. [7] offers a comprehensive survey of such systems. Our work is more in-line with current efforts that leverage database paradigm with P2P, binding a database capability with a peer [8], [9]. In this paradigm, it is assumed that each peer has data to share which resides in a database. One of the relevant issues is to resolve the semantic interoperability between peers. This involves a technique that allows a peer to express how its data relates to the data in other peers (often known as a schema mapping). Using the mapping information, a query posed on one peer can be re-formulated to a query on another peer. In many ways, the work in multi-database systems (eg., MOMIS [10], TSIMMIS [11], etc.) and distributed query processing [12] still apply in this paradigm. In these systems, however, a user issues a query based on a single global schema, and the system maps the query to sub-queries that can be understood by the underlying data sources. Typically, the data sources are assumed to be relational tables only. Also, the global schema is created and maintained centrally by a database designer. These approaches are not directly applicable to the environment in which WS-CatalogNet operates. The data sources can be in any format, the number of participating data sources may be large, and any data source may enter or leave the system frequently. Therefore, this paradigm calls for a more flexible and scalable approach.

PeerDB project [13] uses a relational model to describe the schema of a peer. It employs an Information Retrieval (IR) approach for the query routing to avoid the explicit specifications of mapping between peer schemas. The query reformulation is assisted by an agent automatically identifying the mapping based on a built-in strategy. On the other hand, Piazza [14] considers using a global schema and schema mediation techniques in P2P environments. Each peer schema is described using a relational model. It proposes a language for specifying mappings among peers and a query reformulation algorithm for translating the global schema to a local peer schema. Hyperion [15] proposes use of mapping tables which map data *values* (rather than the schema) between two peers. Such mapping tables are constructed by domain experts. These approaches assume no knowledge about the underlying schema of the peers when forwarding a query, but the efforts required to identify the correct mappings are left with the users.

The systems mentioned above are based on the pure P2P architecture where individual data sources are connected directly as peers, whereas in WS-CatalogNet, each peer is a data integration mediator (similar to the concept of super-peer in [16], [17]) holding a registry/mapping information for the individual data sources. This helps in organizing the available information space into manageable, meaningful spaces. Also, the burden of query reformulation/translation is distributed among the peers, rather than the individual data sources. Another important distinction in our work is that WS-CatalogNet does not make any assumption about the nature of the local data source whereas other works we mentioned focus on either relational databases or XML/RDF resources only.

To the best of our knowledge, little work has been done in addressing the issue of restructuring catalog community networks, as tackled in this paper. There have been some related work in similar areas (tuning P2P networks and ontology restructuring). For example, [18] proposes a self-monitoring P2P network that *tunes* itself according to the workload of a peer. The tuning is based on break() and connect() operations. The network concerned is set on the earlier models of P2P systems (ie., simple file sharing) and it does not consider ontological relationships or interaction patterns between peers. Authors of [19] offer a framework for an ontology discovery, reuse and evolution in a distributed environment. Their work could potentially be applied to ours in the continuing maintenance of the registry/mapping information in a community.

III. WS-CatalogNet: Overview

We first set the conceptual background of WS-CatalogNet as a catalog portal framework [1], [2], [3], [4]. After an overview of WS-CatalogNet, we will then focus our discussion on monitoring and re-structuring. The main ideas of WS-CatalogNet are the concepts of catalog service communities and *peer relationships* among them. Further details about the framework we propose for building and peering communities can be found in [2], [20]. Here, we only describe the concepts that are central to understanding the monitoring/restructuring communities.

A. Catalog Service Communities

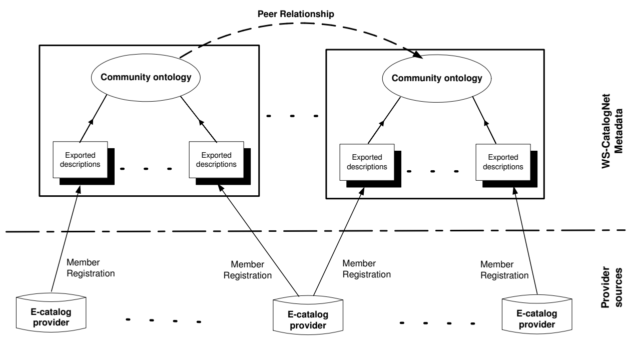
Figure 1(a) shows the metadata model implemented by WS-CatalogNet. A community is a *container* of catalogs of a specific domain (eg., community of `Airlines`, `CarRentals`). A *community ontology* provides a description of desired products (eg., airline tickets) without referring to an actual catalog provider (eg., `Qantas Airlines`). Catalog providers register their catalogs into a community as members by *exporting* (all or part of) their descriptions. Moreover, communities themselves can be linked to facilitate interoperability across overlapping domains.

A community is described by a set of categories. A category, in turn, is described by a set of attributes. Categories within a community may be inter-related via the specialization (subsumption) relationship. For example, in Figure 1(b), the community `FlightCenter` has a category `Flights`, which is described using attributes `arrival`, `departure` and `price`, etc. and the category has two sub-categories (ie., specialization); `Domestic` and `International`.

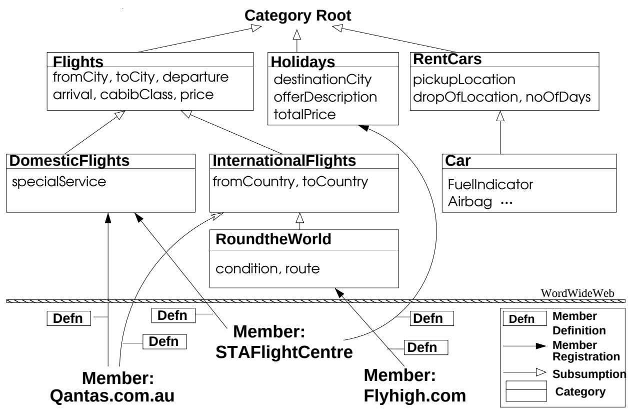
To provide formal semantics, necessary for precise characterization of queries over the communities, we use a (concept) class description language that belongs to the family of description logics [21]. In the following, we illustrate the main constructs of this language via examples[3].

In WS-CatalogNet, a community ontology is described in terms of *classes* (unary predicates) and *attributes* (binary predicates). Class descriptions are denoted by expressions formed by means of the following constructors.

[3]A description of the syntax and semantics of the proposed language is presented in [2].



(a) Metadata Representation of the Network



(b) A Community Definition (categories and members)

Fig. 1. Community Ontology

- class conjunction ($\sqcap$), e.g., the description `Travel` $\sqcap$ `Accommodation` denotes the class of products which are instances of the classes `Travel` and `Accommodation` (e.g., a `Hotel`),
- the universal attribute quantification ($\forall R.C$), e.g., the description $\forall$`arrivalDate.Date` denotes the class of products for which all the values of the attribute `arrivalDate` are instances of the class `Date` (i.e., the data type of the attribute `arrivalDate` is `Date`),
- the existential attribute quantification ($\exists R$), e.g., the description $\exists$`Price` denotes the class of products having at least one value for the attribute `Price`.

Category Definition: A *category definition* is specified as follows: `Cname` $\equiv$ `CatDescr`, where

- `Cname` is the name of the category,
- `CatDescr` is a class description that defines `Cname`.

For example, using the language, the category `Domestic Flights` in Figure 1(b) can be described as follows:

$$\texttt{DomesticFlights} \equiv \texttt{Flights} \sqcap \forall \texttt{specialService.String} \sqcap \exists \texttt{specialService}$$

This definition states that the category `Domestic Flights` inherits all the attributes of the category `Flights`, and has one additional attribute, namely `specialService`.

We would like to emphasize that the use of a formal language is transparent to community providers and users. Indeed, WS-CatalogNet provides a graphical editor that supports a community provider in creating a community and defining the community ontology. After the definition of an ontology, the editor automatically generates the class descriptions of the ontology (i.e., class descriptions of all categories).

Member Definition: When a catalog is registered to a community, the catalog provider supplies a capability description which specifies the categories and attributes of the community ontology that are supported by the catalog. This form of information is referred to as a *member definition or description*. Member definitions are also converted to class descriptions expressed using the description language and stored in the community meta-data repository.

A *member definition* specifies the query capabilities of a given catalog as follows: `Mname_Dname` $\equiv$ `MDescr` where `Mname_Dname` is a member definition name made of `Mname`, a member name (i.e., an unique identifier of a member), and `Dname`, the name of a description. `MDescr` is a class description that specifies which products are actually provided by this member. For example, the catalog `flyhigh.com`, which offers a range of flight information, can be registered with `FlightCentre` using the following member definition:

$$\texttt{flyhigh.com\_International} \equiv \texttt{InternatioanlFlights}$$

This definition states that `flyhigh.com` supports all attributes in the category `InternationalFlights` as well as all attributes inherited from the category `Flights` (as `InternationalFlights` is a sub-category of `Flights`). Each member can provide several definitions. When registering with a community, the catalog providers are in charge of providing and maintaining the exported descriptions of their catalogs as well as the wrapping utilities required to translate user queries into the catalog native query models.

Community Definition: A community ontology consists of a tuple $\texttt{CS} = (\texttt{C}, \texttt{S})$, where `C` is a set of category definitions, `S` is a set of community names that have peer relationships with the community. A catalog community consists of a tuple $\texttt{CAT} = (\texttt{CO}, \texttt{M})$, where `CAT` is the community name, `CO` is the community ontology, and `M` is a set of member definitions. We assume that the set of class descriptions in a catalog community (i.e., category definitions and member definitions) is acyclic, i.e., there does not exist cyclic dependencies between class definitions.

B. *Peering the Communities*

Through a *peer relationship* between communities, they forward queries to each other. To form a peer relationship, community providers need to discover other communities whose domains are relevant/similar to their communities. The community registry is hosted by WS-CatalogNet for the purpose. Typically, a provider of a community $\texttt{C}_\texttt{i}$ may decide to form a peer relationship with another community `Cj`, if `Ci` has categories that are considered analogous, or interchangeable to `Cj`'s categories (e.g., category `Accommodation` in community `Travels` and category `Budget Hotel` in community `Hotels`). When a link is formed, a weight (a value between 0 and 1) is attached to the link as a way of indicating the "relevancy" of the peer relationship.

C. Collaborative Query Processing between Communities

This section explains our approach for the query reformulation. We use a rewriting algorithm, called BQR[4], that allows to reformulate a community query[5] Q, expressed as a class description over the community ontology, into sub-queries that are expressed in terms of the community member definitions. A novel feature of BQR is that besides the re-writing of the community query, it also identifies parts of the query that cannot be answered by any of the community members. This feature forms the core of our collaborative query processing.

The algorithm takes as input a catalog community definition $\mathtt{CAT} = (\mathtt{CO}, \mathtt{M})$ and a community query Q which is written in terms of the ontology `CO`. It then computes a set of rewritings $R(Q) = \{r_i(Q)\}$. A rewriting r_i is a couple $r_i = (Q_{local}, Q_{rest})$ where:

- $Q_{local} = \{(q_j, m_j)\}$ is a set of pairs (q_j, m_j) where q_j is the part of the query Q that can be answered by the member definition m_j.
- Q_{rest} is the part of the query Q that cannot be answered by the members of the actual community.

Let us consider a simplified example, assume a query "select flights from `FlightCentre` where `fromCity` = Sydney, `toCity` = Paris, `price` < 2000 and `Insurance` = included". The key attributes in this query are {`fromCity`, `toCity`, `price`, `Insurance`}. Assume that the outcome of BQR is: Q_{local} = {(`fromCity`, `toCity`, `price`), `Qantas`}, Q_{rest} = {`Insurance`}. This means that the member `Qantas` of `FlightCentre` community has `fromCity`, `toCity`, and `price` information. However, there is no member in the community who can answer `Insurance`. Therefore, to serve the query, the community will collaborate with peers to identify *external* member(s) who can provide the missing information.

Note that a query Q can lead to several alternative rewritings. For example, the Q_{local} of Q is not necessarily unique. That is, there could be many members that can answer the same part of the query. In [20], we introduced an utility function that selects *only* those rewritings that maximize the user satisfaction with respect to a given Quality of Service criteria (called *best quality rewritings*).

In a nutshell, the query rewriting problem can be formally stated as follows: Let $\mathcal{C} = \{c_i \equiv description_i,\ i \in [1, n]\}$ be a set of class definitions corresponding to member definitions, and let Q be a class definition that denotes a community query. Then, can Q be reformulated as a conjunction of class names $E \equiv c_{i_1} \sqcap \ldots \sqcap c_{i_m}$, with $1 \leq m \leq n$ and $c_{i_j} \in \mathcal{C}\ for\ 1 \leq j \leq m$, such that E contains as much as possible of *common information* with Q? (E is called a rewriting of Q using $\mathcal{C}$.)

[4]The readers are referred to [2] for details and proof of this algorithm.
[5]The term refers to the query that is submitted to a community by a user.

Of course, the complexity of this algorithm is transparent to the users. The graphical query interface in WS-CatalogNet lets the user easily formulate a query (by point&clicks). The user clicks a category and then selects attributes to be queried on, and specify values for the attributes if desired (e.g., "*category:Flights, attributes: destination, departure, price, values: price* $\leq$ *2000*"). The query interface automatically converts the user formulated query to a class description.

The collaborative query processing technique consists of two steps. Whenever a query is submitted to a community C, it does the following:

- First Step: Identify the combination of catalog members whose query capabilities, when put together, satisfy all constraints expressed in the query. The catalog members can be local (ie., belonging to C), or external (ie., belonging to C's peers). This step is realized by the BQR algorithm and is referred to as *Meta Query Stage*.
- Second Step: Send the sub-queries to the identified members and collect the results. This step is referred to as *Actual Query Stage*.

Each community has a *forwarding policy*. This controls what should be done with Q_{rest} in the meta query stage. The forwarding policy can express (i) which part of the query should be forwarded, (ii) when the query should be forwarded (e.g., when no local members can answer, when the community is too busy etc.) (iii) to which peer (e.g., all, top K, random, etc.,) the query should be forwarded and (iv) how far the query should be forwarded (hop limit). From the previous example, the result of forwarding may be {`Insurance`, `STA.BestTravel`}, meaning that `BestTravel` (a member of community `STA`) can provide the `Insurance` information.

It should be noted that, although important, the issue of assembling actual results returned by selected catalogs, is outside the scope of the discussion. As a naive solution, we assume that every product information exists in WS-CatalogNet carries a universal product identifier which is uniquely understood (e.g., in the domain of flight tickets, the flight number is unique). This identifier is used to combined (ie., join) the results returned from different members in the actual query stage.

IV. RESTRUCTURING COMMUNITIES NET

In this section, we introduce an approach to monitor and adapt the community network. The approach we propose consists of the following three building blocks:

- *logging community events*: Any events occurring within a community or between communities are logged. Such events may include receiving a query from a user, sending a query to a peer, receiving an error from a member, etc.
- *analyzing community interaction patterns*: The event logs reflect what kind of interactions exist between communities. It is possible that over a period of time such interactions would form certain patterns (eg., community `A` sends a query to community `B` instead of community `C`. The member `M` of community `A` does not respond to queries most of the time, etc.). Amongst all possible

patterns, we pre-define *patterns of interests*. Each pattern of interest is linked with a restructuring operation. If any of the patterns of interests is observed in the log, the system will suggest the administrator to perform the associated operation. The operation can also be automatically performed based on ECA rules defined by the administrator.

- *restructuring operations*: An adaptation process is realized by a set of community network restructuring operations such as `addPeer`(), `upgradePeer`(), `mergeCommunities`(), etc.

WS-CatalogNet realizes the above principles via `Community Monitor`. Every individual community is associated with a community monitor. It gives the community access to the operational knowledge such as interaction patterns (ie., querying for a pattern) as well as a means for analyzing event logs and invoking the restructuring operations.

Under the current implementation, an administrator (e.g., the community provider) would use the monitor to explicitly search a pattern and perform restructuring operations. However, the design of the `Community Monitor` also supports both automatic and manual execution of an *action*. The automatic actions are controlled by *monitoring policies* designed by administrators. The policies specify restructuring directives via ECA (Even Condition Action) rules. For example, a monitoring policy may specify that a peer link will be disabled after a number of unsuccessful query forwarding. Therefore, the complete process of monitoring, detecting possible restructuring needs and performing the operations can be scheduled and automated via the policies.

In the remainder of the paper, a number of interaction patterns and the restructuring operations will be discussed. Before proceeding, we explain how the events are handled.

A. Event Collectors

Collecting events is facilitated by means of a logging service called `Event Collector`. The community monitor captures the events and sends them to the event collector for logging[6]. To maintain uniform way of tracking interactions, all event collectors in WS-CatalogNet agree on the system-wide events and their parameters to be logged. A subset of the events and their parameters are shown in Table I. Note that the events are divided into two classes: *Meta Query Stage* events (names are prefixed with MQ) and *Actual Query Stage* events (names are prefixed with AQ). The MQ events occur during the first step of query processing where the communities collaborate with each other to identify the set of members (local and external) that are relevant to a query. The AQ events occur in the second step of query processing, where the community sends the subqueries to the selected members and collect the results.

To make use of the log data, we make necessary transformation on the log file (eg., cleansing, sessionising). The log collected from a community is kept privately to the associated community monitor. We make an assumption that the log from all monitors can be shared by uploading them periodically to a central location for more global analysis. Hence, the log is analyzed in two scopes: *local* and *global*. In the local scope, a community's administrator uses the log from the community's monitor and focus on what kind of restructuring can be done to the immediate peers (i.e., peers that are directly linked from the community), as well as to the community. The global scope analysis is carried out by WS-CatalogNet system administrators using the log uploaded from all monitors. The restructuring in the global scope focuses on having a global view of interactions between all communities that otherwise not possible to detect from looking at the local scope only.

[6]Logging can be made either to a file or a database.

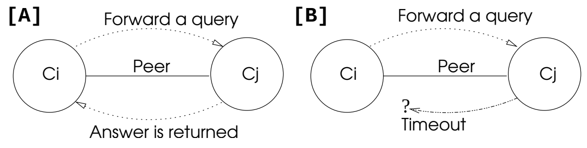


Fig. 2. Interaction Patterns for Upgrading/Downgrading Peer Relationship

B. Query Interaction Patterns

We now introduce a technique to assist the administrators to make decisions as to what kind of adaptation should be done. We propose *Predefined Interaction Sequences (PIS)* as a means to analyze community interaction patterns that may interest administrators. A PIS represents an interaction pattern between communities. The idea is that we associate a restructuring operation (eg., removing a link, merging communities, adding a new member to community, etc.) with a particular PIS. If an analysis shows that a particular PIS occurs frequently in the event log, an action (ie., the restructuring operation(s) which is associated with the PIS) is performed either automatically via a monitoring policy, or manually by the administrator. A predefined interaction sequence is formally defined as follows:

Definition 1: **Predefined Interaction Sequence (PIS).** $\mathcal{E}$ denotes the set of all defined activity events (see Table I). A predefined interaction sequence PIS of length n ($0 \leq n$) is a vector of ordered events PIS $= <e_1, e_2, ..., e_n>$ where $e_i \in \mathcal{E}$ (i = 1, ... n). □

A PIS is matched against each query session[7] in the log file to check whether the sequence exists in the session. We refer to the number of occurrence of a PIS in the log file as *Frequency*[8]. In the following, we present some of predefined interaction sequences.

1) Local Scope Analysis: For the local scope analysis, the PIS is matched against query sessions in a local log file of a single community.

a) Upgrading/Downgrading Relationships.: Consider an interaction sequence in Figure 2 (A), this pattern is designed to capture situations where community c_i forwards a query to one of its peers c_j and c_j returns an answer. The fact that this pattern occurs frequently implies that c_i collaborates with c_j frequently and c_j is responsive to c_i's query. In this case, the

[7]A query session is identified by the `qID` in the event.

[8]We refer readers to [3] for detailed description and issues related to frequency of a pattern

TABLE I
EXAMPLES OF THE COMMUNITY ACTIVITY EVENTS, NOTED $\mathcal{E}$

Meta Query Stage Event	**Description**
MQUserQryReceived(logger, when, qry, qID)	When a community receives a query from the user. `logger` is the name of the community who is logging this event, `when` records the time of the event, `qry` is a string that contains the query itself and `qID` is a unique identifier issued for the query for tracking purposes.
MQForwardedPeer (logger, to, when, qry-part, qID)	When a community forwards `qry-part` to the peer(s). Note that `qID` is a unique identifier issued for the original query.
MQAnswerReceived (logger, from, when, answer, qID)	When a community receives the corresponding result of the forwarding from one of its peers.
MQLogTimeOut (logger, from, qID)	When a community does not receive a response from a peer within a given time. `from` is the peer community that failed to send the response back.
Actual Query Stage Event	**Description**
AQPeerQryReceived (logger, from, when, qry, qID)	When a community receives the query that is to be locally executed by its members. `from` is the sender, `qry` is the query that is to be executed.
AQSentLocalMember (logger, to, when, qry-part, qID)	When a community sends the query to one of its members. `to` is the member of the community, `qry-part` is the query that is to be executed.
AQLocalAnswerReceived (logger, from, when, ans, qID)	When a community receives the corresponding result (`ans`) of the query processing from one of its members.
AQExtAnswerReceived (logger, from, when, ans, qID)	When a community receives the corresponding result of the query processing from an external member of a peer.

administrator of c_i may choose to upgrade the weight of the relationship. We define the related PIS as follows:

Definition 2: (PIS_{up}.) (PIS Upgrade) The pattern represents the situation where a forwarding of a query to a peer (represented by the event `MQForwardedPeer`) is followed by a return from the peer (represented by the event `MQAnswerReceived`):

$$\mathtt{PIS_{up}} = \langle \mathtt{MQForwardedPeer}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid}), \mathtt{MQAnswerReceived}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid}) \rangle$$

$\mathtt{c_i}, \mathtt{c_j}$ are the names of communities and the qid which appears in all events should be the same.
In the above sequence, $\mathtt{c_i}$ has forwarded a query to $\mathtt{c_j}$ and $\mathtt{c_j}$ returned a response[9]. If this pattern is found to be prevalent, the weight of the relationship between c_i and c_j is increased.

On the other hand, Figure 2 (B) shows the pattern which is designed to capture situations where community c_i forwards a query to c_j and c_j does not return anything (i.e., timeout). This may indicate that the given peer relationship is not responsive and does not contribute positively in query sessions. If this happens often, c_i may downgrade the peer relationship. We define the related PIS as follows:

Definition 3: (PIS_{down}.) (PIS Downgrade) The pattern represents the situation where a forwarding of a query to a peer is always followed by a timeout:

$$\mathtt{PIS_{down}} = \langle \mathtt{MQForwardedPeer}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid}), \mathtt{Timeout}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid}) \rangle$$

$\mathtt{c_i}, \mathtt{c_j}$ are the names of communities and the qid which appears in all events should be the same.
If this pattern is found to be prevalent, the weight of the peer relationship between c_i and c_j is decreased.

b) Adding New Members: Figure 3 illustrates a situation where c_i forwards Q_{rest} to peer c_j, and c_j forwards it to c_k. c_k lets c_j know that a member of c_k can answer Q_{rest}. c_j relays it back to c_i. Then, c_i finally sends the query to

[9]Any number of other events may come between the two events. However, the order of events in a pattern is always respected.

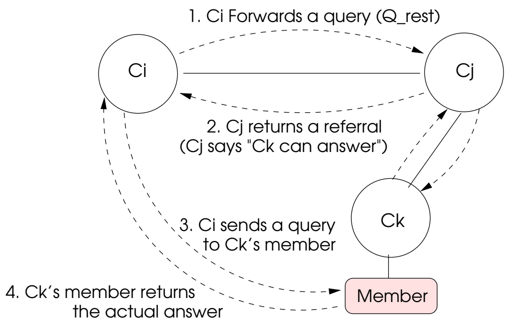

1. Ci Forwards a query (Q_rest)
Ci
Cj
2. Cj returns a referral (Cj says "Ck can answer")
Ck
3. Ci sends a query to Ck's member
4. Ck's member returns the actual answer
Member

Fig. 3. A pattern to find a new member

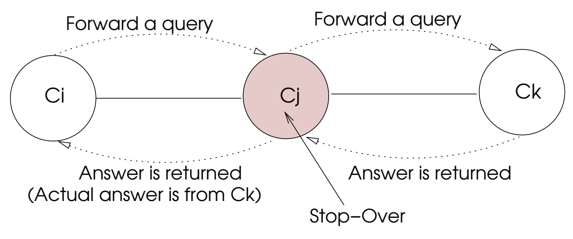
the a member of c_k. That is, the figure summaries a situation where the part of queries that could not be answered by c_i frequently gets resolved by a member of c_k eventually. If this happens frequently, it may indicate that if the member of c_k also becomes a member of c_i, c_i would have not have to forward Q_{rest} anymore (i.e., it can be resolved locally).

Definition 4: (PIS_{newmem}.) (PIS newMember) The pattern represents the situation where a community forwards Q_{rest} to a peer who returns an identification of an external member that can answer Q_{rest}. The community sends the query to the external member and receives the results.

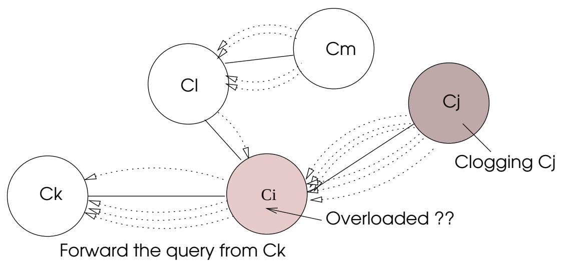
$$\mathtt{PIS_{newmem}} = \langle \mathtt{MQForwardInitiated}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid}), \mathtt{MQAnswerReceived}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid}), \mathtt{AQSentExtMember}(\mathtt{c_i}, \mathtt{c_k}.\mathtt{member}, \mathtt{qid}), \mathtt{AQAExtAnswerReceived}(\mathtt{c_i}, \mathtt{c_k}.\mathtt{member}, \mathtt{qid}) \rangle$$

$\mathtt{c_i}, \mathtt{c_j}, \mathtt{c_k}$ are community names, $c_k.member$ denotes the external member belonging to c_k. qid which appears in all events should be the same.
If the above pattern occurs frequently, c_i might consider adding the member of c_k to itself.

Using the community monitor the administrator would analyze which query attributes frequently appear in Q_{rest}. The attributes that most frequently appear in Q_{rest} is identified as the "missing information" in the community. Once the miss-



(a) Stop-Over



(b) Clogging Community

Fig. 4. Some Global Interaction Patterns between Communities

ing information is identified, the administrator may search, through the WS-CatalogNet's registry, for new members who are capable of supporting the attributes, so that queries can be resolved locally as much as possible.

2) Global Scope Analysis: The global scope analysis enables the WS-CatalogNet administrators to see the interaction between all communities involved in query sessions. We present a few examples here.

a) Finding Stop-Over Communities: Consider the forwarding scenario depicted in Figure 4(a). The forwarding process that involves communities c_i, c_j and c_k shows that the answer for c_i's query is actually obtained from c_k via c_j. In this case, it can be said that c_j served as a stop-over for c_i to reach c_k. If this pattern occurs frequently, c_i may decide to forward queries directly to c_k, by-passing c_j. To detect situations like this, we define a PIS as follows:

Definition 5: ($PIS_{stopover}$.) (PIS Stop-Over) The pattern represents the situation where a forwarding process reveals a stop-over peer:

$$\begin{aligned}\mathtt{PIS_{stopover}} = \langle &\mathtt{MQForwardInitiated}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid}),\\ &\mathtt{MQForwardedPeer}(\mathtt{c_j}, \mathtt{c_k}, \mathtt{qid}),\\ &\mathtt{MQAnswerReceived}(\mathtt{c_j}, \mathtt{c_k}, \mathtt{qid}),\\ &\mathtt{MQAnswerReceived}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{qid})\rangle\end{aligned}$$

$\mathtt{c_i}, \mathtt{c_j}, \mathtt{c_k}$ are community names and the qid which appears in all events should be the same.

b) Finding Clogging Communities: A community has a limited capability in the number of queries that it can process at a given time. In an environment like WS-CatalogNet, some communities can be overloaded with too many requests (i.e., query processing) from peers at a given time. If such overloading frequently happens to a community, the administrator of the community may consider performing some restructuring (e.g., reducing the number of incoming peers). Furthermore, it is worthwhile to investigate and identify who are the peers that are causing the community's overload (e.g., forwarding many requests to the community). Such peers are said to be *clogging* the community.

Consider a situation illustrated in Figure 4(b). Assume a community c_i and its peer c_j. Also, assume that c_j is clogging c_i by forwarding the most queries to c_i among other peers. Then the event log shows that c_i subsequently forwards the queries received from c_j to other peer (e.g., c_k). This means that c_i can not process the queries forwarded from c_j. Hence, by forwarding many queries to c_i, c_j is creating unnecessary workload for c_i. If this happens often, the administrator may decide to disable the link between c_i and c_j (either temporarily, or permanently). Consider the following pattern (which is observed for a given period of time).

Definition 6: (PIS_{clog}.) (PIS Clog) The pattern represents the situation where a community receives a query from a peer and the query is subsequently forwarded to another peer.

$$\begin{aligned}\mathtt{PIS_{clog}} = \langle &\mathtt{MQPeerQryReceived}(\mathtt{c_i}, \mathtt{c_j}, \mathtt{time_n}, \mathtt{qid}),\\ &\mathtt{MQForwardedPeer}(\mathtt{c_i}, \mathtt{c_k}, \mathtt{time_m}, \mathtt{qid})\rangle\end{aligned}$$

$\mathtt{time_m}$, $\mathtt{time_n}$ represent the time-stamp of the event. $\mathtt{time_m}$, $\mathtt{time_n} \in [\delta, \gamma]$ where δ and γ are the period of time this pattern is considered for.
If this pattern occurs frequently and c_i has been overloaded often, the link between c_i and c_j may be disabled.

c) Merging Two Communities: The pattern PIS_{newmem} (Figure 3) illustrates that c_i continuously finds that members of c_k is used to answer queries. Consider that the exactly same sequence of interactions also happens from c_k's point of view. That is, c_k also finds continuously that members of c_i are used in resolving c_k's Q_{rest}. If these interactions are mutual, it may be beneficial that c_i and c_k are merged and become one community. Figure 5 depicts such scenario.

We have presented only a subset of possible patterns. There are other interaction patterns defined (e.g., patterns for monitoring member's quality, splitting a community, etc.) and the administrators may also defined and save ad-hoc patterns for the future use. It is noted that the community monitor provides a user interface to the *PIS library* where patterns are managed (eg., newly identified, created, updated, etc.)

C. Restructuring Operations

The community monitor is equipped with a set of restructuring operations which are performed on communities and their relationships. A few examples of such operations are: *addMember(), addPeer(), updatePeerRelation(), mergeCommunities()* etc. These operations are used, for example, to change the relationship between catalog communities (e.g., update the weight of a relationship), add a new peer to a community, or delete a peer link, etc.

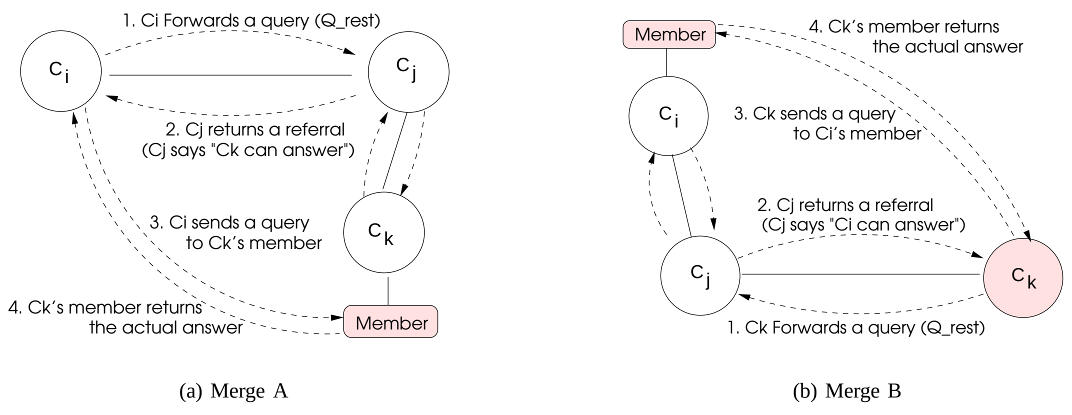

1. Ci Forwards a query (Q_rest)
Ci
Cj
2. Cj returns a referral (Cj says "Ck can answer")
Ck
3. Ci sends a query to Ck's member
4. Ck's member returns the actual answer
Member
(a) Merge A
Member
4. Ck's member returns the actual answer
3. Ck sends a query to Ci's member
2. Cj returns a referral (Cj says "Ci can answer")
1. Ck Forwards a query (Q_rest)
(b) Merge B

Fig. 5. A pattern for merging communities

The operations are divided into two classes: Intra-community, and Inter-community. Intra-community operations will be performed as a result of the local scope analysis. Inter-community operations are performed as a result of the global scope analysis.

V. WS-CatalogNet Implementation

In WS-CatalogNet, both catalogs and communities are presented as Web services[10]. They can be described, advertised and discovered using (XML-based) standard languages, and interact through standard Internet protocols. All services provide a SOAP-based programmatic interface for packaging requests and responses. These standards provide the building blocks for the service API descriptions and communication protocols inter-operation, the two basic elements of any programmatic interaction. The Universal Description, Discovery, and Integration (UDDI) specification is used for directory functionality. We used the IBM Web Services Development Kit 5.0 (WSDK), which provides several components for developing Web services. In particular, we used the UDDI Java API (UDDI4J) to access a private UDDI registry (i.e. hosted by WS-CatalogNet), as well as the WSDL generation tool for creating the WSDL documents and SOAP service descriptors for the catalogs and communities.

In UDDI registry, every Web service is assigned to a tModel. A tModel provides a semantic classification of a service's functionality and a canonical description of its interface. We have designed specific tModels for the catalogs as well as for the communities. However, since the query mechanism supported by WS-CatalogNet is more sophisticated than what UDDI provides, UDDI is only used to advertise and locate the communities.

WS-CatalogNet provides a generic community service, called `CatalogNetService`. This service has a number of pre-built operations which provide the functionality necessary for supporting query processing and the registration. In particular, the following operations are provided to support the proposed query mechanisms: `QueryProcessor()`, `QueryRouter()` and `ResultAssembler()`. The operation `QueryProcessor()` implements the BQR algorithm. The class `QueryRouter()` operation is used for routing queries based on the forwarding policies. The operation `ResultAssembler` is used for combining and selecting relevant catalogs.

[10]The VLDB Journal: Special Issue on E-Services, 10(1), Springer-Verlag Berlin Heidelberg, 2001

The run-time operation of a community service is defined through the combination of `CatalogNetService` and community metadata including: community ontology, member descriptions, P2P relationships, and forwarding policies. The community ontology is encoded as XML documents. The advantage of this architecture is that the developers who want to create a new community simply needs to provide the necessary meta-data information such as a community ontology and query forwarding policies. Tasks such as implementing the query rewriting and forwarding mechanisms are delegated to the generic service, thereby simplifying the development.

The `Community Monitor` implements the observation and restructuring technique. It integrates modules such as the event collector, PIS library and the interface to the restructuring operations.

VI. Experiments and Results

We conducted several experiments and the results are reported in the following.

A. Study on restructuring the network

Here, we looked at the effects of restructuring the community network. Our of the patterns presented in this paper, we chose two patterns: $PIS_{stopover}$ and $PIS_{overloaded}$ (a variation of PIS_{clog}) for the study. Through the experiments, we would like to demonstrate that, after restructuring:

- the average response time per query session *decreases*. Decrease in the response time would mean that restructuring helps reduce the time that takes for a query to be resolved.
- the proportion of incomplete sessions *decreases*. The incomplete sessions are the ones in which answers were not returned. The reasons for an incomplete session are time out, having no peer to which forward a query, and exceeding the hop limit. Decrease in the proportion of

incomplete sessions would mean that restructuring helps improve the chance of a query being resolved.

We have conducted our initial experiments in a simulated environment, in which we created communities as Java objects. Communities have their independent states and query capabilities. They autonomously invoke operations to process query and forward queries.

1) The Setup: Each community is a Java object which runs autonomously on an Intel Pentium IV 1.2GHz machine with 512MB memory. A community is associated with an XML file which includes: a local data set, forwarding policy and a list of its immediate peers. The local data set is a collection of metadata that describes community category definitions and member definitions. This is to represent the community's query capability. For the forwarding policy, each community specifies when to forward a query (e.g., only when query cannot be answered locally, or always) and how many "hops" are allowed. Queries used in the simulations are a set of keywords (randomly chosen from a pool of possible keywords). Each community has been assigned to a monitor to capture events such as receiving queries, forwarding queries etc. The monitor creates the entries to the log. The behavior of a community when processing a query is presented as a UML activity diagram in Figure 6. Note the session will also end when the community has no peers (i.e., leaf node in the WS-CatalogNet graph). The basic steps of the experiments are as follows:

– generate an instance of WS-CatalogNet.
– run 10,000 query sessions.
– collect the log and the measurement: *response time, number of incomplete sessions.*
– from the log, see if the chosen pattern exists.
– perform a restructuring operation according to what the pattern suggests.
– with the "restructured" WS-CatalogNet, run the same 10,000 query sessions.
– collect the measurements again and do "before-and-after" comparison.

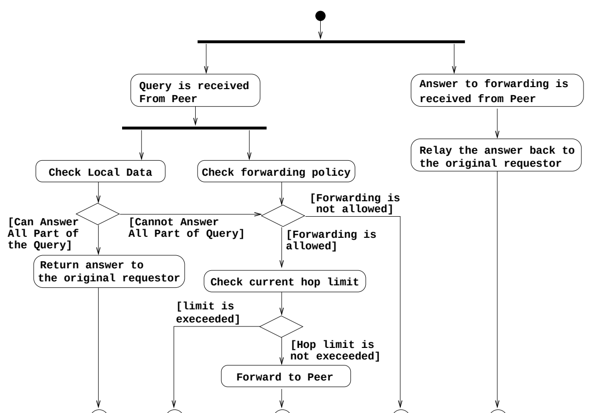

Query is received From Peer
Answer to forwarding is received from Peer
Check Local Data
Check forwarding policy
Relay the answer back to the original requestor
[Forwarding is not allowed]
[Can Answer All Part of the Query]
[Cannot Answer All Part of Query]
[Forwarding is allowed]
Return answer to the original requestor
Check current hop limit
[limit is execeeded]
[Hop limit is not execeeded]
Forward to Peer

Fig. 6. Individual Community's behavior

The program that launches each simulation takes three parameters: (i) *RLoc*, the location of the XML file. (ii) *Qfile*, the file that contains keywords for querying, and (iii) *MaxRun*, how many query sessions should be created. When the simulation starts each test run, it does the following:

– randomly picks, from `Qfile`, a query Q to be issued for the session.
– randomly picks, from the XML file, the first community to which Q is submitted.
– repeat this untill `MaxRun` has reached.

Once a community receives a query from the simulation launcher, it autonomously starts processing the query according to the behavior defined as in Figure 6.

We designed several different scenarios with the purpose of demonstrating the objectives we set out. In the first scenario, we tried the experiments on different P2P topologies as shown in Figure 7. These topologies are the ones that we believe suit the WS-CatalogNet framework, since they are not centralized.

In the second scenario, we tried the experiments on different sizes of the network; 50, 100, 200, 400 and 800 communities. In the third scenario, with the decentralized topology, we studied the clogging pattern.

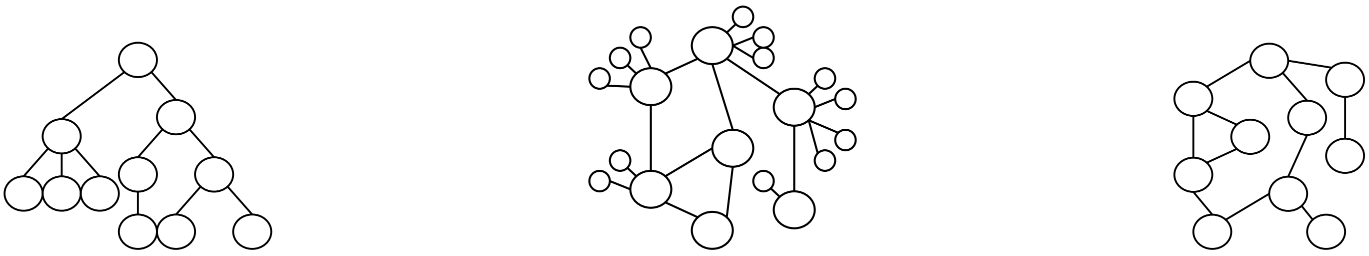
2) The First Scenario: In this scenario, we focused on finding the *stop over* communities. That is, from the event log obtained from the simulation, we searched for the sequence of events specified in the $PIS_{stopover}$ pattern. For example, we looked for sequences of events like the following (here, a query was forwarded, from `c1` to `c2`, then from `c2` to `c6`.):

$$\begin{aligned}&\langle MQForwardedPeer(c1,c2), MQPeerQryReceived(c2,c1),\\&MQForwardedPeer(c2,c6), MQPeerQryReceived(c6,c2),\\&MQAnswerReceived(c2,c6), MQAnswerReceived(c1,c2)\rangle\end{aligned}$$

To collect the "before-restructuring" data, we ran the simulation and obtained the global event log. Then, we searched the log for the $PIS_{stopover}$ pattern in various lengths (length here indicates the number of communities involved in the pattern). We wrote a Perl script for searching and counting of these patterns. The script also collected the average response time for each session, and the number of incomplete sessions. We identified five most frequent $PIS_{stopover}$ patterns, which identified five new peer relationships between communities. We performed `createPeer`$()$ operation, and ran the simulation again to collect the "after-restructuring" data. The result is presented in Table II.

The result showed that the restructuring performed improved both the `IncompS` and `RTime` measures in all topologies considered. However, the improvement was most significant in the decentralized model. This is an encouraging observation since the WS-CatalogNet framework assumes a decentralized model.

3) The Second Scenario: In the second scenario, we tried the same experiments as the first scenario, but on different sizes of the network (ie., different number of nodes in WS-CatalogNet). This time, we fixed the topology to a decentralized model. The result is shown in Figure 8. The numbers shown are the average values of three test runs in each setting (ie., three runs for the size-50 network, both before and after restructuring). The graph shows that regardless of the size, a significant improvement was made after the restructuring.



(a) Hierarchy (b) Central+Decentralized (c) Decentralized

Fig. 7. WS-CatalogNet topologies considered

TABLE II
RESULT OF THE FIRST SCENARIO

Hierarchy

	IncompS		R.Time		Summary	
	Before	After	Before	After	IncompS	R.Time
Run #1	6016	4382	9	7	Down	Down
Run #2	5593	4491	10	8	by	by
Run #3	6106	3801	9	7	28%	21%

Hybrid (Centralized+Decentralized)

	IncompS		R.Time		Summary	
	Before	After	Before	After	IncompS	R.Time
Run #1	5629	3492	9	8	Down	Down
Run #2	6162	3580	11	9	by	by
Run #3	7780	4842	11	9	39%	16%

Decentralized

	IncompS		R.Time		Summary	
	Before	After	Before	After	IncompS	R.Time
Run #1	5121	1183	10	8	Down	Down
Run #2	5753	1630	11	9	by	by
Run #3	5017	1390	11	9	73%	18%

R.Time: Average Response Time (in millisec)
IncompS: No of Incomplete Sessions (out of 10,000 sessions)

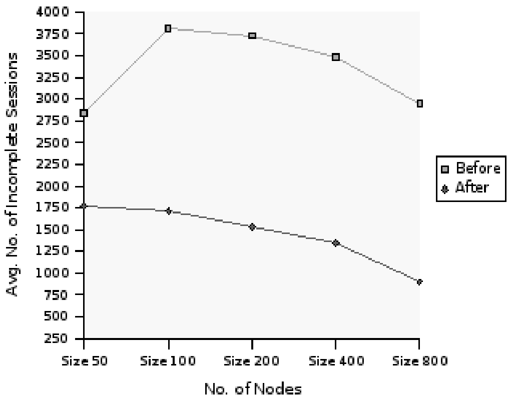


Fig. 8. Second Scenario: Different sizes of the network

4) The Third Scenario: In this scenario[11], we focused on finding the most *overloaded* community and identifying the peers that are overloading the community. We searched for the sequence of events specified in the $PIS_{overload}$ pattern. For example, we looked for sequences of events like the following (here, `c1` has forwarded a query to `c2`, giving `c2` the work):

$$\langle MQForwardedPeer(c1, c2), MQPeerQryReceived(c2, c1)\rangle$$

Once such communities, that is (a) overloaded ones, (b) top K number of peers that made them overloaded, are identified, we tried two different restructuring operations:

- Disable, temporarily, the peer links between the overloaded community and the communities in (b)
- Change the forwarding policy of the communities in (b) so that they forward less queries to others.

To measure the effectiveness of this restructuring, we collect extra measurement, *workload* beside the usual measurements. We defined the workload as the ratio of the number of requests a community receives over its maximum capacity. Each community's maximum capacity was allocated based on the average number of requests a community received in the past (e.g., from a few runs of simulations). The expectation was to see the average workload decreasing after restructuring. The result is shown in Table III. Again, the numbers presented are the average values of three test runs. The results indicate that both "disabling links" and "changing the policy" are effective ways of reducing the workload. Disabling link is slightly better in terms of reducing the number of incomplete sessions and the average workload.

TABLE III
RESULT OF THE THIRD SCENARIO

Disable Link

IncompS		R.Time		A.Wrkload	
Before	After	Before	After	Before	After
712	437	550	300	5.4	2.7
IncompS 46%↓		R.Time 39%↓		A.Wrkload 50%↓	

Change Policy

IncompS		R.Time		A.Wrkload	
Before	After	Before	After	Before	After
712	397	550	310	5.4	2.9
IncompS 44%↓		R.Time 45%↓		A.Wrkload 47%↓	

R.Time: Average Response Time (in millisec)
IncompS: No of Incomplete Session (out of 5,000)
A.Wrkload: Average Workload ($\frac{\text{No of requests}}{\text{Max capacity of community}}$)

B. Study on various network parameters

In this study, we focused on investigating the effects of various network parameters on the query performance. Again,

[11]This scenario was tested on an Intel Pentium III 863MHz machine, with 256MB memory.

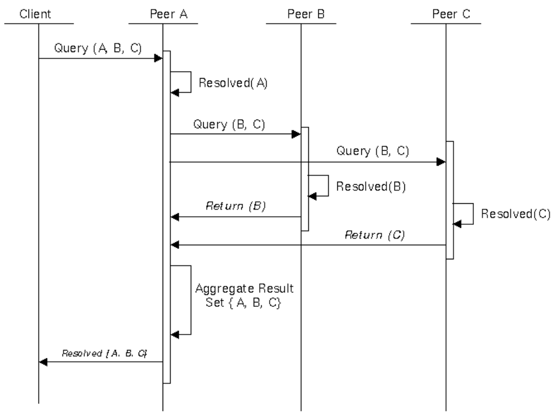


Fig. 9. Successful Query Process

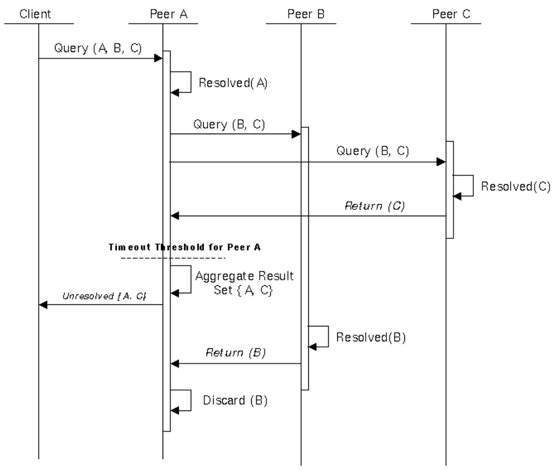


Fig. 10. Unsuccessful Query Process

a decentralized model is used for the P2P topology. The query performance is measured by comparing, for each community, the total number of queries issued, the number of queries that are resolved (`Resolved`), the number of queries that are not resolved (`Unresolved`) and the average response time in *resolved* sessions (`Rtime`), the average response time in *unresolved* sessions (`Ftime`), in difference test cases.

All communities are implemented as Web services (a Java servlet running on Apache Tomcat 4.1.18/Apache AXIS 1.1). We used 30 machines[12], each machine hosts a single community which is running as a Web service. The network topology information is dynamically generated and loaded to a central controller node[13] before running each batch of experiments. Each community provides a number of SOAP services such as processing queries, forwarding queries, managing a forwarding policy, logging, etc.

The forwarding policies of a community have two settings; `MultiCast` and `UniCast`. In the multicast setting, the community forwards a Q_{rest} to *all* of its known peers, whereas in the unicast setting, a single peer is chosen at random for forwarding. The exact behavior of a community is

[12]All Intel Pentium 3, 850Mhz 256RAM, Debian Linux

[13]The controller node is only responsible for initializing the P2P network of communities and is not involved in routing.

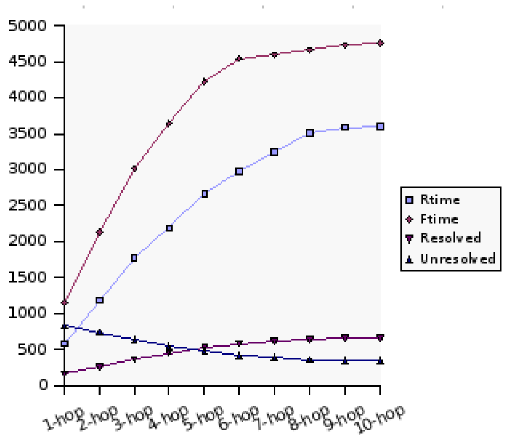


Fig. 11. Case One Results

depicted in Figures 9 and 10. We used four sets of test cases, each containing 1000 randomized queries. To ensure that the randomly generated queries are not biased, the each test case is executed 5 times with a new pseudo-random generation sequence. At the end of every run, statistics are collected for each community to show the total number of queries issued, the number of queries that resolves, the number of unresolved queries and response time. The final statistics gathered are the result of the average values from 5 runs. Out of all the test cases considered, we report the results on varying the hop count and the size of the network.

1) Case one: varying hop counts: In this test case, we varied the hop count setting for each community, ranging from one to ten. The forward policy was set to `Unicast` and the test case included 1000 query runs. Note that the response time is not considered here because it is clearly expected that the higher the hop counts, the longer the response time. The performance of a community is measured by `Resolved` (see Figure 11). Note that the numbers in the figure are the average values of five runs in each setting.

The results have shown that with the increasing number of hops, there is a proportional increase in the number of resolved queries. However, after `5-hop` and onwards the increase has slowed down and by the `8-hop` setting, the number of resolved queries did not show much differences.

2) Case Two: varying the size of the network and the number of constraints in a query.: In this test case, we studied the relationship between the community's performance and the size of the network and the number of constraints in the queries. The size of the network varied from 10 peers to 30 peers and the number of constraints in a query ranged from 10 to 50[14]. From the result (see Figure 12), it can be clearly seen that `15-constraint` setting seemed to result in the highest number of resolved queries (for all sizes of the network). The size of the network does not appear to affect the number of resolved queries. As shown in Figure 12, there is a slight tendency for the performance to degrade as the number of the constraints or the number of peers increases, but, this is

[14]The number of community category attributes in a query is used as the constraints.

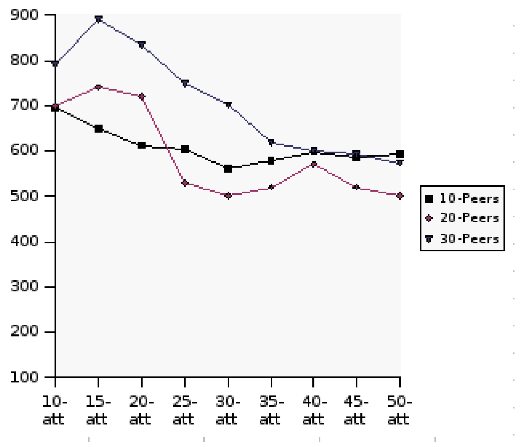


Fig. 12. Case Two Results

definitely sub-linear.

VII. Future Work

We have proposed a framework which supports integration and evolution of catalog portals using the P2P data management paradigm. Future work includes (i) managing changes in community ontology (e.g., propagation of the changes to peers and its members), (ii) automated discovery of potential community members and monitoring community-member interactions, and (iii) further simulation studies on restructuring in a very large-scale systems over time.

Acknowledgement

We thank Meryem Pinar Domez and Vincent Liu for their invaluable contribution in the experiments. The work of the second author is partially supported by an ARC Discovery Grant - DP0452942.

References

[1] K. Baina, B. Benatallah, H. Paik, F. Toumani, C. Rey, A. Rutkowska, and H. Susanto, "WS-CatalogNet: An Infrastructure for Creating, Peering, and Querying e-Catalog Communities," in *Proc. of VLDB 2004*, Toronto, Canada, August 2004, demonstration paper.

[2] B. Benatallah, M.-S. Hacid, H. Paik, C. Rey, and F. Toumani, "Towards Semantic-driven, Flexible and Scalable Framework for Peering and Querying e-Catalog Communities," *Information Systems Journal, Special issue on semantic web services*, 2004, to appear.

[3] H. Paik, B. Benatallah, and R. Hamadi, "Dynamic Restructuring of E-Catalog Communities Based on User Interaction Patterns," *WWW Journal*, vol. 5, no. 4, pp. 325–366, 2002.

[4] A. Bouguettaya, B. Benatallah, L. Hendra, M. Ouzzani, and J. Beard, "Supporting Dynamic Interactions among Web-based Information Sources," *IEEE TKDE*, vol. 12, no. 5, pp. 779–801, Sept/Oct 2000.

[5] B. Yang and H. Garcia-Molina, "Comparing Hybrid Peer-to-Peer Systems," in *VLDB'01*, Rome, Italy, September 2001, pp. 561–570.

[6] A. Crespo and H. Garcia-Molina, "Routing Indices For Peer-to-Peer Systems," in *ICDCS'02*, Vienna, Austria, July 2002, pp. 23–35.

[7] S. Joseph and T. Hoshiai, "Decentralized Meta-Data Strategies: Effective Peer-to-Peer Search," *IEICE Trans. on Communication*, vol. E86-B, no. 6, pp. 1740–1753, 2003.

[8] P. Bernstein, F. Giunchigiloa, A. Kementsietsidis, J. Mylopoulos, L. Serafini, and I. Zaihrayeu, "Data Management for Peer-to-Peer Computing: A Vision," in *WebDB'02*, Madison, Wisconsin, June 2002, pp. 89–94.

[9] S. Bergamaschi, F. Guerra, and M. Vincini, "A peer-to-Peer Information System for the Semantic Web," in *Proc. of AP2PC 2003*, July 2003.

[10] D. Beneventano, S. Bergamaschi, F. Guerra, and M. Vincini, "The momis approach to information integration," in *Proc. of ICEIS'01*, Setbal, Portugal, 7-10 July 2001.

[11] S. S. Chawathe, H. Garcia-Molina, J. Hammer, K. Ireland, Y. Papakonstantinou, J. D. Ullman, and J. Widom, "The TSIMMIS Project: Integration of Heterogeneous Information Sources," in *Proc. of IPSJ*, Tokyo, Japan, October 1994, pp. 7–18.

[12] D. Kossmann, "The State of the Art in Distributed Query Processing," *ACM Computing Survey*, vol. 32, no. 4, pp. 422–469, 2000.

[13] W. Ng, B. Ooi, K. Tan, and A. Zhou, "PeerDB: A P2P-based System for Distributed Data Sharing," in *ICDE'03*, Bangalore, India, March 2003.

[14] A. Halevy, Z. Ives, D. Suciu, and I. Tatarinov, "Schema Mediation in Peer Data Management Systems," in *ICDE'03*, Bangalore, India, March 2003.

[15] M. Arenas, V. Kantere, A. Kementsietsidis, I. Kiringa, R. J. Miller, and J. Mylopoulos, "The Hyperion Project: from Data Integration to Data Coordination," *SIGMOD Record*, vol. 32, no. 3, pp. 53–58, 2003.

[16] W. Nejdl, M. Wolpers, W. Siberski, C. Schmitz, M. Schlosser, I. Brunkhorst, and A. Lser, "Super-Peer-Based Routing and Clustering Strategies for RDF- Based Peer-To-Peer Networks," in *Proc. of WWW '03*. Budapest, Hungary: ACM Press, May 2003.

[17] B. Yang and H. Garcia-Molina, "Designing a Super-Peer Network," in *ICDE'03*, Bangalore, India, March 2003.

[18] B. Cooper and H. Garcia-Molina, "Ad hoc, Self-supervising Peer-to-Peer Search Networks," Database Research Group, Stanford University, California, Tech. Rep., Feb, 2003.

[19] A. Maedche, B. Motik, L. Stojanovic, R. Studer, and R. Volz, "An Infrastructure for Searching, Reusing and Evolving Distributed Ontologies," in *Proc. of WWW '03*. Budapest, Hungary: ACM Press, May 2003.

[20] B. Benatallah, M.-S. Hacid, H.-Y. Paik, C. Rey, and F. Toumani, "Peering and Querying e-Catalog Communities," CSE, UNSW, Sydney, Australia, Tech. Rep. UNSW-CSE-TR-0319, 2003, see ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0319.pdf.

[21] F. Baader, D. Calvanese, D. McGuinness, and e. D. Nardi andP. Patel-Schneider, *The Description Logic Handbook. Theory, Implementation and Applications*. Cambridge University Press, 2003.

Helen Paik is a lecturer at the School of Information Systems in Queensland University of Technology, Brisbane, Australia. Her research interests include P2P data management systems, personalization issues in Web services. She received her PhD in computer science from University of New South Wales, Sydney, Australia. She is a member of IEEE Computer Society and ACM.

Boualem Benatallah is a senior lecturer at University of New South Wales, Sydney, Australia. He received his PhD in computer science from Grenoble University, IMAG, France in 1996. His research interests include Web services, Semantics Web, and autonomous data sharing. He has published widely in international journals and conferences including IEEE TKDE, VLDB, PADD journals, ICDE, ICDS and ACM WWW, etc. He is a member of ACM and IEEE.

Farouk Toumani is a senior lecturer at the School of Engineering in Computer Science, Modeling and Applications (ISIMA), University Blaise Pascal of Clermont Ferrand, France. His research interests include Web services, semantics Web, and knowledge representation for databases. He has published widely in international journals and conferences such as ICDE, IEEE Internet Computing, VLDB, and Information Systems Journal.