PeeRCR：A Distributed P2P-Based Reusable Component Repository System

Dehua Chen[1], Ruiqiang Guo[1, 2], Jiajin Le[1], and Wei Shi[1]

[1] College of Computer Science, University of Donghua, Shanghai (200051), China
[2] College of Mathematics and Information Science, Hebei Normal University, Shijiazhuang (050016), China
{lydehua, grq, shiwei_jsj993}@mail.dhu.edu.cn
lejiajin@dhu.edu.cn

Abstract. For software reuse to be successful, a repository for storing and retrieving reusable components is essential. The traditional component repositories almost adopt Client/Server mode and offer a centralized authority on reusable components stored. However, such C/S-based repositories suffer from several limitations. This paper presents the design and implementation of an original P2P-based reusable component repository system called PeeRCR that enables the sharing of reusable component in a large distributed environment. We build the PeeRCR system using peer-to-peer distributed hash table protocol. The PeeRCR uses two kinds of index dictionaries (Local Dictionary and DHT-based Peer Dictionary respectively) together to assist in processing reusable component queries. We also implement a number of alternative scheme repositories to evaluate the performance of the PeeRCR. The experimental results demonstrate the feasibility and effectiveness of the PeeRCR for sharing reusable components in a large distributed and dynamic environment.

1 Introduction

Software reuse is considered by the software industry and academia as an efficient means for improving software development productivity and software product quality [1]. As a crucial infrastructure for supporting the practice of software reuse, component repository gains more and more attentions from academia and industries of software. Basically, the functionalities of component repository are to store, retrieve and manipulate large amounts of reusable components (e.g., COM, JavaBean or CORBA) in it. In the past decade, several component repositories have already been deployed [2], e.g., ALOAF (Asset Library Open Architecture Framework) [3], REBOOT (Reuse Based on Object Oriented Techniques) Library System [4][5], Agora [6], JBCL[7] and so on. All these repositories share a common feature: they are built in the Client/Server mode. In other words, as far as such C/S repositories are concerned, all reusable components are stored, retrieved and manipulated in a centralized way. However, such centrally authorized component repositories suffer from some limitations. First, it is hard for these repositories to achieve scalability. However, with component repositories developing and their customers mounting up, scalability plays a more and more important role. Second, these repositories tend to suffer from a single point of failure, that is, the bottleneck problem of the server limits the utility of component repositories. Third, centralized administration on reusable components is required, which introduces extra cost. Finally, resources at the network edge are unused and wasted.

To address the above limitations of C/S-based component repositories, this paper presents the PeeRCR that is an original distributed P2P-based component repository system for sharing reusable component in a large distributed environment e.g. Internet. In PeeRCR, we attempt to integrate P2P technology with the research and practice of component repository for the first time. P2P (Peer-to-peer) technology, a newly emerging paradigm of IT, is now regarded as a potential technology that could re-architect a large distributed system such as Internet [8]. Unlike Client/Server mode, in a typical P2P application, all the nodes or peers (e.g. PC over the Internet) share their resources and services by direct exchange between them without any centralized authority. These nodes act as both consumers and providers of data and/or services. Any survey on P2P research and applications shows a lot of desirable features of P2P technology such as pure decentralization, scalability, robustness, autonomy, data availability, easy adaptation, efficient and complex query searching. Following Napster [9], many new P2P file sharing systems keep on emerging in recent years, e.g., Gnutella [10], CAN [11], Chord [12] and Pastry [13]. However, the sharing files in all these P2P systems are mainly confined to music files, video, images or documents, e.g., Napster allows sharing of Mp3 music files. It is nature to expect that reusable components can be shared in the P2P environment as well. To our best knowledge, there is not any P2P system for sharing reusable components among peer nodes in a large distributed environment. However, it is extremely demanded, because in the predictable future, with the rapid growth of software reuse, it is expected that many academic, commercial, governmental and software producing organizations and even vast software developing fans would like to share their reusable components on Internet. Such a large amount of reusable components distributed over the Internet means wealth to any developer.

In this paper, we outline the design and implementation of PeeRCR for storing, retrieving and manipulating substantive reusable components resided at the network edges. The PeeRCR has several key features including:

First, in the PeeRCR, each peer node has installed a component repository management system that supports flexible component retrieval and manipulates components locally.

Second, the PeeRCR supports efficient query routing of reusable component. We adopt indices services (Local Dictionary and DHT-based Peer Dictionary respectively) strategy for assisting in processing user query of component.

Third, PeeRCR handles dynamic and ad hoc P2P environment efficiently.

To evaluate the performance of the PeerCR, we conduct our experiments on a cluster of 30 PCs with PentiumⅣ 1.8GHz to 1.4GHz CPU. Our experiment results show the feasibility and effectiveness of the PeerCR for sharing reusable components in a large distributed environment.

The rest of this paper is organized as follows. In section 2, we summarize briefly some preliminary knowledge including multi-faceted classification scheme, multi-faceted component retrieval and Distributed Hash Tables. Section 3 presents the details of the design of the PeeRCR. Section 4 outlines the method for processing user queries of components using indices services. In Section 5, the ability to adapt the system evolution of the PeeRCR is discussed. Section 6 presents our experimental study to evaluate the PeeRCR. Section 7 draws conclusions and presents future research directions.

2 Preliminaries

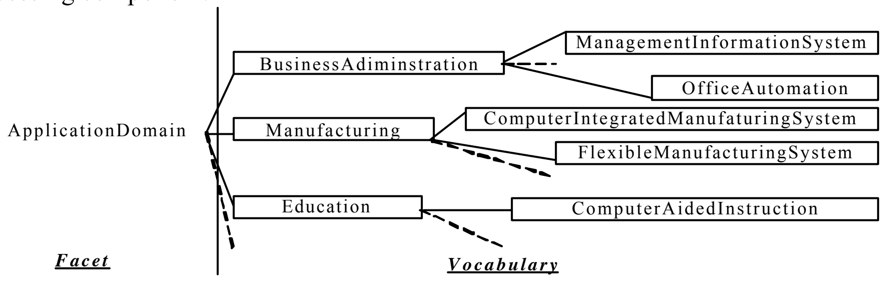
In the PeeRCR, we employ multi-faceted classification scheme for classifying reusable components, and adopt correspondingly multi-faceted component retrieval strategy. Thus, in this section, we first introduce briefly some background knowledge about these two aspects. We will also briefly describe distributed hash tables technique by which the PeeRCR is constructed.

2.1 Multi-Faceted Classification Scheme

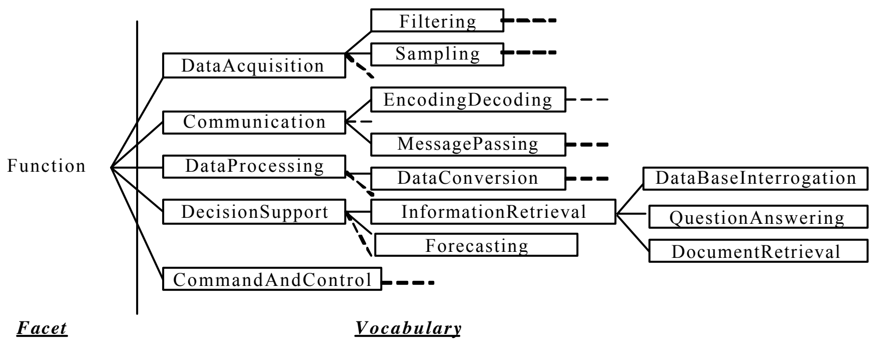
Though there have been different definitions of reusable components [14][15][16], the common features reusable components share are that raw reusable components are independently deployed piece of executed software; they are always encapsulated in the form of binary; they have a/several published interface(s). Because of the encapsulation of raw reusable components, they express little information for direct retrieval. The general solution is to classify/categorize raw components into various groups based on component characteristics. Then the users can search for appropriate components from the component repository according to one or more characteristics. In the last decade, the researches have resulted in a number of proposed methods to classify reusable artifacts. The mostly-utilized methods are those taken from library science including enumerated [17], keyword [18], faceted [19], and hypertext [20]. Of these, faceted classification gains more attentions than others since it presents an approach to classify reusable components based on some facet-term pair [19]. In faceted classification scheme, the essential is to predefine a collection of terms, also called as *Vocabulary*, for each facet by human experts. Notice that faceted classification method does not rely on complete partition of an entire object area; instead, it relies on synthesizing the subject area to identify a set of facet for describing components [19]. For a given vocabulary, the terms of vocabulary may be organized in hierarchical structure. The structural relationship of terms is a loose form of generalization/specialization relationship. Figure 2.1 illustrates two facets: 'application domain' (Fig. 2.1a) and 'Function' (Fig 2.1b), and their vocabulary hierarchy. For a given reusable component, we assign it with terms from the vocabulary of specific facet. The term assignment is referred to as classification.

In the PeeRCR, we predefine five facets: 'ApplicationDomain', 'Functionality', 'Role', 'ApplicationEnvironment', 'Representation', and their vocabulary as well.

Figure 2.2 illustrates an example of multi-faceted classification for a credit-card processing component.



(a)



(b)

Fig. 2.1. Hierarchies of terms for the facets: Application Domain and Function

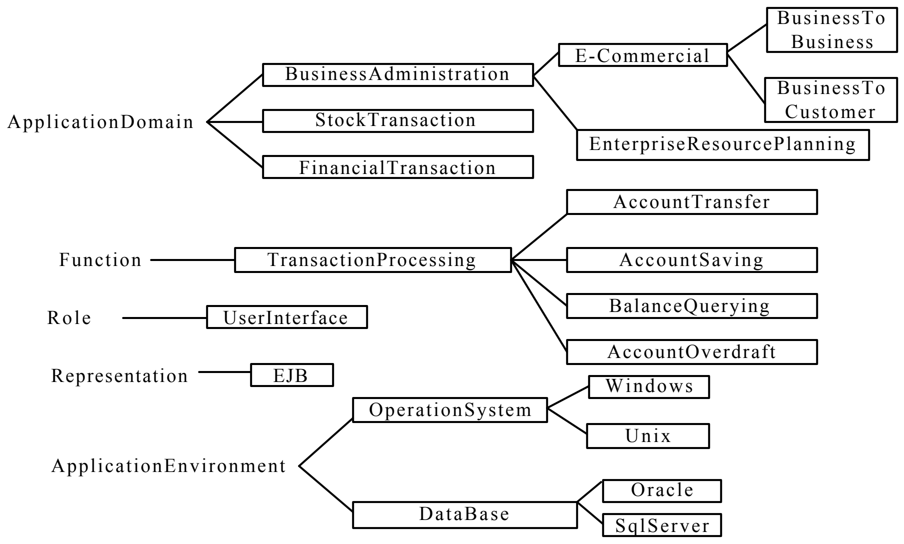


Fig. 2.2. A multi-faceted classification for a credit-card processing component

2.2 Multi-Faceted Component Retrieval

By describing and classifying reusable components with the multi-faceted scheme, a re-user query on reusable components consists of a list of so called *Facet-Term Query* (**FTQ**), considered to be ANDed that means performing intersection operation. Thus, we call the user query as *Facet-Term Querys* (**FTQs**). In its simplest form, each **FTQ** consists of a facet, and a list of terms, considered to be Ored that means performing union operation. In nature, **FTQs** implement a very simple Boolean Retrieval strategy. We symbolize **FTQs** as follows:

FTQs (Query):: =**FTQ**|**FTQ AND** Query

FTQ:: = Facet/ListOfTerms

ListOfTerms:: = Term|Term **OR** ListOfTerms

Example 2.1 Given a user submits a query as follows:

FTQs:: = {FTQ_1, FTQ_2, FTQ_3, FTQ_4, FTQ_5}

FTQ_1::=ApplicationDomain/BusinessAdministration/e-Commercial/B2B;

FTQ_2::= ApplicationDomain/Government;

FTQ_3::= ApplicationDomain/StockTransaction;

FTQ_4::= Function/TransactionProcessing/AccountTransfer;

FTQ_5::= Function/GovernAffairProcessing/Statistics.

Assume that the result set of FTQ_1 is Set of Component SoC_1, SoC_2 for FTQ_2; SoC_3 for FTQ_3; SoC_4 for FTQ_4; SoC_5 for FTQ_5. Then RoQ (Results of Query):: = $\{(SoC_1 \cup SoC_2 \cup SoC_3) \cap (SoC_4 \cup SoC_5)\}$. It is easy to know that the credit-card processing component from Figure 2.1 falls into the RoQ.

2.3 Distributed Hash Tables

The DHTs (Distributed Hash Tables) are proposed to efficiently store and retrieve objects over a large and dynamic distributed P2P environment [11,12,13] without depending on any centralized authority. DHTs have several desirable characteristics. First, while there are several implemented systems based on DHT with different design, they all support two basic hash-table operations: put (key, value) and get (key). Second, these systems can scale gracefully to a large number of nodes. Third, the lookup of these systems can be resolved using O(logn) network messages, where n is the total number of nodes in the system. While the design and implementation of the PeeRCR does not rely on specific DHT protocol and works well with any form of DHT system, our choice of Chord protocol [12] that is used as the experimental underlying circumstance involves the combination of easy-to-avail and popularity. There have been several successful systems adopting Chord, e.g. Cooperative File Storage [22]. Chord provides support for just one operation: given a key, it maps the key onto a node. Depending on the applications using Chord, nodes are responsible for storing the data object that associated with the key. Chord uses consistent hashing function to assign each node and key an m-bit identifier [12]. The set of both key and node identifiers are ordered in an identifier ring modulo 2^m, as shown in Figure 2.3. A key is assigned to the first node (successor node in clockwise direction) whose identifier is equal to or follows the identifier of the key in the identifier space. Chord protocol allows nodes and keys to insert or delete at any time, while it maintains efficient lookup using just O($\log^n$) state on each node in the network. Referring to [12], more detailed description of Chord and its algorithm obtains.

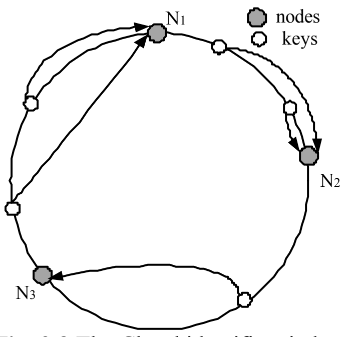


Fig. 2.3 The Chord identifier circle

Table 3.1. Local dictionary for each PeeRCR node P_i

KEYWORD	COMPONENT ID
FT_1	$Com_{1,1}$, $Com_{1,2}$,…, $Com_{1,k1}$
FT_2	$Com_{2,1}$, $Com_{2,2}$,…, $Com_{2,k2}$
…	…
FT_n	$Com_{n,1}$, $Com_{n,2}$,…, $Com_{n,kn}$

Table 3.2. Peer dictionary for the entire PeeRCR system

KEYWORD	PEER ID
FT_1	$Peer_{1,1}$, $Peer_{1,2}$,…, $Peer_{1,k1}$
FT_2	$Peer_{2,1}$, $Peer_{2,2}$,…, $Peer_{2,k2}$
…	…
FT_m	$Peer_{m,1}$, $Peer_{m,2}$,…, $Peer_{m,kn}$

3 System model

Unlike other existing peer-to-peer application systems, the design and implementation of a P2P-based reusable component repository system involves some new innovative ideas and is a challenging task due to the distinguishing features of reusable components. As suggested above, because reusable components are encapsulated in the form of binary, it is hard to retrieve reusable components directly whereby the information components carry. Therefore, as suggested in section 2, reusable components consist essentially of binary-formed artifacts with the accompanying textual classification documents. The popular measure is to transform the retrieval of reusable components into querying the textual classification documents.

Conceptually the PeeRCR has arbitrary number of peer nodes, each of which makes its reusable components available for other peer nodes. Let P_i ($1<i<n$) denote n nodes in the network, each of which publishes a set C_i of components available with a set D_i of accompanying multi-faceted classification documents. Let us imagine the following scenario: At some peer node in the PeeRCR network a user issue a query of reusable components. According to the query, the PeeRCR shall determine which nodes contain the relevant components and then contact each relevant node. The naïve way to processing the user query would be to send the query to each of participating nodes in the PeeRCR network. However, this method would work for a small number of peer nodes it certainly does not scale. To improve the scalability, the popular measure taken in peer-to-peer systems is to introduce indices service that helps to determine which nodes should receive queries based on query content. In PeeRCR, two kinds of index dictionaries (Local Dictionary and DHT-based Peer Dictionary

respectively) are constructed to assist in processing the user query of reusable components.

3.1 Local Dictionary

Designing the *local dictionary* (**LD**) for each peer node P_i is actually the process of building an inverted list index [21] over the multi-faceted classification files set D_i on the PeeRCR node. One natural way to construct **LD** is to map each term pre-assigned under certain facet for reusable components to the reusable components C_i published by the node. While this approach would not work well for the query of components because the users issue the query in the form of FTQs as shown in section 2.2. Hence, we shall construct **LD** (as shown in table 3.1) whereby extracting possible facet-term pairs as so called keywords from the multi-faceted classification document set D_i and indexing each possible facet-term pair (keyword) over the components C_i stored at the node P_i. In the table 3.1 $Com_{j,k} \in C_i$ refers to the reusable component id which contain keyword (facet-term pair) FT_j in its multi-faceted classification document at the node P_i, the keywords FT_j is required to be written in the form of **facet/term$_1$/term$_2$.../term$_l$** where term$_2$ is the specialization of term$_1$. Taking the credit-card processing component from figure 2.2 as an example, it is easy to deduce that **ApplicationDomain/BusinessAdministration/e-Commercial/BusinessToBusiness** is just a keyword, while **ApplicationDomain/BusinessAdministration/e-Commercial/** is also a keyword. We can see that the construction process of the keywords from multi-faceted classification documents is the process of traverse up the hierarchical structure of multi-facet documents from root to leaf, and can easily deduce that each FTQ in a query FTQs can be regarded as a keyword. **LD** is used for selecting reusable components at the node P_i. Given a FTQs of reusable component with multi FTQ (keywords), the PeeRCR shall perform intersection or union operation on the component ids of those keywords to generate the qualified component id list. The process of querying reusable component will be discussed in section 4 in detail.

3.2 DHT-Based Peer Dictionary

In addition of local dictionary at each PeeRCR node, we construct another kind of inverted list index, called as *peer dictionary* (as shown in Table 3.2) that maps facet-term pair (keyword) to peer nodes in the entire PeeRCR system. In Table 3.2, $peer_{j,k} \in P_i$ $(1<i<n)$ refers to the peer node id which contains reusable components whose multi-faceted classification documents hold the keyword FT_j. The keywords of Peer Dictionary are equal to the union set of all keywords published by all PeeRCR nodes. The reason for our construction of peer dictionary is that using the peer dictionary, a PeeRCR node's query engine can efficiently perform the same intersection or union operation to select the relevant remote peer nodes for sending the query directly to those nodes that are likely to have the relevant reusable components. Section 4 shall present how the PeeRCR system uses the peer dictionary.

There are three basic designs for routing user query of reusable components by peer dictionary. The first design is to build central peer dictionary. In this type of design one or more dedicated servers are used to maintain the complete peer dictionary of the entire system, and to servicing query routing. This approach is similar to traditional Client/Server mode and suffers from the problem of scalability. The second design is to duplicate peer dictionary on each peer node. In this type of design, when a new node join into the system it downloads the whole peer dictionary from any existing peer and it can immediately issue any query into the entire system. However, this approach also suffers from the problem of scalability because maintenance of the peer dictionary requires $O(n^2)$ number of messages for n nodes in the systems. Therefore, the PeeRCR chooses the third design in which peer dictionary is fully distributed among peer nodes in the network. The PeeRCR uses DHT mechanism to distribute peer dictionary in pieces, so called *DHT-based peer dictionary* (**DPD**), to each participating node. The construction of **DPD** involves two steps. In the first step, for each keyword in the Peer Dictionary, we use hash function to produce a key value. In the second phase, the Peer Dictionary is partitioned and distributed among the nodes in the entire PeeRCR system according to key values. The result of these two steps is that each node maintains a part of Peer Dictionary, which has/have keywords whose hash values fall into the node's responding range and their corresponding peer ids.

4 Processing Reusable Component Queries Using DPD and LD

The principle of FTQs, the basic mechanism of reusable component retrieval method, has been discussed in Section 2.2. In this section, we shall describe how the PeeRCR system uses DPD (DHT-based Peer Dictionary) and LD (Local Dictionary) for querying reusable components in a large distributed environment. Our algorithm for querying reusable components in the PeeRCR involves two main stages. In its first stage, the Relevant Peer id List (RPL) for FTQs is determined using DPD; in the second stage, the results of the first stage direct FTQs to each relevant peer node which shall return the resulting reusable components back by using LD. We detail these two stages as follows:

Stage Ⅰ:

* For any query FTQs= $\{FTQ_1,\ldots,FTQ_k\}$ received by one PeeRCR node e.g. P_j, the query engine of P_j hash each FTQ_i, $i\in[1..k]$, to get the node P_x, $x\in[1..k]$, which maintains the DHT-based peer dictionary (DPD) containing the Peer id List (denoted as PL_i) corresponding to FTQ_i. Notice that as suggested above, FTQ_i is equal to a keyword in DPD, therefore the hash FTQ_i directly is the same as the hash of keyword.

* P_x sends back PL_i to P_j. After all peer lists $\{PL_1,\ldots,PL_k\}$ corresponding to $\{FTQ_1,\ldots,FTQ_k\}$ are received by P_j, The query engine at P_j shall perform intersection or union operation on $\{PL_1,\ldots,PL_k\}$ to result in the Relevant Peer id List (denoted as $RPL=(RPL_1,\ldots,RPL_n)$) of the FTQs.

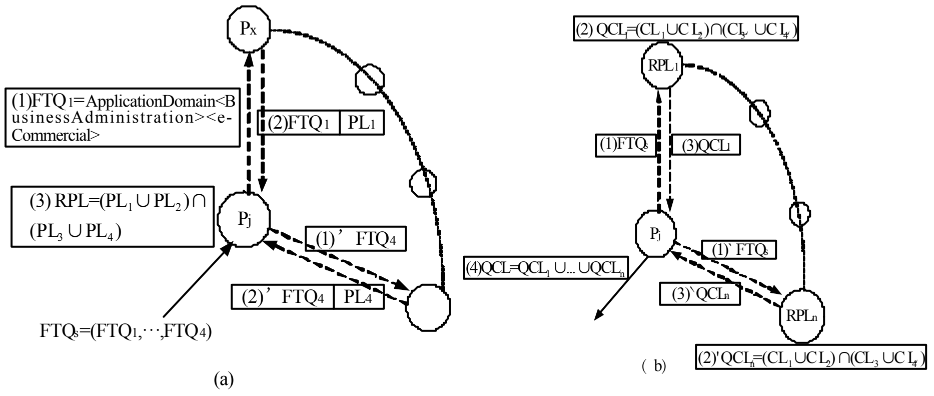
Stage Ⅱ:

* For each relevant peer node P_y in RPL, The submitting peer node P_j sends it FTQs and waits for the resulting components.

* Upon receiving the FTQs, the query engine at P_y is responsible for searching for Local Dictionary (LD) to determine all Component id List $\{CL_1,\ldots,CL_k\}$ corresponding to $\{FTQ_1,\ldots,FTQ_k\}$, and then for performing intersection or union operation on $\{CL_1,\ldots,CL_k\}$ to produce the Qualifying Component id List (QCL) of the FTQs.

* Once the relevant reusable components are found based on the resulting component id list, P_y sends them to submitting node P_j.

Let us consider Example 2.1 again. Figure 4.1 illustrates the two stages of processing reusable components using DPD and LD. The first stage is shown in Figure 4.1a, while Figure 4.1b presents the second stage.



Px
(1)FTQ1=ApplicationDomain<BusinessAdministration><e-Commercial>
(2)FTQ1 | PL1
(3) RPL=(PL1 ∪ PL2) ∩ (PL3 ∪ PL4)
Pj
(1)' FTQ4
(2)' FTQ4 | PL4
FTQs=(FTQ1,···,FTQ4)
(a)
(2) QCL1=(CL1 ∪ CL2) ∩ (CL3 ∪ CL4)
RPL1
(1)FTQs
(3)QCL1
Pj
(1)` FTQs
(4)QCL=QCL1 ∪... ∪QCLn
(3)`QCLn
RPLn
(2)'QCLn=(CL1 ∪ CL2) ∩ (CL3 ∪ CL4)
(b)

Fig. 4.1. Reusable component query example using DPD and LD

5 Evolution of PeeRCR System

P2P network, the infrastructure of the PeeRCR, is a large distributed environment with dynamic and ad hoc characteristics. In other words, the peer nodes over P2P network have a short (maybe last for a few hours) and arbitrary (depending on the node owner) lifetime [23]. Nodes can arrive or depart at will. This section describes how the PeeRCR system deal with the evolution of network when nodes join, leave and update their sharable reusable components. As suggested above, Chord protocol is used as the experimental substrate of the PeeRCR system. Therefore, the evolution of the PeeRCR system is defined by Chord protocol.

5.1 Arrival of Nodes

In Section 3.1, we have outlined the model of Local Dictionary indexing those reusable components his owner would like to share to other peer nodes. For each new arriving node P_n, it is up to the node to construct Local Dictionary LD=$\{FT_i, Com_{j,k}\}$ of reusable components retrieve-able by the nodes already in the PeeRCR system. The process of node inserting into the PeeRCR system involves three main steps.

In the first step, the new arriving node P_n sends join request to any node P_c already in the PeeRCR system (as shown in Figure 5.1a). Based on the Chord protocol, P_n finds its successor P_s, in which the DPD $\{FT_j, Peer_{j,k}\}$ stores, in the identifier ring.

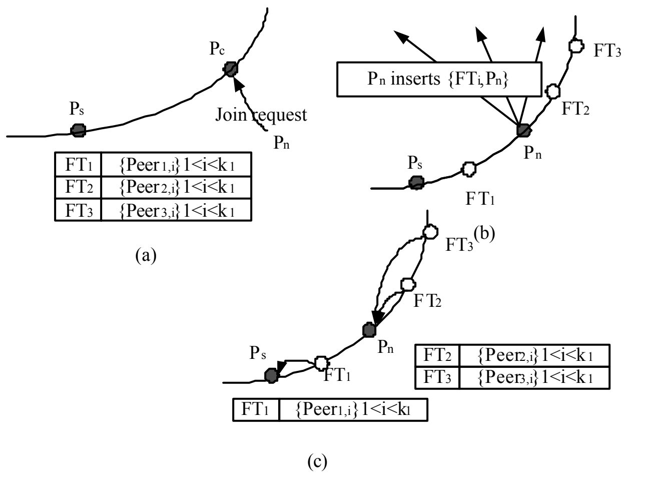


Fig. 5.1. Node P_n arriving at the PeeRCR system

In the second step, P_n injects each tuple (FT_i, N_n) into the PeeRCR system. It is up to the Chord protocol to determine which nodes receive the tuples (as shown in Figure 5.1 b). Accordingly, the receiving nodes will update its DPD.

In the last step, now that P_n becomes part of the Chord identifier ring, it is the responsibility of P_n for hosing parts of peer dictionary already in the network. The Chord protocol assigns DPD containing the keywords FT_i for which P_n is the successor in the identifier ring (as shown in Figure 5.1c).

5.2 Node Departure

When a node departs, the reusable components it stores become unavailable by the rest of peer nodes in the PeeRCR system. In our implementation of the PeeRCR, two steps the departing nodes, says P_d, should follows. The first step is to hand over the DPD resided at the departing node P_d to its successor node according to the Chord protocol. In the second step, P_d notifies the nodes that hold the peer index of P_d with the message about its unavailable. To find the nodes that currently maintain the peer index of P_d, P_d hashes the keywords it inserted into the system during joining to get the keys.

5.3 Update of Reusable Components at Nodes

Possibly, the peer nodes in the PeeRCR want to add new artifact sharable to other peers, or to drop components from sharable set. Moreover, they can make modifications on the reusable components, such as adding or delete interfaces, methods, and attributes of artifacts. All these changes on reusable components indicate that peer nodes perform the update operations on reusable components. In this subsection, we discuss how DPD, Local Dictionary evolve when node performs the update of reusable components. In the PeeRCR system, it is a specification that any modification on reusable components e.g. additions of interfaces, methods, attributes is regarded as creation of new reusable components. Therefore, we shall mainly discuss the two cases: new component inserting and outdated component deleting. When a peer node inserts new sharable components into the system, it first updates the Local Dictionary locally, and then sends update request and inject the peer indices related to new components, then the Chord protocol takes the similar way as the second step of node arrival (as shown in Figure 5.1b) to determine which nodes that shall hold the peer index. When a peer node decides to drop components from sharable lists, it just updates its local dictionary locally, and performs no modification of DPD for simplicity.

6 Experiments

We have implemented a prototype PeeRCR system with the features discussed in the previous sections. The entire system is written in Java. After installing the PeeRCR software, a node will become a PeeRCR node by performing node's joining operation discussed in Section 5.1. In this section, we present the res ults of experiments conducted to evaluate PeeRCR's performance. Section 6.1 describes the experiment setup, including the experimental network environment, the reusable component set and so on. The performance metrics are introduced in Section 6.2. Finally, Section 6.3 analyses the experimental results and evaluates the performance of the PeeRCR.

6.1 Experimental Design

Our experimental environment consists of 30 PCs with Intel PentiumⅣ 1.8GHz to 1.4GHz CPU and 523MB RAM. Each PC runs Windows 2000 operation system. The PeeRCR system is installed on every node. The machines are fully dedicated when we are conducting the experiments.

In our experiments, we choose a large amount of classes and a relatively small number of actual components such as COM, JavaBean as raw component set. The reasons for such design involve the trade-off between feasibility and effectiveness. In order to design classes, we choose about 20 familiar application domains, e.g. mathematics, commerce, finance, business. The number of self-designed classes is about 1500. And we select 300 classes from Microsoft's MFC (Microsoft Foundation Classes), Borland's VCL (Visual Class Library). The last about 100 raw components are actual COM, JavaBean components. The entire number of raw component used in the experiment is about 2000, and each component is accompanied with a 5-facet classification document described in Section 2.1. All these components are distributed among 30 nodes over the network to ensure that each node maintains about 80 raw components. The distribution allows nodes have part of same components.

With reusable component retrieval, the gap between problem statement (a requirement) and solution description (a specification) is not only terminological, but also conceptual [24]. Therefore, it is necessary to establish a controlled environment to query reusable components. We pre-design an initial query set of more than 300 FTQs. The subjects randomly pick variable number of FTQs from the set as queries to submit.

In our experiments, we have also implemented other three different schemes of reusable component repository, namely, C/S architecture, Gnutella and Completion. The performance of PeeRCR is compared against these three schemes.

1. **C/S architecture** - In C/S architecture, we store all about 2000 raw component in one PC, so called as the Server. Other 29 PCs act as clients to retrieval components from the Server. For C/S architecture, only *Local Dictionary* (**LD**) of these 2000 components are required to construct for the lack of the concept of peer.
2. **Gnutella** - This network topology has one common point with C/S architecture: no peer dictionary. In Gnutella, raw components are distribute-stored in peer nodes. It is required to construct *Local Dictionary* (**LD**) for the reusable components it owns. The Gnutella topology was generated to obey the power laws [25]. On the average, every node has 4 neighbors. The average distance between any two nodes in the network is 3.5 hops. In our experiment, we don't set any TTL on query.
3. **Completion** - In this scheme, each node holds one copy of the whole *peer dictionary*.
4. **PeeRCR** - In our P2P reusable component repository system, we apply Chord protocol [12] to partition and distributed the peer dictionary among nodes (as discussed in Section 3.2).

In addition, we also compare the ability of dealing with system evaluation of PeeRCR against Completion.

6.2 Performance Metrics

In this section, we present two metrics used to measure the efficiency and effectiveness of different scheme system. The efficiency deals with the performance issue, and the effectiveness deals with the quality of results. Two metrics, which are used to measure both querying speed and querying quality, are defined as follows.

1. **Query Response Time** - This metric measures the speed of query processing. It is defined as the time between submission of a query and when the first result is received.
2. **Recall** - This metric measures the number of relevant reusable component by a query to the total number of relevant components in the entire system. Both numbers are determined offline. We define the relevant component as the retrieved components with which a developer is able to solve a problem at hand.

6.3 Experimental Results

We conduct a set of experiments for each scheme system. The number of queries of reusable components submitted by a peer node simultaneously varies from 1 to 10.

The queries are all selected randomly from the query pool. Each subject on peer nodes performs 4 trials for each query group and calculates the average as the experiment result.

6.3.1 On the Query Response Time

Figure 6.1 shows the average query response time. As shown in figure, with the number of query submitted simultaneously increasing, the query response time of C/S and Gnutella mounts up quickly. This is because C/S and Gnutella both suffer from the scalability problem when a large amount of queries submitted: C/S adopts one point server strategy, while Gnutella adopts flooding-based routing strategy. On the other hand, the PeeRCR outperform C/S and Gnutella except for the first few query groups, and are close to Completion that is the best case since every node has a copy of the complete peer dictionary.

6.3.2 On the Recall

Figure 6.2 illustrates the recall of component query. As the number of query submitted simultaneously increases, the recall of C/S and Gnutella drops quickly, which indicates that a relatively large proportion of queries can't be successfully processed. The PeeRCR method returns more relevant results than C/S and Gnutella because more queries are successfully processed, and is close to Completion method.

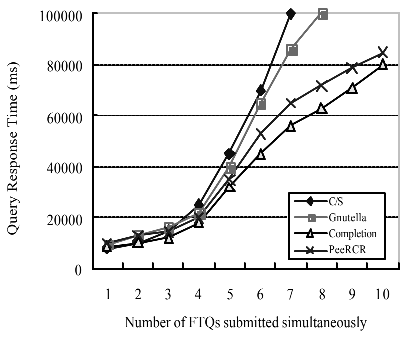


Fig. 6.1. Query Response Time

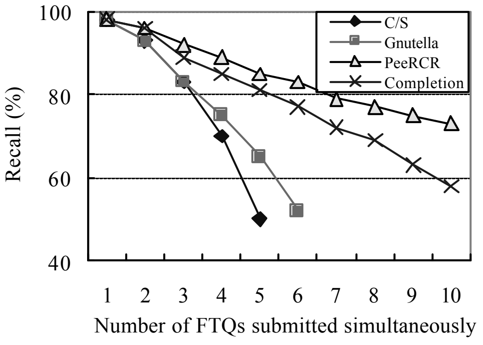


Fig. 6.2. Query Recall

6.3.3 On the Effect of Scaling

In addition, we also conduct an experiment to verify the effect of network evolution on the system performance. We first run 15 PCs as original network, and then add 5 nodes into the network at minute 0, 5, 10. The new peer node has the same number of reusable components resided at it. At each original PC, the subject submits 5 FTQs simultaneously, and no FTQs are submitted at new nodes. Figure 6.3 shows the average query response time of the entire network over the period. From the figure, we can see that the Completion strategy has the higher peak value because the message exchange and update of *Peer Dictionary* among nodes after the joining of new nodes delay the user queries. The PeeRCR has the lower peak value since Chord method just has to redistribute some part of DPD to the new nodes according to the hash function.

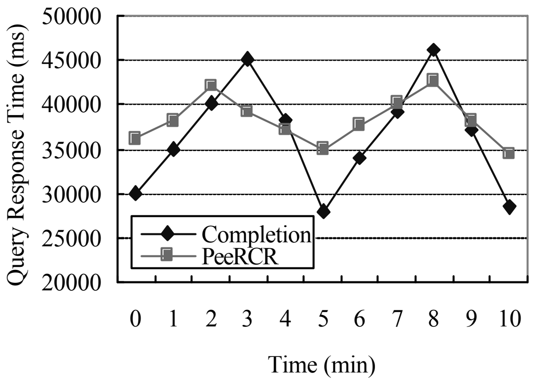


Fig. 6.3. Query Response Time when network evolution

7 Conclusions and future work

In this paper, we have presented an original distributed P2P-based reusable component repository system called PeeRCR. In the PeeRCR, raw reusable components are classified by multi-faceted scheme. In addition, two kinds of indexing mechanics, *Local Dictionary* (**LD**) and *Peer Dictionary*, are introduced. The PeeRCR applies Chord protocol to partition and distribute the Peer Dictionary among nodes, which generates *DHT-based Peer Dictionary* (**DPD**). Results from the experiments on a prototype PeeRCR system demonstrates that our P2P-based reusable component repository system is a promising distributed and scalable system for sharing reusable components in a large distributed environment.

Because the PeeRCR makes a tentative study of building a reusable component repository on P2P network, there are still a lot of jobs remaining for future research. We plan to extent this work in several directions. First, our current prototype system only provides simple component retrieval mechanism. We plan to adopt advanced information retrieval methods for supporting more complex reusable components query. Second, we plan to further study the strategy of distributing Peer Dictionary to obtain better performance. Finally, we plan to evaluate the performance of the PeeRCR with actual reusable components.

References

1. I. Jacobson, M. Griss, P. Jansson: Software Reuse: Architecture, Process and Organization for Business Success. Addison-Wesley, MA (1997).

2. A. Mili, R. Mili, R.T. Mittermeir: A Survey of Software Reuse Libraries. In: W. Frakes (ed.): Systematic Software Reuse (1998) 317~347.
3. STARS Technique Committee: Asset Library Open Architecture Framework (Version 1.2). Information Technology Report, STARS-TC-04041/001/02 (1992).
4. G. Sindre, R. Conradi, E. Karlsson: The REBOOT Approach to Software Reuse. Journal of Systems and Software, Vol. 30 (1995) 201~212.
5. J.M. Morel, J. Faget: The REBOOT Environment. In proceeding of 2nd International Workshop on Software Reusability (1993) 80~88.
6. R.C. Seacord, S.A. Hissam, K.C. Wallnau: Agora: a Search Engine for Software Components. Technical Report, CMU/SEI-98-TR-011 (1998).
7. K. Li, L. Guo, H. Mei, F. Yang: An Overview of JB (Jade Bird) Component Library System JBCL. In proceeding of the 24th International Conference TOOLS Asia, (1997).
8. W.S. Ng, B.C. Ooi, K.L. Tan, A.Y. Zhou: PeerDB: A P2P-Based System for Distributed Data Sharing. In Proceedings of the 19th International Conference on Data Engineering (2003).
9. Napster. http://www.napster.com.
10. Gnutella. http:// www.gnutella.com.
11. S. Ratnasamy, P. Francis, M. Handley, R. Karp, S. Shenker: A scalable Content Addressable Networks. ACM SIGCOM (2001).
12. I. Stoica, R. Karger, M.F. Kaashoek, and H. Balakrishnan: Chord: A scalable peer-to-peer lookup service for internet applications. In Proceeding of SIGCOMM, 2001.
13. A. Rowstron, P. Druschel: Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. In proceeding of 18th IFIP/ACM Intl. Conf. on Distributed Systems Platforms, 2001.
14. C. Szyperski: Component Software, Addison-Wesley. 1998.
15. Bachman, et al: Technical Concepts of Component-Based Software Engineering Technical Report, CMU/SEI-2000-TR-008, 2000.
16. K.C. Wallnau, S.A. Zahedi, R.C. Seacort: Building Systems from Commercial Components. Addison-Wesley, 2002.
17. W.B. Frake, T. Pole: An Empirical Study of Representation Methods for Reusable Software Components. IEEE Transactions on Software Engineering, 1994, 20(8): 617~630.
18. T. Isakowita, R. J. Kauffman: Supporting Search for Reusable Software Objects. IEEE Transaction on Software Engineering, 1996, 22(6): 407~423.
19. R. Prieto-Diaz, P. Freeman: Classifying Software for reusability. IEEE Software Journal (1987): 6~16.
20. J. Sametinger. Software engineering with reusable components. Springer-Verlag, 1997.
21. G. Salton, M.J. McGill: Introduction to Modern Information Retrieval. McGraw Hill, New York, 1983.
22. F. Dabek, M.F. Kaashock, D. Karger, R. Morris, I. Stoica: Wide-Area Cooperative Storage with CFS, SOSP2001.
23. P. Ganesan, M. Bawa, H. Garcia-Molina: Online Balancing of Range-Partitioned Data with Application to Peer-to-Peer Systems. In proceedings of the 30th VLDB Conference, Toronto, Canada, 2004: 444~455.
24. H. Mill, E. Ah-Ki, R. Godin, H. Mchieck: An Experiment in Software Component Retrieval. Information and Software Technology, 2003, 45(10):1~17.
25. C.R. Palmer, J.G. Steffan: Generating Network Topologies That Obey Power Laws. IEEE Globecom2000, 2000.