An Adaptive Agent-based Network for Distributed Component Repositories

Yunjiao Xue, Leqiu Qian, Ruzhi Xu[1], Bin Tang, and Xin Peng
Department of Computer Science and Engineering, Fudan University, Shanghai, China
School of Computer Engineering, Shandong University of Finance, Jinan, Shandong, China
yjxue@fudan.edu.cn; lqqian@fudan.edu.cn; rzxu@fudan.edu.cn; tangbin@fudan.edu.cn; pengxin@fudan.edu.cn

Abstract

The application of component repository has become more and more popular in the practice of component-based software development. Most of the repositories are physically isolated and vary in the classification and specification mechanisms, forming a distributed and heterogeneous network, which hinders the share and reusability of component resource. Therefore, it is important to integrate the distributed component repositories into a logically unified architecture. This paper adopts the intelligent agent-based technique to achieve the adaptive and extensible integration. The network architecture and the structural design of the agents are proposed. Some key issues in the design are discussed further.

Keywords: Adaptive Network, Agent-based, Component Repository, Distributed, Integrate

1. Introduction

The reusable software component repository, which provides effective management to reusable software components in specification, classification, storage and retrieval, is an important infrastructure for component-based software development (CBSD). With the growing of the CBSD, more and more software companies are building their own component repositories, which are physically isolated. They expect their components to be retrieved and reused by as many users as possible. However, most component users are familiar with only a few public component repositories, and they would not be willing to access all the companies' private component repositories one by one to find their preferred results. Therefore, it is crucial to integrate the distributed component repositories and provide a unified interface for users to retrieve all the repositories.

An intuitive solution is to gather all the components to a centralized repository, but it is not applicable due to the following three reasons:

- For the sake of copyright protection, most of the companies would not be willing to submit the information about their components to the centralized public component repository.
- The distributed repositories employ various component specification and retrieval mechanisms, so it is costly and complicated to convert them into a unified one, which is essential in the centralized repository.
- A large number of retrieval requests will be put on the only one host running the centralized repository, which could become the bottleneck of the whole system.

Now, the problem can be defined as: can we make up an alternative integration policy so that we could construct a logically integrated system covering the physically isolated repositories and provide a unified access interface to these resources. Moreover, the solution should not collect all the component information to a centralized component repository.

In this paper, we tackle the problem using intelligent agent-based techniques. We present an agent-based network for the distributed repositories. In the integrated architecture, there is a request server that receives the retrieval requests and publishes the results. An agent runs in each member component repository, which actively contacts the request server to fetch the retrieval requests. The agent performs the request in local repository as well as forwards it to the agents running in other component repositories, and then gathers the results from these repositories and returns them to users.

Our solution integrates the distributed component repository without gathering the resources into a central server, thus avoiding the high cost of integration and the heavy workload on central server while working. Moreover, the member component repositories do not need to share the component information explicitly, so their privacy is

Proceedings of the First International Multi-Symposiums on Computer and Computational Sciences (IMSCCS'06)
0-7695-2581-4/06 $20.00 © 2006 IEEE

preserved. This agent-based integration policy is also flexible due to the autonomy and intelligence of agent.

The rest of the paper is organized as follows. In Section 2, we provide some essential background and review the related work. Section 3 presents an overview of the agent-based network architecture. Section 4 describes the architectures of the two different kinds of agents in the network and discusses some crucial issues in the design of the agents, while Section 5 gives some results in the implementation of the agent-based network. Section 6 concludes the paper with a discussion about our future work.

2. Related Work

The major topics in component repository construction include defining types of reusable software components, defining classification methods, and defining storage, search, and retrieval mechanisms [1]. Many researches have touched these issues. The typical projects include REBOOT (Reuse Based on Object-Oriented Techniques)[2] and PCTE (Portable Common Tool Environment) in Europe [3], STARS program in USA, which proposes a reference model of component repository and an instance named ALOAF (Asset Library Open Architecture Framework) [4]. These programs focus on the component models and specification paradigms. Other projects, such as ComponentRank [5], CodeBroker [6], and the system of N. Ohsugi et al. [7] dedicate to the exploration of component retrieval in a repository system. S. Tangsripairoj et al. introduce the concept of SOM (Self-Organizing Map) into the construction of component repositories, which is an interesting attempt in this field [1].

The theory and technology of intelligent agents have been studied by many projects and got widely application [8, 9, 10, 11, 12]. Because of the fundamental properties of an intelligent agent including autonomy, social ability, reactivity, and pro-activeness, agents could help meet the growing need for more functional, flexible, and personal computing and telecommunications systems.

The agent techniques have been applied in many distributed computing fields [13], such as data mining [14], agent-based middleware [15] and information retrieval with distribute data sources [16], etc. These works brings much valuable reference to our research on distributed component repositories.

3.1 Agent-based Network

3.1 Architecture of the Network

The agent-based network is a communication and collaboration model of the integration solution. It consists of a request server and multiple member component repositories (running on other hosts). The request server receives the retrieval requests from users and returns them the results. Each member component repository R_i has an agent A_i running on it, and the agent initiatively contacts the request server to fetch requests when it is free. If A_i gets the request, it executes the request locally as well as passes them to other agents in the network, and then gathers the results and returns them to the request server. In the request server, we employ an agent to maintain the route information of the member component repositories and to discover the "preference" of each repository (that is, the general information about what particular categories of components are stored in the repository. Section 3.2 discusses "preference" in detail). By running in this way, any agent in the network can collaborate with others, and the users can "retrieve" multiple repositories by accessing the request server only, even though they are not aware of the existence of these repositories.

Figure 1 depicts a typical agent-based network consisting of a request server and 6 member component repositories. In each node N_i of the network, there is an agent A_i running in the repository R_i. A_i is in charge of issuing the retrieval request to R_i, and communicating with other agents in other nodes. The request server interacts with users, and buffers the records of retrieval requests and results. Agent *MA* runs on the request server, maintaining the structural information of the network and the preference of each member component repository.

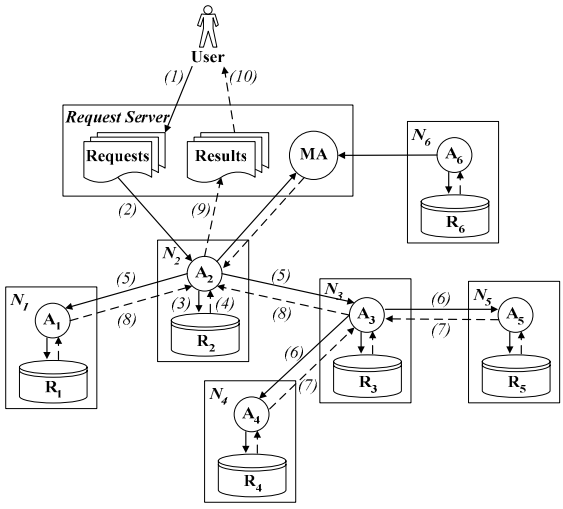

Figure 1. A typical agent-based network.

As a typical execution flow (shown in Table 1), a user sends request *req* to the request server; the agents will initiatively access the server to get the request. Assume A_2 gets request *req*, first it will perform *req* in its local component repository R_2, as well as forwards the request to its neighbors (that is, a set of nodes more reachable than other nodes in the network from A_2. The technical details of

neighbors are discussed in section 4.2.2). The neighbors receiving the request will perform it in the local component repositories and re-forward the request to their neighbors, and then gathers the result to return to A_2. After receiving the results, A_2 will submit them to the request server. In the process of answering *req*, A_2 acts as the *leader* and other agents are *cooperators*.

Table 1. The actions in the network

No.	*Actions*
(1)	User issues a retrieval request;
(2)	A_2 gets the request;
(3)	A_2 performs the request in R_2;
(4)	A_2 gets the result in R_2;
(5)	A_2 forwards the request to its neighbors;
(6)	The neighbors of A_2 perform the request locally and forward it to their neighbors
(7)	The neighbors gather the results;
(8)	The results are returned to A_2;
(9)	A_2 return the results to the request server
(10)	The Request Server publishes the results;

Each of the agents will actively access the request server to get the unanswered requests when it is free. Thus all the agents will share the workload of the entire network, which is a crucial topic in the distributed environment. To avoid different agents getting the same one request, the agent that first accesses the request changes a flag of it indicating that it has been accessed. Other agents will omit it when they scan the requests queue. The answered and expired requests will be archived to save space of the request queue.

3.2 Preference

In the network, a member component repository may focus on certain kinds of components, which is defined as the *preference* of it. According to the preference, sometimes a request only needs to be performed in the repositories possessing the preferred kinds of components, which can avoid unnecessary operations. Hence, we employ *MA* to maintain a list of preferences of all the repositories. For example, if repository R_3 only owns the components based on ***Windows OS***, then any request for ***Linux-based*** components would not be performed in A_3. But the request is still forwarded to A_3, and A_3 will propagate it to its neighbors. How to identify the preference of a component repository is discussed in section 4.1.2.

Instead of contacting *MA* every time when they need the route and preference information, the agents may cache them locally. *MA* will periodically update the information and push it to other agents.

If a new component repository wants to join the network (for example, the repository R_6 in Figure 1), first an agent needs to be deployed on the host of R_6 and interacts with the repository to collect the necessary information (such as, the route information and the specification & classification mechanism of the repository), and then contacts agent *MA* on the request server to register itself in the network.

3.3 Neighbors

In order to avoid flooding the network with messages, any agent getting the request only forwards it to the most reachable agents in the network (that is, the agents that are most likely to respond in a certain time period), which are defined as the neighbors of the agent. Moreover, the neighbors who receive the request may perform it and forward it to their neighbors. As shown in Figure 1, assuming that A_2 gets a request, it will perform the request and propagate it to its neighbor A_1 and A_3. Since A_1 does not have any neighbor, the propagation is terminated. A_3 will further propagate the request to its neighbors A_4 and A_5. The propagation ends when all the neighbors of the agent have already received the request, or the request has been propagated the predefined maximum times.

The details of neighbor-determination will be discussed in section 4.1.2

4.1 Agent Architecture

The agent-based network consists of a set of agents running automatically and cooperatively. Figure 2 depicts the architecture of an agent. There are four main parts in an agent: *message port*, *message queue*, *evaluator*, and *actuator*. The agent gets the net and local messages through *message receiving ports* either passively or actively, and stores them in a *message queue*. The messages are parsed by an *evaluator* that determines what actions should be taken as response to the messages. The actions are issued by the *actuator* in forms of net and local messages.

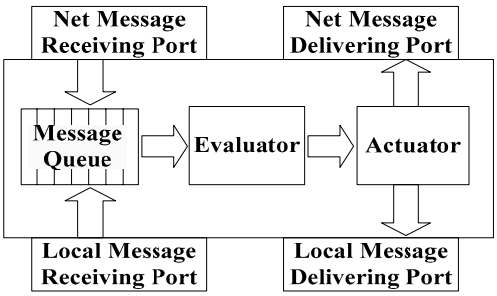

Figure 2. Architecture of an agent.

The agents communicate with each other using a KQML-like language. KQML is a practical agent communication language (ACL), which consists of tagged messages [17]. For example, an ACL message sent by the agent on a newly joined node to *MA* to register itself in the network is as follows:

```
<message>
  <msg type>type_1</msg type>
  <function>registration</function>
  <receiver>MA</receiver>
  <receiver ip>221.225.2.78</ receiver ip >
  <host ip>221.225.2.6</host ip>
  <host name>shcr</host name>
  <specification type>facet</specification type>
  <preference>
      <domain>financial</domain>
      <environment>
          <OS>Windows</OS>
      </environment>
  </preference>
</message>
```

Figure 3. A message of registration.

In the message, the tag "msg type" and "function" identify the utility of the message, while "receiver" and "receiver ip" specify the destination of the message. The other tags depict the basic information of the new component repository.

Generally, there are two kinds of agents in the network: *Network Monitor Agent* and *Executing Agent*. In the following sections, we will discuss the behaviors of these two kinds of agents.

4.1 Network Monitor Agent

Network Monitor Agent is in charge of managing the agents in the network. The agent *MA* in Figure 1 is an instance of it. The architecture of a network monitor agent is shown in Fig. 4.

There are 3 kinds of messages passed to the network monitor agent (as message (1)(2)(3) shown in Figure 4):

(1) ***Registration Message.*** This message is sent to the network monitor agent by an agent that expects to join the network. It contains the basic information about the newcomer repository, such as host IP, specification type and so on.

(2) ***Unregistration Message.*** This message is issued by the agent that is about to quit the network, which contains the host IP and the agent ID of the leaving node.

(3) ***Preference Updating Message.*** To detect the preference changes of the component repository, the network monitor agent periodically contacts other agents in the network. If the preference of a repository has changed, the agent in the corresponding repository would send a preference-updating message containing the new preference information to the network monitor agent.

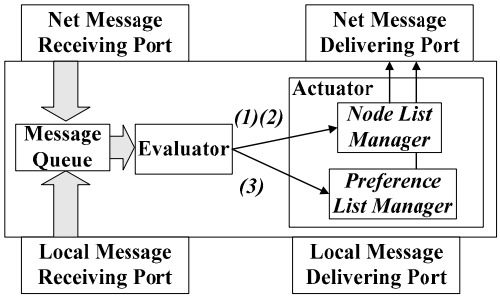

Figure 4. Architecture of the network monitor agent.

4.1.1 Agent Registration and Unregistration. The evaluator of the agent parses the messages and determines the actions in the actuator according to the types of the messages. If a registration or unregistration message has been received, the network monitor agent would modify the node list and propagate the modification to all other agents in the network. A typical node list is shown as follows.

```
<node list>
  <node>
      <node id>1</node id>
      <host ip>221.225.2.1</host ip>
      <host name>shcr</host name>
      <specification type>facet</specification type>
  </node>
  <node>
    <node id>2</node id>
    <host ip>221.225.2.6</host ip>
    <host name>cncr</host name>
    <specification type>keyword</specification type>
  </node>
  ...
</node list>
```

Figure 5. A typical node list.

4.1.2 Preference Management. We adopt the facet method to specify the preference of the member component repositories. Basically we select the two most important facets "Domain" and "Environment" to depict the preference of repositories as shown in Figure 6.

The facet "Domain" is used to specify the component's area of usage [18]. For example, the term "Finance" of this facet is the indication for a category of components that are used in financial applications. The facet "Environment" specifies the environment where the components run. For example, "Windows" depicts a category of components that only run in Windows OS.

Assume the repository R_1 in node N_1 possesses the components used for financial applications. Moreover, these components only run in Windows OS. In this case, the preference is depicted as follows:

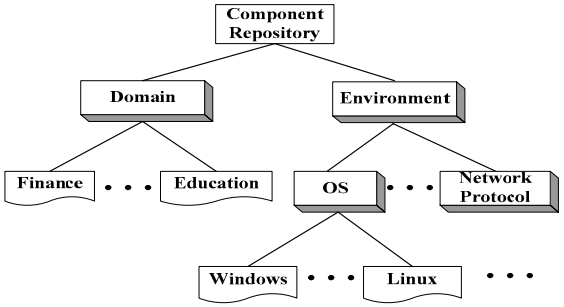

Figure 6. The facet hierarchy of preference.

```
<node id>1</node id>
<host ip>221.225.2.1</host ip>
<preference>
      <domain>Finance</domain>
      <environment>
 <OS>Windows</OS>
      </environment>
</preference>
```

The preference of a repository can be obtained by performing a local retrieval, such as finding the terms of preferred facets from its database. Sometimes manual interference is required.

4.2 Executing Agent

Executing agents, the main working unit in the network, run on the member component repositories in the network (such as A_1 in Figure 1). The architecture of executing agent is shown in Figure 7.

There are 5 kinds of messages acceptable for an executing agent (as message (1)(2)(3)(4)(5) shown in Figure 7):

(1) ***Asking For Preference Message.*** The network monitor agent sends this message to the executing agents to get the preference of the repository. The executing agent responds to this message by issuing a local retrieval and returns the preference information to the network monitor agent if its preference has been changed.

(2) ***Request Message***. The executing agent contacts the request server when it's free. If there is an unanswered request, the server will send the agent a *request message*. Typically, the request is specified by several terms within some facets. If the component repository is not specified under facet mechanism, the internal specifications will be converted into tree-like structures that can be matched with the request in the "request performer". The conversion methods will be discussed in section 4.2.1. And then the request will be passed to the "Request Propagator" to be forwarded to the neighbors of the agent. The neighbor information is managed by "Neighbor Manager".

(3) ***Cooperating Message.*** The cooperating message includes the request forwarded by other executing agents and the route information of these agents. After receiving this message, the executing agent will perform it in local repository and propagate it to its neighbors.

(4) ***Network Updating Message.*** When there is repository entering or leaving the network, the network monitor agent will send a network-updating message to all the executing agents specifying the active nodes and their routing information. This information is useful for deciding the neighbors of the agents.

(5) ***Preference Updating Message.*** If the network monitor agent detects the preference changing of some repository, it will inform all the executing agents with this message. The executing agents will take this message into consideration when they decide the target repositories to forward requests.

There are two crucial issues in the design of the executing agent that are introduced in the following sections.

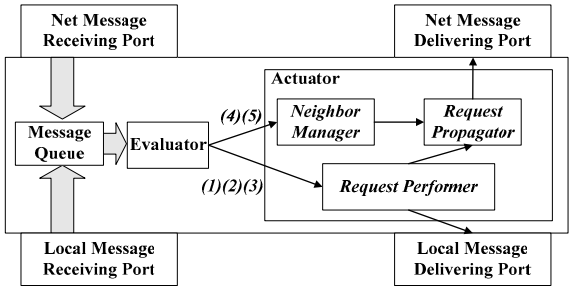

Figure 7. Architecture of the network monitor agent.

4.2.1 Request Performing. In the distributed environment, different specifying or classifying models (jointly called representation model) are employed by the enterprises severally. We need a mechanism to implement the inter-conversion of component representation, and to make the retrieval crossing repositories feasible. The following is the problem definition.

Problem Definition: Given a retrieval request R (in a format with structured semantics to be depicted later), a set of representation models $RM = \{rm_1, \ldots, rm_n\}$, where each rm_i is a kind of component representation model. A set of conversion mapping $CM = \{cm_1, \ldots, cm_n\}$ is to be defined, where each cm_i is a mapping: cm_i: $R \times rm_i \rightarrow Instance_{rmi}$, where $Instance_{rmi}$ is the set of all specification instances of components under the model rm_i.

In practical application, the representation models of components are divided into two major categories: specification and catalog. In the specification models, some key aspects are proposed and the concrete descriptions

under each aspect are given to depict the characteristic of a component. The *3C*, *REBOOT*, and *Facet* model are often used instances. An instance of a specification model forms a tree structure, where the component is at the root node, the aspects are non-leaf nodes, and the depicting words form the leaf nodes.

The catalog model emphasizes on constructing a classification hierarchy system via partitioning the concept domain into more precise sub-domains gradually, and locating a component into a specific position in the hierarchy based on its characteristic. The classification hierarchy also forms a tree structure. Note that we need to add a virtual root node at the top of the structure to make it a "tree".

To derive a conversion mechanism between different models, we change the specification tree by putting the component on below the term nodes, and add a virtual root note at the top to keep it still a tree. So, based on the topological structure, the two models could be unified to a tree structure (called a concept-dividing tree), which is the fundamental of the conversion mechanism. In such a tree each non-leaf node is called a descriptor. A sample concept-dividing tree is shown in Figure 8.

The retrieval requests can also be formatted into trees (facet model adopted in this paper) for two reasons: a) It is comprehensible to guide a use to write the request via the user interface step by step by giving detailed descriptions to each aspect of the potential component that the user is interested in. b) With tree-match algorithm (mentioned later) the retrieval tasks can be accomplished with a fine precision and recall. With the common tree structure acting as a middle form, we change the problem into finding the mappings that convert the component representation instances into the common concept-dividing trees.

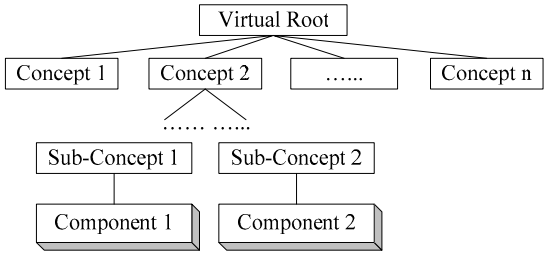

Figure 8. A concept-dividing tree.

To construct unified tree structures for each repository, we define the following concepts:

1. D is the set of descriptors, $D=\{d_1, d_2, ..., d_n\}$, $\forall d_i \in D$ is a descriptor, which may be a facet, a category, or an attribute, etc.

2. DH is the set of descriptor's hierarchy relations, $DH=\{(pd, cd) | pd, cd \in D \wedge pd \lessdot cd\}\}$, where $\lessdot$ means a partial-ordering relation in which pd is the parent descriptor and cd is the child one.

3. C is the set of components, $C=\{c_1, c_2, ..., c_m\}$ $\forall c_i \in C$ is the identifier of a component.

4. DC is the set of component-describing relations, $DC=\{(d, c) | d \in D \wedge c \in C\}$, where d acts as one of the descriptors of c.

It is not difficult to construct D, DH, C, and DC from a repository by gathering the information stored in it. And then we introduce a construction algorithm based on these concepts (As shown in Figure 9). The algorithm can be refined further for better performance.

Algorithm: CDTree (Concept-Dividing Tree)
Input: Concept sets D, DH, C and DC
Output: A set T of concept-dividing trees
Method:

```
1.  T = ∅;
2.  while (there are unused elements in DC )
3.      t = node(virtual root);
4.      //t is a tree variable
5.      // node(n) constructs a tree node with the label n.
6.      for an unused tuple (d, c) ∈DC
7.          find a path p from DH where the consecutive node
            n1, n2, ..., nk satify that (n1, n2), (n2, n3), …, (nk-1, nk)
            ∈DH, and nk-1=d, nk=c;
8.          add p into t;
9.          if part of p overlaps others in t then
10.             merge them;
11.     T := T∪t;
```

Figure 9. Algorithm CDTree.

After the construction is completed, the *request performer* of the agent is prepared to do the retrieval. In our implementation we employ a matching model and corresponding matching algorithm based on the tree matching, which is one of our previous work and introduced in [19]. The details of matching are omitted due to the length limitation.

4.2.2 Neighbor Selecting Policy. When an executing agent gets a request from the request server, it will perform it in its local component repository, and besides, forward the request to other repositories in the network. However, propagating the request to all the agents in the network at one time is not applicable. The reason is that: first, all the requests sending at the same time will become a heavy workload for the network; second, not all the nodes could be connected from the current node. Therefore, we only forward the request to the nodes that is most likely to give the response, and we call these nodes the neighbors of the current node.

We use the following method to find the neighbors. Assume that we want to find k neighbors for agent A_i. First we randomly pick k nodes in the network to be the neighbor of A_i and assign a credit number c_{ij} for each neighbor

(c_{i1}=c_{i2}=…=c_{ik}=C). When getting a request, A_i forwards it to the k neighbors. If the j_{th} neighbor doesn't respond in the required time, the corresponding credit number c_{ij} will be decreased by 1 as the penalty of delay. If any credit number equals to 0, we will remove the corresponding node from the neighbor list and add another one randomly. All credit numbers will be reset to C if the neighbor list changes.

The algorithm of answering the request is shown in Figure 10. First, the leader agent A_i will perform the request Q locally (line 2); then in line 3 to line 9, A_i forwards the request to its neighbors and changes their credit number if necessary. In line 10 to 15, A_i checks whether some neighbors should be removed from the neighbor list due to the low credit numbers.

Algorithm: ARProcessing (Agent Request Processing)
Input: leader agent A_i, neighbor List $N[A_i]$ of A_i, request Q and user specified credit number C
Output: result R
Method:

1. R=∅;
2. perform Q in A_i;
3. for each agent A_{ij} in list $N[A_i]$
4. if Q is covered by the preference of A_{ij}
5. R'=ARProcessing(A_{ij}, $N[A_{ij}]$, Q) ;
6. if R'=∞
   // A_{ij} doesn't respond in the required time
7. c_{ij}=c_{ij}-1;
8. else
9. R=R+R' ;
10. for each agent A_{ij} in list $N[A_i]$
11. if c_{ij}=0
12. randomly pick A_k to join $N[A_i]$ (A_k∉$N[A_i]$);
13. remove A_{ij} from $N[A_i]$;
14. for each agent A_{ij} in list $N[A_i]$
15. c_{ij}=C;
16. break;

Figure 10. Algorithm APRrocessing.

5. Implementation

The architecture and its implementation described in this paper have been applied in a research project of Shanghai government. In the prototype system we involve several enterprise repositories connecting to the Internet, employ a computer server as the request server, and establish an integrated system that can perform the retrieval requests from users in distributed hosts and combine the returned results into a unified set, finally publishing to the users. Figure 11 shows the integrated view of the system observed from the central server's monitor tool.

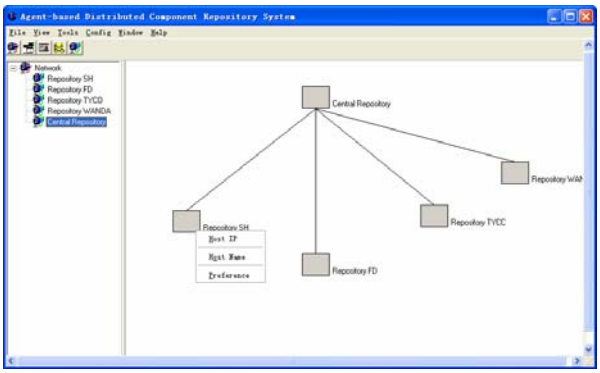

Figure 11. An integrated view of the system.

As shown in Figure 12, we provide a wizard to guide users constructing their retrieval request. The final result is formatted into a concept-dividing tree and appended to the server's request queue.

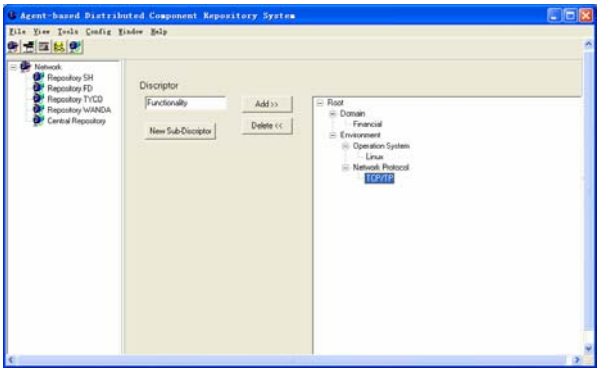

Figure 12. Request construction wizard.

6. Conclusions and Future Work

The integration of distributed component repositories is not fully explored until now. Intelligent agent is an appropriate way to achieve the integration. In this paper we propose an adaptive agent-based network, and provide a detailed design of the agents. With the tentative implementation and application in several practical repositories, we validate its feasibility and effectiveness.

The research issues introduced in this paper are the beginning of an on-going work. Even if the basic results and the main investigation direction have been given, much effort, both theoretical and practical, is needed to achieve a complete solution of the problem. The future work includes the further refinement of the agents to make them usable in a wider range of network. The new retrieval method in the distributed environment is another vital topic to improve the integration.

7. Acknowledgement

This work was supported by the National Natural Science Foundation of China under Grant No. 60473062, National 863 High-Tech Research and Development Program of China under Grant No. 2002AA114010 and No 2004AA113030, and Science and Technology Development Program of Shanghai under Grant No. 025115014 and 04DZ15022.

8. References

[1] S. Tangsripairoj and M. Samadzadeh, "Application of Self-Organizing Maps to Software Repositories in Reuse-Based Software Development," in Proceedings of the 2004 International Conference on Software Engineering Research and Practice (SERP'04), Volume II, pp. 741-747. Las Vegas, Nevada, June 2004.

[2] J. Faget and J. Morel, "The REBOOT Environment," in *Proc. of 2nd International Workshop on Software Reusability (Reuse'93)*, Lucca, Italy, March 1993.

[3] F. Long and E. Morris, "An overview of PCTE: A basis for a portable common tool environment," *Technical Report CMU/SEI-93-TR-1, ESC-TR-93-175, Software Engineering Institute*, Carnegie Mellon University, March 1993.

[4] STARS Technical Committee, "Asset Library Open Architecture Framework: Version 1.2," *Informal Technology Report*, *STARS-TC-04041/001/02*, August 1992.

[5] K. Inoue et al., "Component Rank: Relative Significance Rank for Software Component Search," in *Proceedings of the 25th international conference on software engineering*, Portland, Oregon, USA, May 6-8, 2003.

[6] Y. Yunwen and G. Fischer, "Information delivery in support of learning reusable software components on demand", in *Proceedings of the 7th international conference on intelligent user interfaces*, California, USA, 2002. ACM Press.

[7] N. Ohsugi et al., "Recommendation System for Software Function Discovery," in *Proceedings of the 9th Asis-Pacific software engineering conference*, 2002.

[8] Etzioni, O., Lesh, N., and Segal, R., "Building softbots for UNIX," in *Software Agents -- Papers from the 1994 Spring Symposium (Technical Report SS–94–03)*, pages 9–16. AAAI Press.

[9] M. Roesler and D. Hawkins, "Intelligent agents", *Online*, vol. 18, no. 4 (1994), pp. 18-32.

[10] L. Rosen, MIT Media Lab presents the interface agents symposium: "Intelligent agents in your computer?," *Information Today*, vol. 10, no. 3 (1993), p. 10.

[11] P. Maes, "Agents that reduce work and information overload," *Communications of the ACM*, vol. 37, no. 7 (1994), pp. 30-40.

[12] P. Ciancarini, et al, "Coordinating Multi-Agents Applications on the WWW: A Reference Architecture," *IEEE Transactions on Software Engineering*, 1998, 24(8): 362~375.

[13] D.L. Martin, A.J. Cheyer, and D.B. Moran, "The open agent architecture: a framework for building distributed software systems", *Applied Artificial Intelligence*, 13(1-2), 1999, pp. 91-128.

[14] M. Klusch, S. Lodi, and G. Moro, "Agent-based distributed data mining: The kdec scheme," in *AgentLink*, Vol. 2586 of LNCS. Springer, 2003.

[15] S. Lipperts and A.S. Park, "An agent-based middleware – a solution for terminal and user mobility," *Computer Networks*, 31(19), 1999, pp. 2053-2062.

[16] N.G. Shaw, A. Mian and S.B. Yadav, "A comprehensive agent-based architecture for intelligent information retrieval in a distributed heterogeneous environment," *Decision Support Systems*, 32(4), 2002, pp. 401-415.

[17] T. Finin, R. Fritzson, D. McKay, R. McEntire, "KQML as an agent communication language," in *Proceedings of the third international conference on Information and knowledge management*, Gaithersburg, Maryland, United States, 1994, pp. 456 - 463.

[18] V. Lucena, "Facet-Based Classification Scheme for Industrial Automation Software Components," in *Proceedings of Sixth International Workshop on Component-Oriented Programming (WCOP 2001)*, Budapest, Hungary, 2001.

[19] Y. Wang, Y. Xue, Y. Zhang, S. Zhu, and L. Qian, "A Matching Model for Software Component Classified in Faceted Scheme," *Journal of Software*, Vol.14, No.3, P.401-408, March 2003.