A Model of Web Service Discovery Based on BalanceChord

Li CHEN1,†, Zilin SONG1, Shiming ZHENG1, Wenjie SUN2, Zhanfeng WANG1

1 Institute of Command Automation, PLA University of Science and Technology, Nanjing 210007, China
2 CSMTC Yuanwang-6 Tracking Ship, Jiangyin 214431, China

Abstract

The research of Web service discovery can be distinguished into two layers: service discovery model and service discovery algorithm. In this paper a kind of services discovery model named WSDBC (Web Service Discovery based on BalanceChord) is proposed in order to extend the Web services discovery scope and improve discovery efficiency. In order to achieve load balancing between different nodes in WSDBC model, node join-in algorithm and self-balancing algorithm are proposed. When a new node joins in the BalanceChord, it must get the network load information and set its identifier in order to share the load of overloaded node. By the self-balancing algorithm a node would periodically compare its load with its neighbors and share its load clockwise. A simulative system ChordSim is designed and implemented. Experimental results show this model and algorithms have a good performance.

Keywords: BalanceChord; Web Service Discovery; NACIS (North American Industry Classification System).

1. Introduction

Web Service is a “self-inclusive, self-descriptive and modular application program” which has no relationship with platform [1, 2]. Web Service also can interoperate with other compatible application programs [3]. Discovery of Web services is becoming a hot topic as Web services become more widespread. The research of Web service discovery can be distinguished into two layers: service discovery model and service discovery algorithm. Service discovery model focuses on the relationship among participators in service discovery and the difference in the architecture of service discovery. On the other hand, service discovery algorithm focuses on the matching between user’s request and advertised service in the service registry. Service discovery algorithm is constructed on the service discovery model and constrained by the service discovery model.

Much of the work on Web services discovery is based on centralized registries such as UDDI or the OWL-S Matchmaker [4]. An architecture based on a centralized registry assumes that every Web service coming on line advertises its existence and its capabilities/functionalities with the registry; and that every service requester contacts the registry to discover the most appropriate Web service and gather information about it. Centralized registries are effective since they guarantee discovery of services that have registered. On the other hand, they suffer from the traditional problems of centralized systems, namely they are performance bottlenecks and single points of failure.

In order to overcome the problems of centralized registry and extend the Web services discovery scope, some kinds of methods using P2P technology are proposed [5, 6, 7, 8, 9]. These methods usually connect several service registries and form a service discovery system with large scale. In reference [7] a kind of

† Corresponding author.
Email addresses: ivan_chenli@tom.com (Li CHEN).
Web service discovery model based on unstructured P2P network (Gnutella) was proposed. When users have a service request, the model firstly searches in the neighbor nodes and then broadcasts the service request to the whole network with find flooding. Every node compares the service request with all the registered services in it and returns the result to the request node. The advantage of this model is that the scope of service discovery is expanded greatly. However, this method will take some new problems at the same time. The information of registered service is stored in the nodes disorderly. Every service request will be broadcasted to almost every node in the model. Obviously every user’s request will possess the whole bandwidth and computing resource of the network, the efficiency is suspicious.

Corresponding to the shortage of methods above, in this paper we propose the model named WSDBC (Web Service Discovery based on BalanceChord). We classify services are firstly classified by NACIS (we will mention it in the rest of paper) and then published to nodes by routing algorithm. When a user has a service request, the NAICS category code of the service is obtained. Then we locate to the node which stores the appropriate service through routing algorithm. Because there are similar services stored in the node, we can reduce the bandwidth and computing resource greatly.

The rest of the paper is organized as follows. In Section 2 we introduce the related technology of our work, propose the WSDBC model and the load balancing algorithm of nodes in detail. In Section 3, we present experiments for evaluating our proposed algorithm. The last section presents the conclusions and future work.

2. Web Service Discovery Model Based on BalanceChord

2.1. Related Technology

Chord [10] is a structured P2P network model which is based on consistent hashing. It has good expansibility, robustness, equality of node load and capacity of self-organization. In Chord network, every object has an identifier which is named the Key and every node has an identifier which is usually named ID. The nodes in Chord are constructed to a circle in logic. The node ID and the object Key are mapped into the same space. An object is stored at the node responsible for the relationship between its Key and the node ID.

NAICS [11] (North American Industry Classification System) is usually used to classify the industry product; it also can be used to classify the service. In OWL-S1.1 [12], the ServiceCategory in ServiceProfile defines the classification information of the service. W3C doesn’t appoint the classification standards; just recommends some kinds of standard such as SIC, NAPCS, NAICS. In this paper, we select NAICS 2007 as our standard. NAICS gives every service a category name and a category code, such as figure 1. So a service can be defined as an ordered 3-tuple: \( <SN, CN, C> \) where

\( SN \) represents the service name which corresponds to the ServiceName in ServiceProfile;

\( CN \) represents the category name which corresponds to the CategoryName of ServiceCategory in ServiceProfile;

\( C \) represents category code which corresponds to the Code of ServiceCategory in ServiceProfile.

![Fig. 1 The Classification Information of a Service](image-url)
2.2. WSDBC Model

**Definition 1 (BalanceChord).** BalanceChord is a transformation of Chord. Compared to Chord, the characteristic of BalanceChord is that node ID is composed by the code of NAICS and the quantity of node is limited. The word “balance” has two meanings. Firstly, the distance between nodes is balanceable, i.e. if there are 500 nodes in the BalanceChord network and max size of network is 10000, then the distance between nodes is 200. Secondly, the load of nodes is balanceable. After the construction of BalanceChord, the distance between nodes will not be balanceable with the nodes insertion and leaving, but the load of nodes will always be balanceable.

In Chord network, node ID usually can be provided by hashing information of the node itself and object Key can be provided by hashing the name of object. The node ID and the object Key are mapped into the same space. An object is stored in the first node whose ID is greater or equal to its Key. In this paper a node represents a service registry and the object represents the registered service information. We select NAICS category code space as the space of node ID and the object Key. NAICS category code is composed by six numbers and the last number of code is reserved. Accordingly, the quantity of node in BalanceChord is less than 100000, much smaller than ordinary Chord network. A simple example of BalanceChord is shown in figure 2.

![Fig. 2 An Example of BalanceChord](image)

In BalanceChord model, Web services are classified based on NAICS and taking NAICS category code as the Key. Then the Web service can be published to the service registry which are connected each other through BalanceChord. In this way, similar services are stored in the same node so that the bandwidth and computation resource of the network can be reduced greatly in service discovery.

Now we consider the situation of service discovery. At first according to the user’s request we will get the NAICS code of service request and locate to the node which stores the service information in BalanceChord. Then some relevant service discovery algorithms will be invoked to match the service to service request. At last, the result returns to the user.

Compared with the service discovery model using find flooding, the advantage of WSDBC model is that we can complete node routing and location in $O(\log N)$ ($N$ is the number of nodes in the model) and finally complete the service discovery.

Since the node ID and service Key disperse in our model almost equally, the load of network should be shared by all nodes, that’s the load of nodes should be balanced (sometimes may not, we will discuss this problem below). The problem of capability bottleneck or single point failure can be avoided successfully because there is no special node in the model.

2.3. Load Balancing of Nodes

Because of the distributed environment, the registered service information in nodes is usually different and the accessing frequency of services is also different. There is the load balancing problem in the BalanceChord.

**Definition 2 (Load of Node).** The load of node is the summation of invoking times of all services in the
node. It can be defined as follows:

\[ L_{\text{node}} = \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} \]  

(1)

Where \( L_{\text{node}} \) represents the load of node, \( m \) represents the numbers of service category in the node, \( n \) represents the numbers of service in a category. \( p_{ij} \) represents the times which the service is invoked. At the beginning, \( L_{\text{node}} = 0 \). As time passing by, the load of nodes will be different. In order to improve the load balancing of nodes in the WSDBC model, two kinds of strategy are proposed in this paper: ①new node joins beside the node which is overloaded in order to share the load; ②the load of nodes can be automatic self balancing. Therefore, node join-in algorithm and self-balancing algorithm are proposed in this paper.

2.3.1. New Node Join-in Algorithm

When a new node wants to join in the BalanceChord, it must get the network load information and find the node which has the greatest load. When finding the node with greatest load, other information will be gained at the same time, such as the numbers of service category, the numbers of service in the category, the times which the services are invoked. The new joined node sets the ID according to the information gained above and the relationship between Key and ID in the BalanceChord. After the setting, the registered service information will transfer from node with greatest load to new joined node. Generally about half of registered service information will move in order to keep the load balancing for all networks. The algorithm is shown in figure 3.

**Step 1.** New node applies to join in;
**Step 2.** Searching for the node with greatest load;
**Step 3.** New joined node sets its ID;
**Step 4.** Service information transfer to new joined node.

```java
addNode(List chord,int percent){
   for (int i=0;i<chord.size()*percent/100;i++) {
      CNode m_node=fineMaxLoadNode();
      int serviceKind=findDividedServiceNumber(m_node);
      CNode n_node=new CNode();
      n_node.setNodeID (serviceKind);
      BalanceNodeload(m_node,n_node,serviceKind);
   }
}
```

Fig. 3 New Node Join-in Algorithm

2.3.2. Self-balancing Algorithm

**Definition 3 (Overload Rate).** A node periodically compares its load with its neighbors. Usually the node can get the load information of its successor and predecessor. Suppose there is a node \( p_1 \), the successor of \( p_1 \) is \( p_2 \), and the predecessor of \( p_1 \) is \( p_0 \). The load of three nodes are \( L_{p_0} \), \( L_{p_1} \), \( L_{p_2} \). Overload rate \( L_{\Delta} \) can be defined as follows:
There is another parameter $\sigma$ which is the threshold of Overload rate. When the overload rate exceeds the threshold, the node is considered to be overloaded. The self-balancing algorithm will be invoked. In this algorithm, the node will transfer the registered service information to its two neighbors (successor and predecessor). The process will continue and affect the whole network. However this method may cause the concussion in network and the convergence speed is very slow. Therefore, we need a better method in order to transfer service information through one direction. In BalanceChord, the nodes are constructed to a circle in logic. The transfer of service information can be clockwise or anticlockwise. In this paper, we select two nodes $p_1$ and $p_2$, $(p_1 < p_2)$ and Overload rate can be redefined as follows:

$$L_\Delta = \frac{|L_{p_1} + L_{p_2} - L_{p_1}|}{3} \quad 0 \leq L_\Delta \leq 1$$

(2)

In this way when the overload rate exceeds the threshold, the node will transfer the registered service information to its successor. Similarly, the other nodes take the same work. In a limited time the load will balance in whole network. The algorithm is shown in figure 4.

**Step 1.** Comparing load between the node and its successor;
**Step 2.** If the node is overloaded, go to Step 3; else go to Step 1;
**Step 3.** Node sets its new ID;
**Step 4.** Service information transfer to its successor.

The transfer of service information between nodes must be controlled. If every node can invoke the self-balancing algorithm at any time, there will be a balancing storm in BalanceChord. In order to prevent the storm, another method is proposed in this paper. There is a unique identifier named HP (Hot Potato) in BalanceChord. If the node has the HP, it can invoke the self-balancing algorithm and transfer HP to the next node after self balancing. If the node isn’t overloaded, it will do nothing but transfer HP to the next as fast as it can (just like transfer a hot potato). The number of HP depends on the scale of BalanceChord. When the scale of network is large, there will be the efficiency problem using only one HP. We set several HPs, and every HP is transferred in corresponding part of ID space. This method improves the efficiency of self-balancing algorithm and will not cause the balancing storm.

```java
BalanceChord(List chord, int percent) {
    for (int i = 0; i < chord.size(); i++) {
        CNode cnode = chord.get(i);
        if (IsOverLoad(cnode)) {
            int serviceKind = getNewChordID(cnode);
            cnode.setChordID(serviceKind);
            BalanceLoad(m_cnode, serviceKind);
        }
    }
}
```

Fig. 4 Self-balancing Algorithm
3. Experiment and Result Analysis

In order to simulate and verify the WSDBC model proposed in this paper, we design the simulative system ChordSim. The ChordSim is composed by several modules: display module, simulative data creation module and routing module. Experimental project includes three parts: Part 1, firstly using random function we select some nodes and construct Chord network and BalanceChord network. Then we disperse several services in the nodes of two networks and test the load information. Part 2, we test the load information after nodes start self-balance algorithm when the number of nodes doesn’t change. Part 3, we test the load information when some nodes join in the networks. Experimental results are shown as figure 5.

Figure 5(a) shows the load information of Chord; Figure 5(b) shows the load information of BalanceChord. From the figures we can see that the load information of nodes in Chord is very unbalanced. The load information of some nodes exceeds 100 and some near to 0. Compared to Chord, our BalanceChord has better load balancing. Figure 5(c) shows the load information of BalanceChord after self balancing. Compared with Figure 5(b), the load of nodes is more balanced. The load of nodes trends to the average. In Figure 5(b), the load information of some nodes is greater than 35 or smaller than 10. But in Figure 5(c) the load information of all nodes is not greater than 30 or smaller than 10. It is the contribution of self balancing. Figure 5(d) shows the load information of BalanceChord after about 10% new nodes join in. The new joined nodes share part load of original nodes, so that the whole load of model is still balanced. Here we don’t give the figure about the load information of Chord after some nodes join in because after some nodes join in the load information of Chord doesn’t change greatly. The threshold of overload rate used in the experiment is 25%. Compared to Chord, our BalanceChord and load balancing strategy can work effectively.
4. Conclusions and Future Work

With the development of Web service, the number of available Web service is growing day by day. How to improve the efficiency of service discovery continues to be an important issue. In this paper, we propose the WSDBC model in order to extend the Web service discovery scope and improve discovering efficiency using P2P technology. To achieve load balancing between nodes in WSDBC model, new node join-in algorithm and self-balancing algorithm are proposed. The experimental results show this model and algorithms have a good performance.

Web service locating is the first step of Web service discovery. We also need some relevant service matching algorithms to provide accurate services to users. If there are several candidate services with same function, the quality of service may be considered as well. These are all included in our future work.

Reference