

GISP: Global Information Sharing Protocol

— a distributed index for peer-to-peer systems —

Daishi Kato

Computer Science Department, Stanford University
Visiting from NEC Corporation

Abstract

This paper proposes GISP (Global Information Sharing Protocol), which aims at a world-wide distributed index. A distributed index consists of a set of pair data (key, value) shared by many peers. Each peer is responsible for a part of the index based on a hash function. Every peer is basically flat and there is no single point of failure. A distributed index is an essential building block for peer-to-peer systems.

The design of GISP is simple, open, and easy to develop. GISP deals with peer heterogeneity and undesirable peers. Each peer promotes its strength so that stronger peers contribute more than weaker peers. Redundancy is important for defending against undesirable peers. Peers replicate pair data so that each pair data of the index is covered by several peers.

There is a project at jxta.org for developing GISP. JXTA is a set of protocols for a peer-to-peer platform and provides a Java reference implementation. At the project, GISP is implemented in the Java language on JXTA. By building GISP on top of JXTA, a peer could reach a peer behind a firewall and even a peer in a different network transport. Jnushare is another project at jxta.org, which is to provide an application of GISP. Using Jnushare, people would share information such as files, messages and bookmarks.

1. Introduction

Peer-to-peer systems have become popular with the growth of PCs and networks. Powerful PCs and broad-band networks made it possible to do calculations on the network edge. The most known peer-to-peer applications are file-sharing systems such as Napster, Gnutella and Freenet. Those systems are quite successful in the sense of promoting peer-to-peer file-sharing systems. However, these systems have a limitation of scalability.

Napster has a centralized index where the scalability can be limited by the machine power and the network bandwidth of the central point. There is a way to have multiple

index servers which are all identical, but it would only scale if the machine power were big enough and there were much less frequent insert messages than query messages.

Gnutella and Freenet are fully decentralized systems and have no single point of failure. However, their messaging mechanisms are based on application-level broadcasting, which is likely to limit the scalability. Why do the broadcast messages limit the scalability? As an example, suppose there are 10,000 peers (computers) connected somehow, and each peer send a 1K byte query message every minute. One peer would get 10M byte every minute, which would require 1.3Mbps connection speed. This connection speed is not impossible these days, but not all peers have that fast connection speed yet.

Recent Gnutella-like systems have a solution. In the previous example, you may notice that some of peers may have 1.3Mbps connection speed. That's correct. You don't have to broadcast messages to all peers but just some of them; this is called "super peer". By having "super peer"s and restricting broadcast message only among "super peer"s, the entire network would be more scalable. So far, this "super peer" idea seems to be working well.

The other approach, which this paper takes, is called a hash-based distributed index. In this approach, a peer would never send a broadcast message. Each peer decides which peer to send a message to in a common way. Usually, the common way which peer to select is based on a hash function. There has been some research proposed to build a hash-based distributed index. The basic idea is to share a set of pair data of (*key*, *value*) in many peers. The index is distributed typically based on the hash value of the *key* and the hash value of the peer ID. As an example, let's take the case putting the pair data of ("url:car", "http://car.car.car") into a distributed index. Suppose the hash value of "url:car" is "123", and there are two peers peer1 and peer2 whose hash values are "120" and "130", respectively. If the definition is to select the peer whose hash value is numerically the closest to the hash value of the *key*, peer1 will be selected as a target peer. If the definition is to select the peer whose hash value is numerically the next largest to the hash value

of the *key*, peer2 will be selected. The target peer is used both for inserting the pair data and for retrieving the pair data, while no other peers are involved.

There is an important notion about this idea. Since a hash function is used to build a distributed index, a peer can't guess what kind of data it has to keep. Therefore, all peers have to volunteer to provide PC power and network band-width, and to admit getting any kinds of data.

The idea of distributed indexes is adaptable to many peer-to-peer applications. File-sharing systems can use it for locating files by keywords. Instant messaging systems can specify the pair data of (*username*, *ipaddress*). The distributed index can also be used to find some specific web sites regarding to some topics. Most peer-to-peer applications get benefits from distributed indexes at some level.

This paper proposes Global Information Sharing Protocol (GISP) to provide a protocol for a world-wide distributed index. GISP shares some commonalities with other systems which provide distributed indexes. In GISP, as well as in the other systems, each peer is assigned a m -bit value, each key also has a m -bit value, and a peer responsible for a key is defined based on these values. While the other systems typically try to provide more efficient routing algorithms with small routing tables, GISP intends to leverage those algorithms and make a practical protocol. In order to make a protocol practical, some of the practical issues such as data replication, peer heterogeneity and undesirable peers are discussed. There is a reference implementation of GISP on a peer-to-peer platform, JXTA.

This paper consists as follows: Section 2 discusses some related work, the design of GISP is described in section 3, section 4 shows how to implement GISP, how GISP works on JXTA is discussed in section 5, and section 6 concludes this paper.

2. Related work

2.1. Chord

Chord [7] provides a `lookup(key)` algorithm which returns the IP address of the peer responsible for the key. A distributed index can be easily constructed by this algorithm. Each peer in Chord has m -bit ID and is placed in a circle by the numerical order of its ID. A key ID is produced by hashing the key, and a peer whose ID is equal to or follows the key ID is responsible for the key.

Chord guarantees that in N -peer network, each peer maintains information only about $O(\log N)$ other peers, and a lookup requires $O(\log^2 N)$ messages. Simulation and experimental results in the real network show the scalability of Chord.

2.2. CAN

CAN (Content-Addressable Network) [8] is also what is called a distributed index. The design of CAN is a virtual d -dimensional space. The entire space is partitioned among all peers in CAN, and each peer is responsible for its individual distinct zone within the virtual space. A key will be hashed into a zone and the peer in the zone is responsible for the key.

The interesting point of CAN is that each peer has only $2d$ neighbor peers, where d is independent from the network size. This is good when a new peer joins the network, because the number of peers which will be affected can be kept small even in a larger network. On the other hand, the average number of routing hops is $(d/4)(N^{1/d})$ in N -peer network.

2.3. Pastry

Pastry [9] is a self-organizing overlay network. Each peer in the network is assigned a 128-bit ID. A key ID is also 128-bit and the peer whose ID is numerically the closest to the key ID is responsible for the key. Pastry provides such an efficient routing table that the number of routing hops is less than $\lceil \log_{2^b} N \rceil$, where N is the number of peers in the network and b is a configuration parameter with typical value 4.

Experimental results are performed with up to 100,000 peers in a simulated network. The results show that even in the 100,000 peers network, the average number of hops is about 4 and the maximum number of hops is 5.

2.4. Tapestry

Tapestry [10] is based on Plaxton algorithm [11]. Each peer has a digit ID. The peer whose ID shares the longest suffix with a key ID is responsible for the key. For routing, a peer has to keep a neighbor map which contains $(b \log_b I)$ peers, where b is the base of ID and I is the ID of the peer. Based on the Plaxton algorithm, Tapestry adds some functions such as fault handling and dynamic peers.

Using the Plaxton algorithm is remarkable in the following sense. Since IP addresses of peers in the same LAN share some length of the same prefix, using IP addresses as peer IDs make the overlay network topology to be somewhat similar to the real network topology. Therefore, the latency of one message hop tends to be small.

3. GISP design

GISP, Global Information Sharing Protocol, was proposed at first from the community that is interested in peer-to-peer technology and its software. Since the first idea is to

overcome the limitation of the original Gnutella, GISP aims not at proving a theory but at providing a practical protocol. Here are GISP's design policies:

- To provide more effective information sharing mechanism than application-level broadcasting in a peer-to-peer network.
- To work with a fully decentralized peer-to-peer network, but considering heterogeneity of peer capabilities.
- To consider more simplicity in the protocol and less completeness in getting information.
- To deploy easily on a peer-to-peer platform, such as JXTA.
- To work functionally in a real network, assuming some attacks.

Some details are described in the following sections.

3.1. Non-broadcasting messages

In order to avoid broadcast messages, it is a good idea to determine one target peer for each insert/query. Let's take an example. Suppose there are three peers (peer1, peer2 and peer3). When peer1 wants to query for "car", what would peer1 do? One thing it can do is to always ask only peer2. What if peer3 wants to publish some information about "car"? If peer3 knew that peer1 always ask peer2 about "car", peer3 would notify peer2 that it had some information about "car".

By doing it this way, there would be no broadcast messages, but just peer-to-peer messages. The point is that all peers have to have common knowledge of which peer to select for each specific keyword.

3.2. Hash-based indexing

The common knowledge for all peers can be based on a hash function. A hash function is useful because it converts any binary data including a keyword into a number such as a 128-bit integer. Both a keyword and a peer ID can be converted into numbers (hash values) by a hash function. A peer ID could be any string data, which typically is an address of a peer.

Once you get the two hash values, all you have to do is to define the way of determining the target peer corresponding to the keyword. GISP defines to select the peer whose hash value is numerically the closest to the hash value of the keyword. If a hash value is a 128-bit integer, this calculation is done modulo 2^{128} . The range of hash values is usually very large, for example between 0 and $(2^{128} - 1)$, and it is unlikely that any two hash values of peers are exactly the same. Which hash function to use is up to developers, but it has to be common in all peers.

There is an important point to use hash-based indexing. Each piece of data has to have a keyword so that a peer can determine a target peer based on the keyword. However, the keyword does not have to reflect the meaning of the data. How to put a keyword is totally up to application developers.

3.3. Data inserting, querying and deleting

Using a hash function, data inserting and querying are performed as follows: When a peer (peer1) wants to insert a pair data ("foo", "http://foo/"), it takes the hash values of peers that it knows of, and inserts the pair data into the peer whose hash value is numerically closest to the hash values of "foo." When another peer (peer2) wants to query about "foo," it takes the hash values of peers that it knows of, and queries about "foo" to the peer whose hash value is numerically closest to the hash value of "foo". At this point, peer1 and peer2 have to know of the same set of peers in order to search the information correctly. The case that peers know of different sets of peers is discussed later.

Data deleting is somewhat different from inserting and querying. Once you publish a pair data, you lose the control of the data and it is very hard to delete the data completely. It is a good idea to have an expiration date for each data. A peer should check its local data periodically and delete expired data.

3.4. Data Replication

In the real peer-to-peer network, peers join and leave the network continuously. Furthermore, it is more likely that some peers get off the network without any notification, or just become unreachable because of a network problem. In order to deal with this, replication of data is necessary for redundancy.

A peer should insert/query a pair data not only to one target peer but to a set of peers whose hash values are numerically closer to the keyword hash value of the pair data than the hash values of other peers. The number of replicas that one peer should make can be either statically or dynamically defined in a common way.

Let's take an example where the number is statically defined as 4. Suppose there are six peers (peer1, peer2, ..., peer6) in the network, knowing each other and hash values 122 (peer1), 125 (peer2), 127 (peer3), 130 (peer4), 133 (peer5) and 135 (peer6).

Let's consider the case peer1 wants to insert a pair data of ("bar", "http://bar/") and the hash value of "bar" is 124. Peer1 inserts the pair data to the top 4 peers whose hash values are numerically closest to 124: peer2, peer1, peer3 and peer4 in this case. Because peer1 itself is in this list, peer1 is responsible for the pair data and does the following:

- If peer3 goes down and peer1 detects the failure of peer3, peer1 inserts the pair data into the next closest peer, peer6.
- If peer1 detects a new peer (peer7 with hash value 126), peer1 inserts the pair data into peer7, because peer7 is now the third closest to “bar.”

This also happens to peer2, peer3 and peer4. This continues until a peer finds 4 peers closer than it and loses the responsibility. In another case, peer1 wants to query about “abc”, whose hash value is 132. Peer1 queries to the top 4 peers: peer5, peer4, peer6 and peer3.

In this example, where the number of replica is set to 4, unless 4 peers go down at the same time, it is expected that data will be kept in the network.

3.5. Message routing

If the world of peer-to-peer networks were small, the mechanism described above would work fine and there would be no problem. However, in a real peer-to-peer network, there are so many peers that one peer cannot hold the information of all the peers. Furthermore, a new peer will not be known to other peers until a period of time passes.

To deal with such situations, message routing is required. If a peer receives a message and the peer knows more appropriate peers, which means there are peers whose hash values are numerically closer to the hash value of the message keyword, the peer routes the message to the more appropriate peers.

For example, peer1 with hash value 125 inserts a pair data with hash value 130 to peer2 (140), peer3 (150) and peer4 (160). Peer2, who knows peer5 (135), peer6 (145), peer3 and peer4, routes the insert message to peer5, peer6 and peer3. This routing continues until no peer receives messages.

In order to avoid message looping, a peer should include the list of peers that it has already sent. For example, peer7 sends a message to peer8 and peer9. When peer7 sends it to peer9, it also tells to peer9 that it has already sent it to peer8, so that peer9 does not have to route it to peer8.

3.6. Local peer information

To reduce the traffic in the entire network, it is desirable to route messages as rarely as possible. It is also important that all peers in the network should connect somehow and never split in groups. There needs to be a common method in all peers how to collect local peer information. GISP requires the following rules.

- A peer should collect as much peer information as possible to reduce message routing.
- A peer can discard information of unreachable peers.

- If a peer cannot keep all peer information, it can discard some of it.
- A peer should keep more peer information whose hash value is numerically close and less peer information whose hash value is numerically far.
- A peer should keep peer information so that any peers in the network can reach any other peer in a reasonable hop count.

As is shown in section 2, there is some work which discuss how to provide an efficient routing table. GISP could leverage this work to have a better routing mechanism. GISP extends the Chord routing table to have more peer information as a cache. Based on the Chord routing table, the routing requires at most $O(\log^2 N)$ messages, and in some cases, it requires much fewer messages because of the cache information.

3.7. Peer joining and leaving

When a new peer wants to join the network, it has to know at least one existing peer in the network. Once the new peer knows the existing peer, it can ask the existing peer to learn about some more peers. How to know the first existing peer is an implementation issue.

When an existing peer gets off the network, it would be a good idea that the peer notifies other peers that it is going down. However, some of peers could be down just because of a network problem. In the GISP design, the leaving notification is not mandatory. Unless many peers go down at the same time, no data will be lost because of data replication.

3.8. Peer heterogeneity

Since not all peers have the same capabilities, some peers can keep more data than others. An integer value “peer strength” makes it possible to have peer heterogeneity. Each peer includes the “peer strength” value in its information. All peers should reflect “peer strength” into calculating the distance of peers in a common way.

In the GISP design, the distance between two peers is calculated as $distance(X,Y)/(2^{L-1}2^{M-1})$ where X,Y are hash values of the two peers and L,M are “peer strength” values of them. The knowledge of the way to change the distance based on “peer strength” has to be shared by all peers.

3.9. Overlay network topology

GISP provides an overlay network. The topology of this overlay network is basically independent from the topology of the real network. Even if it is one hop in the GISP network, it could be the other side of the earth. The worse case

is if there are two or more hops. It could travel all over the world to reach a peer at the next table.

To reduce the network cost of message routing, the overlay network topology should be reflected by the real network topology. Since GISP requires a peer to know about more peers whose hash value is numerically close to its hash value, two peers whose hash values are numerically close send more messages each other than two peers whose hash values are numerically far. Happily, in the GISP design, a peer ID is independent from anything such as an network address. It can be decided by its own way so that near peers in the real network have numerically close hash values.

One simple way to accomplish this is to introduce a latency-based join mechanism. When a new peer is joining the network, it first looks for a set of peer information out in the network. This information includes peer ID and latency value. The new peer could determine its peer ID based on these latency values. It is expected that peers that are near in the real network have numerically close hash values.

3.10. Undesirable peers

GISP should be open so that every developer could have his own implementation. However, this also means it is possible to make undesirable peers or even attacking peers. Some example undesirable peers would be:

- a peer which receives messages but never sends messages,
- a peer which sends incorrect messages,
- a peer which sends too many messages, and so on.

Each peer should defend against these undesirable peers. It should deal with unexpected messages and discard them properly. If a peer detects a flooding message, it discards the information of the source peer. Data replication, as is described above, is necessary to defend against attacking peers, because a peer could evaluate the worth of a message based on messages from other peers.

3.11. Fuzzy search

If you want to search information about “car”, you might also want some about “Car” and “CAR”. GISP is based on a hash function. By the hash function, the hash value of “car” is usually far different from that of “Car”. This means the target peer for “car” and the target peer for “Car” could be different. If a peer inserts the information about “car” and other peer tries to search for “Car”, it is more likely to fail.

Since GISP cannot handle this problem because it doesn't know “car” and “Car” are same meanings, application developers should carefully handle this. For example, one might want to convert all keywords into lower case. If there is a phrase such as “white car”, one could insert two pair data whose keys would be “white” and “car”.

4. GISP implementation

This section describes how to implement GISP based on the design. It should be independent from the programming language and the network transport; however, the following sections assume the Java language and the XML messaging transport for convenience. GISP does not assume a reliable network such as TCP, instead, it should work on a best-effort network such as UDP. It is required to have a message failure detection mechanism.

4.1. Hash function

It is not important which hash function to use, as long as it hashes string data into a number of some bits consistently and evenly. MD5 and SHA-1 are such hash functions. One other concern is which encoding to use for string data before hashing it. All peers have to use a common encoding such as UTF-8.

4.2. Local data

Each peer contains a set of local data. Here is a example of the local data. A peer could have other local data for any purpose.

- `peertable`
This keeps peer information. If a peer detects peer failure, it removes its information from its `peertable`.
- `datalist`
This keeps (*key, value*) pair data. Each pair data has an expiration date. If pair data is expired, a peer removes it from `datalist`.
- `replytable`
This keeps IDs of messages which are originally sent. If a peer gets a reply message back, it removes the ID. If no reply message is received and a certain time is passed, it will timeout and the ID will be removed. In either case, the peer may perform some actions.

4.3. Message format

A message format is the most important to develop a protocol. Here an example message is shown and each element is described in detail.

```
<message>
<msgid>00031</msgid>
<source>
  <address>192.168.0.1</address>
</source>
<reply><msgid>00123</msgid></reply>
<peerinfo>
  <peerid>c8z5e1k2v0i3</peerid>
```

```

<address>192.168.0.2</address>
<strength>3</strength>
<expire>Tue, 4 Jun 2002
 14:53:32 -0000</expire>
<sent><address>...</address></sent>
</peerinfo>
<peersearch>
  <limit>100</limit>
  <peerid>c7pla8m5ml</peerid>
</peersearch>
<insert>
  <keyword>url:car</keyword>
  <strdata>http://car.car/</strdata>
  <expire>Tue, 4 Jun 2002
    15:40:14 -0000</expire>
  <sent><address>...</address></sent>
</insert>
<query>
  <keyword>url:dog</keyword>
  <!-- <substr>...</substr> -->
</query>
<result>
  <keyword>url:cat</keyword>
  <strdata>http://cat.cat/</strdata>
  <expire>Tue, 4 Jun 2002
    15:40:14 -0000</expire>
</result>
<requery><peerinfo>...</peerinfo></requery>
</message>

```

4.3.1. Message element. A message element is the top-level element of messages which go back and forth between peers. It has several sub-elements, some of which are described here and the rest of which are described in later sections.

In a message element, there can be zero or one `msgid` elements, zero or one `source` elements and zero or one `reply` elements. A `msgid` element keeps the ID of the message, which is represented by any string data. A `source` element has one `address` element which indicates the source peer address of the message. A `reply` element has one `msgid` element which keeps the ID of the received message. A message with `reply` element is also referred as a reply message.

When a peer gets a message with `msgid` and `source` elements, it should send a reply message which is any message with a `reply` element having the `msgid` of the received message in it. When a peer gets a message with `reply` elements, it should clear the corresponding items in `replytable`.

4.3.2. Peerinfo element. There can be zero or more `peerinfo` elements in a message element. Each `peerinfo` element includes exactly one `peerid` element, one `address` element, one `strength` element, one `expire` element and zero or more `sent` elements. A `peerid` element has any

string data identifying the peer. An `address` element has the peer address in the network, such as IP address. A `strength` element has an integer value between 1 and 10 indicating “peer strength.” An `expire` element has a string representation of date, for example in RFC-822 date format, which specifies when this `peerinfo` will expire. A `sent` element has one `address` element which indicates the address that this `peerinfo` is already sent to.

When a peer receives a message with `peerinfo` elements, it should add all `peerinfo` into `peertable` except for its own `peerinfo`. For each piece of `peertable`, if there are more appropriate peers, the peer should route a `peerinfo` message to the appropriate peers.

4.3.3. Peersearch element. There can be zero or one `peersearch` elements in a message element. A `peersearch` element can have zero or one `limit` elements and zero or one `peerid` elements.

When a peer receives a message with a `peersearch` element, it should send back all `peerinfo` in its `peertable` and its own `peerinfo` to the specified address in a `source` element. If a `limit` element is specified, the peer should limit the number of returning `peerinfo` up to the specified number. If a `peerid` element is specified, the peer should return `peerinfo` whose hash values are numerically closer to the hash value of the specified peer ID.

4.3.4. Insert element. There can be zero or more `insert` elements in a message element. Each `insert` element includes exactly one `keyword` element, one `strdata` element, one `expire` element and zero or more `sent` elements. A `keyword` element and a `strdata` element make up pair data of (*key*, *value*). An `expire` element, which is described earlier, specifies when this pair data will expire. A `sent` element has one `address` element which indicates the address that this `insert` is already sent to.

When a peer receives a message with `insert` elements, it should add all data of the elements into `datalist`. For each piece of data, if there are more appropriate peers, the peer should route a `insert` message to the more appropriate peers.

4.3.5. Query element. There can be zero or one `query` elements in a message element. A `query` element must have exactly one `keyword` element and it may have one `substr` element.

When a peer receives a message with a `query` element, it should search `datalist` for the pair data whose `keyword` is equal to the string in the `keyword` element, and return the results in `result` elements to the address indicated in the `source` element. If the `substr` element is specified, the peer searches for the pair data whose *value* has the substring of the element.

If there are more appropriate peers for the keyword, It should also return the peers with `requery` elements.

The message with `query` element always has to contain a `msgid` element and a `source` element. The result message should have `reply` element in order to relate the results with the query,

4.3.6. Result element. There can be zero or more `result` elements in a message element. A `result` element can contain the same elements as in an `insert` element except for `sent` elements. In order to relate the results with the query, the message always has to contain a `reply` element.

When a peer receives a message with `result` elements, it will keep them in a queue or call a registered listener.

4.3.7. Requery element. There can be zero or more `requery` elements in a message element. A `requery` element can have one `peerinfo` element which contains more appropriate peers for the query. The message with `requery` elements should contain a `reply` element so that a peer can recognize the previous query that this `requery` is for.

When a peer receives a message with `requery` elements, it can send the query message to some of the peers specified in `requery` elements.

4.4. Maintaining local data

A peer has to maintain local data to keep it updated. Since each pair data of (*key*, *value*) has an expiration date, the peer should delete the data if it is expired from `datalist`. As well as the expiration date, peer information is also deleted from `peertable` if the error count of the peer exceeds a limit.

As described before, the peer has to look up in `replytable` to find any message timed out. If a timed out message is found, the error count of the target peer will be increased.

4.5. Bootstrapping

Bootstrapping is what a new peer does when it goes up. When a new peer wants to join GISP network, it has to know one or more seed peers in GISP network. Usually it should have more than one of them for redundancy, because some of the peers could be already off or undesirable.

In order to get seed peers, there could be several ways:

- *cache file.* This contains a lot of `peerinfo` some of which is expected to be up in the network. This file comes with the package. A peer would try to connect to each of these `peerinfo` until it finds an available peer.
- *outside method.* A peer gets some information somehow from outside of this protocol. A peer owner can

get one via e-mail. There could be a central server to notify existing `peerinfo`.

After finding seed peers, the new peer tries to connect to some of the peers. If it connects at least one peer, it can send a `peersearch` message to the target peer. If it gets some results, it can even re-send a `peersearch` message to some of the peers in the results.

When the new peer is ready to publish its own `peerinfo`, it sends a `peerinfo` message to basically all known peers.

4.6. API

GISP tries to provide as simple an API as possible. All application developers have to do is to send `insert` messages, to send `query` messages and to register listeners for query results. Here is a part of Java interface:

```
void insert(String data,
            String keyword, Date expire);
void queryByKeyword(String keyword);
void queryBySubstr(String keyword,
                  String substr);
void addResultListener(ResultListener l);
```

5. GISP on JXTA

GISP started as a project at jxta.org in August 2001. This section describes JXTA briefly, and explains some points on how to deploy GISP on JXTA. It also introduces the Jnushare project which tries to build an information sharing application using GISP.

5.1. Project JXTA

Project JXTA (or simply JXTA) is an open source community for peer-to-peer systems sponsored by Sun Microsystems, Inc. JXTA provides a peer-to-peer platform which includes the JXTA protocol specification and a Java reference implementation. Developers can build their own services and applications on the JXTA platform.

One of the remarkable things about JXTA is that it gives you an overlay network. This network could be over any existing network transport, and it even traverses firewalls and NATs. This is very good for peer-to-peer application developers who don't want care about underlying network topology. More information is available at Project JXTA web site [1].

5.2. Deploying GISP on JXTA

Since JXTA provides a set of APIs to build a peer-to-peer application, it is somewhat easy to deploy GISP on JXTA. Here are some points on how to do it.

5.2.1. Message transport. JXTA provides its own message transport, called the Endpoint Service. To send and receive messages via the Endpoint Service, you need to use an Endpoint Message structure. The Endpoint Message is a set of pair data of (*name*, *value*) where *name* is any string data and *value* is any binary data, and does not just allow hierarchical XML document. It is necessary to encode a GISP message into an Endpoint Message.

The address of a peer for the Endpoint Service is basically the JXTA-PeerID which is assigned when a peer is first created. The Endpoint Service gets an address (JXTA-PeerID) and a message (Endpoint Message), finds a route to the target peer using Endpoint Routing Protocol (ERP), and sends the message.

5.2.2. Seed peers. As is discussed, there needs to be some way to find seed peers. JXTA has Peer Resolver Protocol (PRP) which supports generic queries within the JXTA network. GISP peers can use PRP to find seed peers. Once the peer can connect to an available peer, it can send a peer-search message to get more peerinfo.

5.3. Jnushare project

There is another project at jxta.org named Jnushare. The intention of Jnushare project is to provide an information sharing application. It utilizes the JXTA CMS (Content Management System) for uploading and downloading content, and utilizes GISP for their indexes. Jnushare uses GISP to keep pair data of (*name*, (*address*, *contentid*)) so that it can search any content by name.

The Jnushare project as well as the GISP project is an on-going project and members at the projects are still trying to improve the protocol and the software.

All the source code described above are available at projects' web sites [2, 3]. Those who are interested in details could take a look at the source code.

6. Conclusion

This paper proposed GISP to provide a world-wide distributed index. The design of GISP is simple, open and easy to develop. The design showed the requirement of a distributed index in the real network, such as replication, routing, heterogeneity and defending against attacks.

The description of the GISP implementation showed mainly about the message format, because it is the most important part of the protocol. The message format is so flexible that there can be many kinds of elements in one message. The most essential behavior of the protocol is also described.

Future work should follow by simulating the protocol and improving the protocol especially for the message routing.

Reducing message hops and detecting message loops are the most important.

JXTA is introduced as a platform of GISP. JXTA provides a set of protocols for a peer-to-peer platform and a Java reference implementation. GISP is developed on JXTA and it utilizes the JXTA features, such as the Endpoint Protocol and the Resolver Protocol. Jnushare is an application of GISP. It lets people share information such as files, messages and bookmarks.

The GISP project and the Jnushare project at jxta.org are open-source projects, and everyone can participate in the projects to work on them.

Acknowledgements

The author thank Prof. Michael Genesereth, Dr. Charles Petrie and Michael Kassoff for helpful advice.

References

- [1] Project JXTA. <http://www.jxta.org/>
- [2] GISP Project at jxta.org. <http://gisp.jxta.org/>
- [3] Jnushare Project at jxta.org. <http://jnushare.jxta.org/>
- [4] Napster. <http://www.napster.com/>
- [5] Gnutella. <http://www.gnutelliums.com/>
- [6] Freenet. <http://freenetproject.org/>
- [7] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek and H. Balakrishnan. *Chord: A scalable peer-to-peer lookup service for internet applications*. In Proc. ACM SIGCOMM (San Diego, California, Aug. 2001).
- [8] S. Ratnasamy, P. Francis, M. Handley and R. Karp. *A scalable content-addressable network*. In Proc. ACM SIGCOMM (San Diego, California, Aug. 2001).
- [9] A. Rowstron and P. Druschel. *Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems*. In Proc. IFIP/ACM International Conference on Distributed Systems Platforms (Middleware), (Heidelberg, Germany, Nov. 2001).
- [10] B. Y. Zhao, J. Kubiatowicz and D. Joseph. *Tapestry: An infrastructure for fault-tolerant wide-area location and routing*. UCB Tech. Report UCB/CSD-01-1141 (University of California Berkeley, Apr. 2001).
- [11] C. G. Plaxton, R. Rajaraman and A. W. Richa. *Assessing nearby copies of replicated objects in a distributed environment*. In Proc. ACM SPAA (Newport, Rhode Island, Jun. 1997).