A Framework for Constructing Peer-to-Peer Overlay Networks in Java

Rui Shen[1], Ji Wang[1], Shengdong Zhang[1, 2], Siqi Shen[1], Pei Fan[1]
[1] National Laboratory for Parallel and Distributed Processing, Changsha, 410073, China
[2] School of Computing, University of Leeds, Leeds, LS2 9JT, UK
rui.shen@gmail.com, jiwang@ios.ac.cn, scsszh@leeds.ac.uk

ABSTRACT
Peer-to-peer emerges as a better way for building applications on the Internet that require high scalability and availability. Peer-to-peer systems are usually organized into structured overlay networks, which provide key-based routing capabilities to eliminate flooding in unstructured ones. Many overlay network protocols have been proposed to organize peers into various topologies with emphasis on different networking properties. However, applications are often stuck to a specific peer-to-peer overlay network implementation, because different overlay implementations usually provide very different interfaces and messaging mechanisms. In this paper, we present a framework for constructing peer-to-peer overlay networks in Java. First, networking is abstracted by interfaces that use URIs to uniformly address peers on different underlying or overlay networks. Then, asynchronous and synchronous messaging support is built upon these interfaces. Finally, overlay networking interfaces are sketched to handle specific issues in overlay networks. We have constructed several overlay networks in this framework, and built peer-to-peer applications which are independent of overlay implementations.

Categories and Subject Descriptors
D.1.5 [**Programming Techniques**]: Object-Oriented Programming. D.2.3 [**Software Engineering**]: Coding Tools and Techniques – *object-oriented programming*.

General Terms
Languages, Design.

Keywords
Peer-to-peer, Overlay network, Messaging.

1. INTRODUCTION
Peer-to-peer networks are distributed computer architectures that use diverse connectivity between participants for the sharing of resources (content, storage, CPU cycles and etc.) on the Internet, rather than requiring the support of centralized servers [1]. They emerge as a better way for building applications on the Internet that require high scalability and availability. On the one hand, peers act as both clients and servers of resources by coordinating with each other. As new peers arrive and demand on the system increases, the total capacity of the system also increases. On the other hand, the distributed nature of peers also increases availability by replicating resources over multiple peers. Peer-to-peer architectures are widely used for Internet applications such as file sharing, video streaming, distributed computing, distributed collaboration, military botnet and etc.

Peer-to-peer systems are usually structured into overlay networks by organizing peers into certain topologies, and employ globally consistent protocols that can efficiently route a key to the corresponding peers [12]. This key-based routing capability can be used to eliminate flooding in unstructured peer-to-peer systems. However, to ensure the consistency of the routing process, the topology must be properly maintained when nodes arrive or depart, and application data also need to be replicated or migrated properly when the topology changes. Many overlay protocols have been proposed to organize peers into various topologies with emphasis on different networking properties, e.g. routing path length (FISSIONE [6]), churn overhead (Bamboo [9]), proximity metric (Pastry [11]) and etc.

There are many implementations of peer-to-peer overlay networks, e.g. Chord [12], Pastry [11], OpenDHT [10] and etc. However, applications are often stuck to a specific overlay network implementation, because different overlay implementations usually provide very different interfaces and messaging mechanisms. JXTA is a set of open protocols that enable any connected device on the network to communicate and collaborate in a peer-to-peer manner [4], but it does not specify the constructing of overlay protocols. OverSim is a C++ overlay network simulation framework to evaluate new protocols [2]. Although it provides a key-based routing interface, applications need to know the details of the overlay protocol to explicitly handle data replication and migration.

In this paper, we present a framework for constructing peer-to-peer overlay networks in Java. First, networking is abstracted by interfaces that use URIs to uniformly address peers on different underlying or overlay networks. Then, asynchronous and synchronous messaging (remote method call) support is built upon these interfaces. Finally, overlay networking interfaces are sketched to handle specific issues in overlay networks, e.g. key-based routing, application data replication and migration. We have constructed several peer-to-peer overlay networks (Chord [12], FISSIONE [6] and etc.) using this framework, and built

PPPJ'09, August 27-28, 2009, Calgary, Alberta, Canada.

peer-to-peer applications (e.g. a distributed hash table application, and a content-based publish/subscribe service) which are independent of overlay implementations.

2. PEER-TO-PEER OVERLAY NETWORK

Peer-to-peer overlay networks permit routing messages to destinations not specified by the underlying network address, e.g. IP address. Instead, the routing destination is determined by a *key*, which can be any object that an application might use, e.g. integers, strings and etc. Overlay networks employ globally consistent protocols to efficiently route messages based on their keys. To achieve this key-based routing capability, an overlay network is usually constructed by specifying: 1) a topology to organize peers, and its corresponding identifier (ID) space to address peers and map all possible keys into; 2) distribution of the keys to peers and the corresponding routing algorithms; 3) maintenance of the topology as well as related application data, when nodes arrive or depart.

2.1 Topology and ID Space

Peers in an overlay network are organized into a certain topology, such as ring, mesh, torus and etc. The topology should be dynamic to accommodate changes and self-organize when new nodes arrive or existing ones depart. The topology is usually associated with an ID space to identify its nodes, i.e. peers and all possible keys.

For example, Chord uses a ring topology [12], and its ID space is the integers from 0 to 2^n-1, and the successor of 2^n-1 warps back to 0. Essentially, each peer has a pointer to its nearest successor peer to keep the ring structure. FISSIONE uses Kautz graph topology [6], and its ID space is Kautz strings with base 2 and length *k*. Each peer has pointers to its in-neighbors and out-neighbors to keep the Kautz graph structure.

The mapping of peers and keys into the ID space is usually based on some hash function (e.g. SHA-1, MD5 and etc.) to uniformly distribute them in the ID space, thus these peer-to-peer overlay networks are also called distributed hash tables (DHTs). The ID space should be large enough to eliminate the collisions when hashing a large number of keys. For example, *n* is usually set to 160 for the Chord ring, and *k* is usually set to 160 for the FISSIONE Kautz strings.

2.2 Key Distribution and Routing Algorithm

Since the ID space is much larger than the number of peers, the whole ID space is partitioned by the peers. Thus, keys are distributed to their corresponding peers, i.e. a routing process to a key will always reach its corresponding peers. Note that a key may be distributed to more than one peer in some protocol to tolerant failures such as involuntary node departures.

For example, Chord distributes a key to the peer whose ID is the nearest successors of the key's ID among all the other peers [12]. FISSIONE distributes a key to the peer whose ID is the exact prefix of the key's ID [6].

Then, the overlay network needs a set of routing algorithms to ensure the consistency of the key-based routing, i.e. the routing of a key issued from any peer should always reach the corresponding peers of the key. Routing algorithms are distributed algorithms that elaborately construct a routing path on the topology by the collaboration of multiple peers along the path, using their pointers to other peers as routing tables.

2.3 Topology and Data Maintenance

During the execution of an overlay network, new nodes may arrive and existing ones may depart. In such occasions, the topology needs to be properly maintained to ensure the consistency of the key-based routing by adjusting routing tables of related peers.

For example, in Chord [12], when a new peer arrives, it must acquire a pointer to its nearest successor peer, and the original peer pointing to the successor peer should update its pointer to the new peer instead. When a peer departs, the peer pointing to the departing peer should update its pointers to the next successor peer.

Moreover, this adjustment also affects key distributions. For instance, when a new node arrives, it should take over its corresponding keys from their former corresponding peers; when an existing node departs, it should delegate its corresponding keys to their new corresponding peers. Thus application data associated with these keys should also be replicated or migrated to the new corresponding peers, otherwise they may never be found by the key-based routing process again.

3. RELATED WORK

Many peer-to-peer overlay networks have been designed and implemented. They usually provide very different interfaces and messaging mechanisms, especially for application data replication and migration.

The official Chord implementation [12] provides interfaces such as **`find_successor()`** to finding the successor of a key, since a key is distributed to its nearest successor peer. It also supports asynchronous remote procedure call (RPC) using callback functions. The maintenance of application data must be explicitly handled, which requires profound understanding of the protocol.

The Pastry implementation [11] provides interfaces to send data encapsulated in a **`Message`** object. Various remote operations are achieved using their corresponding subclasses of the **`Message`** class. Although applications can use the **`NodeSetListener`** interface to handle topology changes, data replication and migration still need certain understanding of the protocol.

OpenDHT [10] provides interfaces with **`put()`** and **`get()`** methods similar to hash tables, together with asynchronous RPC using callback functions. It can replicate/migrate data stored in the DHT. However, other applications using more complex data structures such as content-based publish/subscribe service are not well supported.

All the above projects implement a single overlay protocol, and they are all based on TCP transport. However, there are frameworks that can support constructing overlay networks based on various underlying transport protocols. JXTA is a set of open protocols that enable any connected device on the network to communicate and collaborate in a peer-to-peer manner [4], but it does not specify the constructing of overlay protocols. OverSim is a C++ overlay network simulation framework to evaluate new protocols [2]. Although it provides a key-based routing interface, applications need to know the details of the overlay protocol to explicitly handle data replication and migration.

Peer-to-peer applications need the ability to run on different underlying network protocols, as well as different overlay networks, so that they can choose the ones with desired networking properties. Thus a framework is needed to provide a set of interfaces for tasks such as mapping arbitrary objects into the ID space, messaging on the underlying networks for distributed algorithms, and application data replication and migration.

4. FRAMEWORK

In this section, we illustrate our framework to facilitate the constructing of peer-to-peer overlay networks in Java[1]. First, URI is used to uniformly address peers and resources on various underlying networks and overlay networks, as shown in Figure 1.

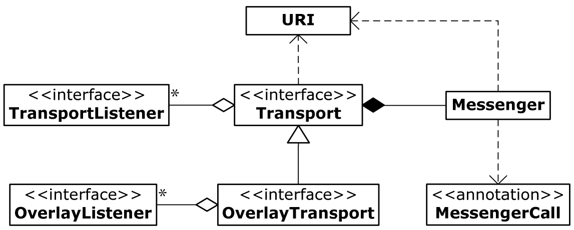

Figure 1. Overview of the framework.

A transport layer abstracts end-to-end communications in a network by **`Transport`** and **`TransportListener`** interfaces, which can be used to send and receive data in the network.

Then, a messaging module associates a **`Messenger`** with the transport layer to provide both asynchronous and synchronous calls to remote methods annotated by **`MessengerCall`**.

Finally, an overlay transport layer abstracts overlay networks by **`OverlayTransport`** and **`OverlayListener`** interfaces, which support mapping object into ID space, key-based routing, application data replication and migration.

4.1 Uniform Resource Identifier

Uniform Resource Identifier (URI) allows different types of resource identifiers to be used in the same context, even when the mechanisms used to access those resources may differ [3]. Since different network protocols use different addressing schemes, URI is appropriate to uniformly represent various addresses. It also allows introducing addresses for new network protocols without interfering with the way that existing addresses are used.

URI enables uniform addressing of peers and resources via a separately defined extensible set of naming schemes [3]. How identifications in a scheme are accomplished, assigned, or enabled is delegated to the corresponding network protocol. A URI is an identifier consisting of a sequence of characters in the following form:

```
SCHEME://HOST:PORT/PATH#FRAGMENT
```

In the framework, both underlying network addresses and overlay network addresses are represented using URIs. For example, a typical TCP address located at port 2000 of node 10.0.0.9 is represented by the following URI

```
tcp://10.0.0.9:2000
```

For an in memory network protocol that simulates networking in a single JVM using a hash map, a peer identified by string “node1” is represented as

```
map://node1
```

For an overlay network, the scheme is the protocol name, and the host part contains the ID of a resource. The following are example URIs for Chord and FISSIONE, respectively.

```
chord://393847626363648292384736363525282
fission://1201021021012012120212010101021
```

Moreover, remote objects used by the messaging module are also addressed using URIs, by assigning a path in the URI to each object that will receive remote calls, such as

```
tcp://10.0.0.9:2000/test
chord://393847626363648292384736363525282/dht
```

4.2 Transport Layer

The transport layer abstracts end-to-end communications in a network by **`Transport`** and **`TransportListener`** interfaces, which can be used to send and receive data in the network. Figure 2 shows the class diagram of the transport layer.

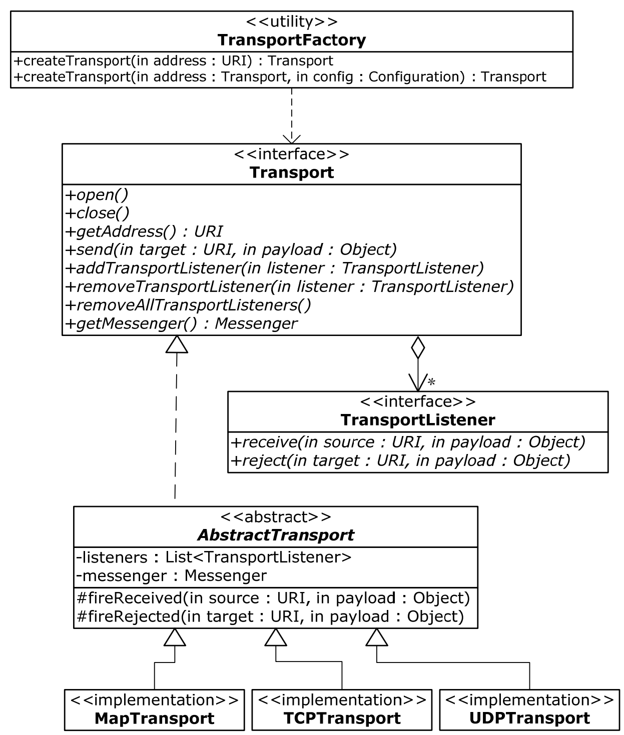

Figure 2. Transport layer class diagram.

The **`Transport`** interface defines common operations for various network protocols: open or close the network, query the address of a peer, send data to a target address, manage callback listeners and etc.

[1] This framework requires Java 1.5 or higher, since it use features such as the **`java.util.concurrent.Future`** interface, generic types and annotations.

The **TransportFactory** utility class is used to create concrete transport instances specified by the given URI. For example, in the following code snippet, **TCPTransport** and **MapTransport** are created by binding to specified addresses.

```
Transport transport;
transport = TransportFactory.createTransport(
      URI.create("tcp://10.0.0.9:2000"));

transport = TransportFactory.createTransport(
      URI.create("map://node1"));
```

Then, the **open()** method must be called to allocate JVM resources (e.g. buffers and sockets), while the **close()** method will release the allocated resources. The **getAddress()** method returns the address of the instance in the network.

The **send()** method are used to asynchronously send data to target addresses. The data can be any object that is serializable[2]. For example, the following code snippet sends “Hello” to a node in a TCP network.

```
transport.send(
        URI.create("tcp://10.0.0.9:2000"),"Hello");
```

To receive data, implementations of the **TransportListener** interface should be registered to the **Transport** instance. The following code snippet registers a **TransportListener** implementation (as an anonymous class).

```
transport.addTransportListener(
  new TransportListener() {
    public void receive(URI source, Object payload) {
      // receive a object from the source node
    }

    public void reject(URI target, Object payload) {
      // a send to the target node is failed
    }
  }
);
```

When data is received, all the registered listeners will be notified by invoking their **receive()**method. Moreover, since data sending is asynchronous, when a sending failed, all the registered listeners will also be notified by invoke their **reject()** method, so they can resend the data if necessary.

To facilitate the implementation of the **Transport** interface, the framework provides an **AbstractTransport** class that implements some common tasks, such as listener management and callback notification. And several concrete implementations are provided for commonly used networking protocols.

The **TCPTransport** class uses Java NIO to implement asynchronous communication for the TCP protocol. The **UDPTransport** class is a simple implementation for the UDP protocol.

The **MapTransport** is an in memory network protocol that simulates networking in a single JVM using a hash map. It is useful for simulation purpose and unit testing of overlay network implementations.

[2] The framework indeed supports pluggable object serializers that transform objects into binary forms. Thus it is not mandatory for objects to implement the **java.io.Serializable** interface. However, Java serialization is the default behavior.

4.3 Messaging Module

Remote method calls are useful for constructing overlay networks, because distributed routing algorithms, topology maintenance, and data migration, all need asynchronous or synchronous remote invocations. Applications may also need remote method calls to accomplish various tasks.

Although Java RMI provides convenient synchronous remote method invocations, it is based on TCP/IP network and cannot be used on custom network protocols. And asynchronous remote calls are also not well supported.

In this framework, a messaging module associates a **Messenger** with the transport layer to provide both asynchronous and synchronous calls to remote methods annotated by **MessengerCall**. Figure 3 shows the class diagram of the messaging module.

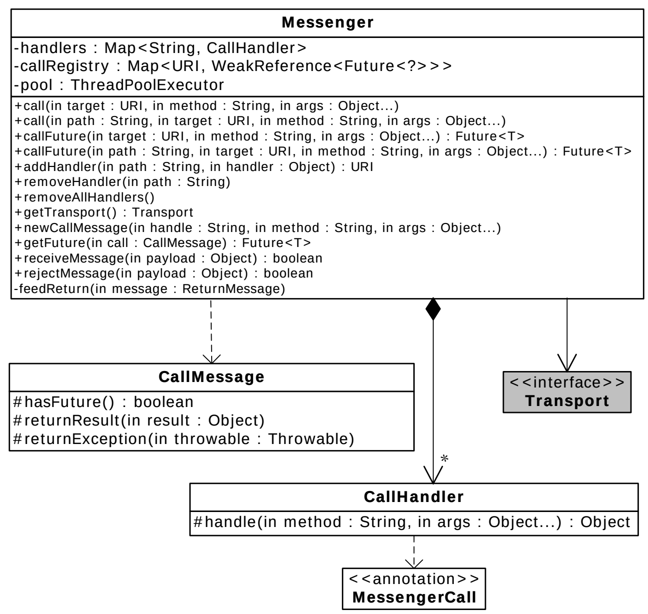

Figure 3. Messaging module class diagram

The **Messenger** class provides methods for conducting asynchronous and synchronous remote calls based on the transport layer. Objects with annotated methods are registered to the **Messenger** as **CallHandler**s. Remote calls are concurrently processed by a thread pool, and they return **Future**s that serve as the placeholder of the potential return values.

A **Messenger** object is associated with each **Transport** instance. The following code snippet acquires the associated object.

```
Messenger messenger = transport.getMessenger();
```

The **addHandler()** method is used to register objects with paths. A path uniquely identifies an object registered in a messenger. In the following code snippet, two objects are registered to “/test” and “/calc”, respectively.

```
messenger.addHandler("/test", new TestHandler());
messenger.addHandler("/calc", new CalcHandler());
```

Any object can be registered to a messenger, and methods that can be called remotely should be explicitly annotated by **MessengerCall**. Such as the **sayHello()** method in the following code snippet.

```
public class TestHandler {

  @MessengerCall
  public void sayHello(String name) {
    System.out.println("Hello, " + name);
  }
}
```

Asynchronous remote calls can be invoked by the **call()** methods with the address of the remote object, name of the method, and required parameters. The following code snippet calls the above **sayHello()** method with a parameter “World!”,

```
URI addr = URI.create("tcp://10.0.0.9:2000");
messenger.call("/test", addr, "sayHello", "World!");
```

The same method can also be invoked by specifying the full address of the object in a single URI.

```
messenger.call(
    URI.create("tcp://10.0.0.9:2000/test"),
          "sayHello", "World!")
```

The **callFuture()** methods are similar to the **call()** methods, except that they immediately return **Future** objects as the placeholders of any potential return values. Then the **Future**’s **get()** method can be invoked to wait for the remote call to return, i.e. making the call synchronous. For example,

```
Future<Void> future = messenger.callFuture("/test",
                         addr, "sayHello", "World!");
future.get();
```

The following is another example that waits three seconds for the return value, or a timeout exception will be thrown (the exception handling code is omitted).

```
Future<Double> future =
   messenger.callFuture("/calc", addr, "sqrt", 100);
double sqrt = future.get(3, TimeUnit.SECONDS);
```

In fact, the **Messenger** class keeps track of all remote calls and their **Future** objects in an internal data structure, to correlate return messages to corresponding future objects. However, it is possible that some return messages may be lost and never return. To eliminate memory leaking, the **Future** objects should be appropriately released. In this framework, we use **WeakReference** in the internal data structure to make unreferenced **Future** objects garbage collectable and automatically release them at finalization. Thus alleviate the burden of the programmers.

The **Messenger** class also provides some advanced APIs, e.g. **newCallMessage()**, **getFuture()** and **receiveMessage()**, to customize the delivery and invocation of **CallMessage**s. For example, a call message can be explicitly constructed in the following code snippet without specifying the target address:

```
CallMessage call = messenger.newCallMessage("/calc",
                            "sqrt", 100);
Future<Double> future = messenger.getFuture(call);
```

Then, the call object can be delivered to other nodes via certain algorithms, e.g. key-based routing algorithms. And the call can be explicitly invoked by using the **receiveMessage()** method.

```
// if payload is a remote call, invoke it
if (! messenger.receiveMessage(payload)) {
   // otherwise, it’s not a call
}
```

4.4 Overlay Transport Layer

The overlay transport layer abstracts overlay networks by **OverlayTransport** and **OverlayListener** interfaces, it extends the transport layer with additional support for mapping object into ID space, key-based routing, application data replication and migration. Figure 4 shows the class diagram of the overlay transport layer.

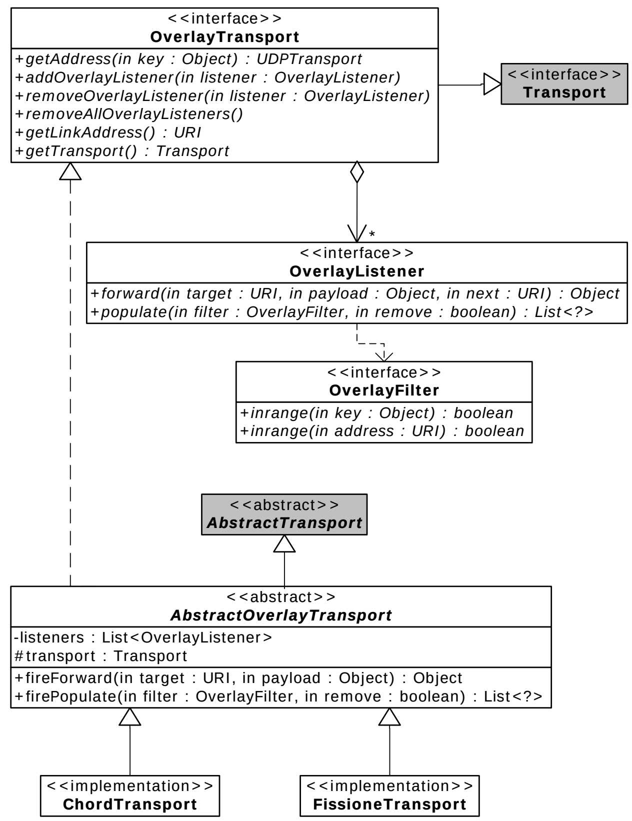

Figure 4. Overlay transport layer class diagram.

The **OverlayTransport** interface extends the **Transport** interface, so it can be used in the same way as described in section 4.2. However, the **OverlayTransport** interface also defines overlay specific operations: get the address of a key object, managing overlay related callback listeners, and acquire the address and **Transport** instance of the underlying network.

Since an overlay network needs an underlying network to operate on, it is constructed with a **Transport** instance that represents the underlying network. Moreover, peers usually join an overlay network by contacting some well-known peers, or seeds. In the following code snippet, a **ChordTransport** instance is constructed by specifying the underlying transport object, the seeds and many other Chord protocol specific parameters.

```
OverlayTransport overlay = new ChordTransport(
    transport, 80, 2, 10000, 100000, 100000, 300000,
        seeds);
```

The **open()** method must be called for a node to join the overlay network by contacting seeds via the underlying transport object, while the **close()** method will detach the current node from the overlay network. The **getAddress()** method maps a key object to an address in the ID space, while the **getLinkAddress()** method returns the address in the underlying network.

As described in section 4.2, we can add **TransportListener** to receive data, and use the **send()** method to send data in the overlay network, i.e. key-based routing. As shown in the following code snippet:

```
overlay.addTransportListener(
    new TransportListener() { ... } );

URI addr = overlay.getAddress("any key object");
overlay.send(addr, "Hello, World!");
```

The messaging module can also be used in the same way as described in section 4.3. For example:

```
Messenger messenger = overlay.getMessenger();
messenger.addHandler("/test", new TestHandler());

// on another peer
URI addr =
     URI.create("chord://1050136146584401999253070");
messenger.call("/test", addr, "sayHello", "World!");
```

There is one complication here. Since keys may be distributed to more than one peers (in case of data replication), the **send()**, **call()** and **callFuture()** methods may actually notify multiple peers. This represents the high availability of overlay networks. In this case, any related **Future** objects will wait for the first return messages.

To support the replication and migration of application data, **OverlayListener** and **OverlayFilter** interface is designed to handle overlay specific callback notifications. The following code snippet registers an **OverlayListener** implementation (as an anonymous class).

```
overlay.addOverlayListener(new OverlayListener() {
  public Object forward(URI target,
                                Object payload, URI next){
    // process the payload on the routing nodes
    return payload;
  }

  public List<?> populate(OverlayFilter filter,
          boolean remove) {
    // use the filter to get a list of object to be
    // replicated or migrated
    return ...;
  }
});
```

The **forward()** method is used to notify nodes along the path of a key-based routing process, right before sending data to the next node. It gives applications the opportunity to participate in the routing process to accomplish some application specific tasks, e.g. changing the content of the data.

The **populate()** method is used to collect application data that need to be replicated or migrated when nodes arrive or depart. Since the data are in the application and only the overlay network knows what kind of data should be replicated or migrated, they must work together to accomplish the task. So when invoking the **populate()** method, the overlay network will provide an **OverlayFilter** instance that has the knowledge of what kind of data should be collected. It also tells the application whether the collected data need to be removed, i.e. either replication or migration.

An **OverlayFilter** interface defines **inrange()** methods to tell whether a key is in some desired range in the ID space. For instance, the following code is the Chord implementation of the **OverlayFilter** that tells whether a key is in a certain interval (*start*, *end*] on the ring.

```
public class ChordFilter implements OverlayFilter {
  private final ChordID start, end;

  public ChordFilter(ChordID start, ChordID end) {
    this.start = start;
    this.end = end;
  }

  public boolean inrange(Object key) {
    ChordID id = start.hash(key);
    return id.between(start, end) || id.equals(end);
  }

  public boolean inrange(URI address) {
    ChordID id = ChordID.parse(address);
    return id.between(start, end) || id.equals(end);
  }
}
```

To facilitate the implementation of the **OverlayTransport** interface, the framework provides an **AbstractOverlayTransport** class that implements some common tasks, such as listener management and callback notification. Several overlay networking protocols are implemented, e.g. **ChordTransport** and **FissioneTransport**, and they can run on various underlying networks.

5. APPLICATIONS

Based on this framework, peer-to-peer applications can not only be easily built, but also be independent to the overlay implementations. In this section, we illustrate the building of two typical applications: a distributed hash table application and a content-based publish/subscribe service.

5.1 Distributed Hash Table Application

Distributed Hash Tables (DHTs) are distributed systems that provide operations similar to a hash table, i.e. (key, value) pairs can be stored (put) to the DHT and any participating node can efficiently get or remove the value associated with the given key.

Building DHT applications using key-based routing on peer-to-peer overlay networks is straightforward, just augmenting each peer with a local repository to store its corresponding (key, value) pairs. The local repository can be in the memory, in a file, or even in a database. Anyway, the (key, value) pairs may need to be replicated or migrated when nodes arrive or depart.

The following source code of the **DHTApp** class is a simple implementation that stores (key, value) pairs in a **HashMap** object. It provides **put()**, **get()**, and **remove()** methods to operate on the (key, value) pairs. Similar applications can be built using other local repositories.

```java
public class DHTApp implements OverlayListener {
  private OverlayTransport overlay;
  private Messenger messenger;
  private Map map =
        Collections.synchronizedMap(new HashMap());

  public DHTApp(OverlayTransport overlay) {
    this.overlay = overlay;
    this.messenger = overlay.getMessenger();
    overlay.addOverlayListener(this);
    messenger.addHandler("/dht", new DHTHandler());
  }

  public void exit() {
    messenger.removeHandler("/dht");
    overlay.removeOverlayListener(this);
  }

  public Object get(Object key) throws
     InterruptedException, IOException, ExecutionException {
   return messenger.callFuture("/dht",
     overlay.getAddress(key), "get", key).get();
  }

  public Object put(Object key, Object value) throws
     IOException, InterruptedException, ExecutionException {
   return messenger.callFuture("/dht",
     overlay.getAddress(key),"put",key,value).get();
  }

  public Object remove(Object key) throws
     IOException, InterruptedException, ExecutionException {
   return messenger.callFuture("/dht",
     overlay.getAddress(key),"remove", key).get();
  }

  public Object forward(URI target,
                        Object payload, URI next){
    return payload;
  }

  public List<?> populate(OverlayFilter filter,
                          boolean remove) {
    List list = new ArrayList();
    for(Iterator<Entry> i=map.entrySet().iterator();
         i.hasNext();) {
      Entry e = i.next();
      if (filter.inrange(e.getKey())) {
        list.add(messenger.newCallMessage("/dht",
              "put", e.getKey(), e.getValue()));
        if (remove) i.remove();
      }
    }
    return list;
  }

  // export remote methods from an inner class
  public class DHTHandler {
    @MessengerCall
    public Object put(Object key, Object value) {
      return map.put(key, value);
    }

    @MessengerCall
    public Object get(Object key) {
      return map.get(key);
    }

    @MessengerCall
    public Object remove(Object key) {
      return map.remove(key);
    }
  }
}
```

The **DHTApp** class can be constructed using any implementation of the **OverlayTransport** interface. During construction, it registers itself as an **OverlayListener** to the overlay instance. An inner class **DHTHandler** object is registered (as "dht") to the messaging module associated with the overlay instance to handle remote method calls from other peers.

In the **put(key, value)** method, first the key is mapped to an address in the ID space by the **overlay.getAddress(key)** method, and then a remote call is initiated to invoke the **put(key, value)** method of the registered **DHTHandler** object at the address. The remote call returns a **Future** object, whose **get()** method is invoked immediately to wait for the result, making the call synchronous. Similar processes occur in the **get(key)** and **remove(key)** methods.

Moreover, data replication and migration can be easily achieved by properly implementing the **populate()** method of the **OverlayListener** interface. The data that need to be replicated or migrated are wrapped into **CallMessage** objects which will invoke the **put(key, value)** method of the registered **DHTHandler** object on the nodes that will be in charge of the data.

As we can see, the DHT application does not need to know any details of the overlay implementation, while it can still employ the full power of peer-to-peer overlay networks.

5.2 Content-based Publish/Subscribe Service

Content-based publish/subscribe (pub/sub) is a system in which messages are only delivered to a subscriber if the content of those messages match criteria defined by the subscriber. Thus publishers and subscribers are fully decoupled in a pub/sub system.

The following is part of the **PubSub** class that implements a peer-to-peer content-based pub/sub system similar to [13]. This code snippet highlights the replication and migration of subscription data.

The **PubSub** class can be constructed using any implementation of the **OverlayTransport** interface. During construction, it registers itself as an **OverlayListener** to the overlay instance. An inner class **PubSubHandler** object is registered (as "/pubsub") to the messaging module associated with the overlay instance to handle remote method calls from other peers.

In the **subscribe(crt, listener)** method, first a key is obtained from the criteria object and is mapped to an address in the ID space, and then a remote call is initiated to invoke the **subscribe(sub)** method of the registered **PubSubHandler** object at the address.

```java
public class PubSub implements OverlayListener {
  private OverlayTransport overlay;
  private Messenger messenger;
  private SubscriberList subscribers;

  public PubSub(OverlayTransport overlay) {
    this.overlay = overlay;
    this.messenger = overlay.getMessenger();

    overlay.addOverlayListener(this);
    messenger.addHandler("/pubsub",
                         new PubSubHandler());
  }
  ...
```

```
...
public Subscription subscribe(EventCriteria crt,
          EventListener listener) throws
  IOException,InterruptedException,ExecutionException{
  URI addr = overlay.getAddress(criteria.getKey());
  Future<Subscriber> future = messenger.callFuture(
      "/pubsub", addr, "subscribe",
          new Subscriber(addr, crt, listener));
  return future.get();
}

public List<?> populate(OverlayFilter filter,
                  boolean remove) {
  List<CallMessage> list =
           new ArrayList<CallMessage>();

  for(Iterator<Subscriber>i=subscribers.iterator();
          i.hasNext();) {
    Subscriber sub = i.next();
    if (filter.inrange(sub.getAddress())) {
      CallMessage call = messenger.newCallMessage(
          "/pubsub", "subscribe", sub);
      list.add(call);
      if (remove) i.remove();
    }
  }
  return list;
}

// export remote methods from an inner class
public class PubSubHandler {
  @MessengerCall
  public Subscription subscribe(Subscriber sub) {
    return subscribers.add(sub);
  }
  ...
 }
}
```

In the **populate()** method, subscription data that need to be replicated or migrated are wrapped into **CallMessage** objects which will invoke the **subscribe(sub)** method of the registered **PubSubHandler** object on the nodes that will be in charge of the subscription. Again, based on this framework, the pub/sub service does not need to know any details of the overlay implementation.

6. CONCLUSIONS AND FUTURE WORK

We have presented a framework for constructing peer-to-peer overlay networks in Java. First, networking is abstracted by the transport layer that uses URIs to uniformly address peers on different underlying or overlay networks. Then, a messaging module is associated with the transport layer to support both asynchronous and synchronous remote method calls by using futures. Finally, the overlay transport layer is sketched to handle the additional issues in overlay networks, e.g. key-based routing, data replication and migration. Several overlay network protocols have been implemented in this framework. And using this framework, peer-to-peer applications such as distributed hash table applications and content-based publish/subscribe services, have been constructed, which are independent of the overlay implementations. The sources presented in this paper are available online at http://overlay.sourceforge.net/.

We are working on implementing more transport layer protocols such as HTTP and UDT [5], as well as overlay network protocols such as Pastry [11], Kademlia [8] and etc. We are also planning to integrate application layer multicast [7] into the framework.

7. ACKNOWLEDGMENTS

This work is partially supported by the National Basic Research Program of China (973) under Grant No. 2005CB321802 and Natural Science Foundation of China under Grant No. 90612009.

8. REFERENCES

[1] Androutsellis-Theotokis, S. and Spinellis, D. A survey of peer-to-peer content distribution technologies. *ACM Computing Surveys*, 36(4): 335-371, 2004.

[2] Baumgart, I., Heep, B. and Krause, S. OverSim: A Flexible Overlay Network Simulation Framework. In *Proc. 10th IEEE Global Internet Symposium (GI'07)*, Anchorage, AK, USA, May 2007. (http://www.oversim.org)

[3] Berners-Lee, T., Fielding, R. and Masinter, L. Uniform Resource Identifier (URI): Generic Syntax. *Internet Standard RFC 3986*, 2005. (http://tools.ietf.org/html/rfc3986)

[4] Gong, L. JXTA: a network programming environment. *IEEE Internet Computing*, 5(3): 88-95, 2001. (http://jxta.org)

[5] Gu, Y. and Grossman, R.L. UDT: UDP-based Data Transfer for High-Speed Wide Area Networks. *Computer Networks*, Elsevier, 51(7): 1777-1799, 2007.

[6] Li, D., Lu, X. and Wu, J. FISSIONE: a scalable constant degree and low congestion DHT scheme based on Kautz graphs. In *Proc. 24th Annual Joint Conf. of the IEEE Computer and Comm. Societies (INFOCOMM'05)*, IEEE CS Press, pp. 1677-1688, 2005.

[7] Kostas, K. and May, M. Application-Layer Multicast. *Peer-to-Peer Systems and Applications*, Springer-Verlag, LNCS 3485: 157-170, 2005.

[8] Maymounkov, P. and Mazieres, D. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. In Proc. 1st Int'l Workshop on Peer-to-Peer Systems (IPTPS'02), Springer-Verlag, LNCS 2429: 53-65, 2002.

[9] Rhea, S., Geels, D., Roscoe, T. and Kubiatowicz, J. Handling Churn in a DHT. In *Proc. USENIX Annual Technical Conference (USENIX'04)*, 127-140, 2004.

[10] Rhea, S., Godfrey, B., Karp, B., Kubiatowicz, J., Ratnasamy, S., Shenker, S., Stoica, I. and Yu, H. OpenDHT: A Public DHT Service and Its Uses. In *Proc. 2005 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM'05)*, ACM Press, pp. 73-84, 2005. (http://www.opendht.org/)

[11] Rowstron, A. and Druschel, P. Pastry: Scalable, Decentralized Object Location, and Routing for Large-Scale Peer-to-Peer Systems. In *Proc. 2nd IFIP/ACM Int'l Conf. on Distributed Systems Platforms (Middleware'01)*, Springer-Verlag, LNCS 2218: 329-350, 2001. (http://www.freepastry.org)

[12] Stoica, I., Morris, R., Liben-Nowell, D., Karger, D.R., Kaashoek, M.F., Dabek, F. and Balakrishnan, H. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1): 17-32, 2003. (http://pdos.csail.mit.edu/chord/)

[13] Zhu, Y. and Hu, Y. Ferry: a P2P-based architecture for content-based publish/subscribe services. *IEEE Trans. on Parallel and Distributed Systems*, 18(5): 672-685, 2007.