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, wang.pdl}@gmail.com, 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 the 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, Asynchronous 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 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 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 [14]. 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 accordingly when the topology changes. Many overlay protocols have been proposed to organize peers into various topologies with emphasis on different networking properties such as routing path length (FISSIONE [8]), churn overhead (Bamboo [11]), proximity metric (Pastry [13]) etc.

There are many implementations of peer-to-peer overlay networks, e.g. Chord [14], Pastry [13] and OpenDHT [12]. However, applications are often stuck to a specific overlay network implementation, because different overlay implementations usually provide very different interfaces and messaging mechanisms. For instance, 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 [5], 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 the interfaces that use URIs to uniformly address peers on different underlying or overlay networks. Then, asynchronous and synchronous messaging (remote procedure calls) 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 [14], FISSIONE [8] etc.) using this framework, and built peer-to-peer applications (e.g. a typical distributed hash table application and decentralized content-based publish/subscribe services) 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, such as integers, strings etc. Overlay networks employ globally consistent protocols to efficiently route messages based on their destination keys. To achieve this key-based routing capability efficiently, 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 together with the related application data, in case of node arrival and departure.

2.1 Topology and ID Space

Peers in an overlay network are organized into a certain topology, such as ring, mesh, torus 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). The ID space should be much larger than the number of potential nodes in a system to comfortably accommodate them without collisions.

For example, Chord uses a ring topology [14], and its ID space consists of the integers ranging 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 closest successor peer to keep the ring structure. FISSIONE uses Kautz graph topology [8], and its ID space consists of 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) to uniformly distribute them in the ID space, thus these peer-to-peer overlay networks are also called distributed hash tables (DHTs). In this case, 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 destination key should always reach its corresponding peers. Note that a key may be distributed to more than one peer in some protocol to tolerate failures such as involuntary node departures.

For example, Chord distributes a key to the peer whose ID is the closest successor of the key's ID among all the online peers [14]. FISSIONE distributes a key to the peer whose ID is the exact prefix of the key's ID [8].

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

2.3 Application 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 process by adjusting routing tables of the related peers.

For example, in Chord [14], when a new peer arrives, it must acquire a pointer to its closest successor peer, and the original peer pointing to the successor peer should update its pointer to the new peer (because the new peer is now its closest successor). 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, which is obviously undesirable.

3. RELATED WORK

As an emerging technology, 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. Developers need to decide which overlay network to use before building the applications. Then, those applications are often stuck to the chosen overlay network, or mass modifications are required to change to an alternative one.

For example, the official Chord implementation [14] provides interfaces such as **`find_successor()`** to find the successor of a key, since a key is distributed to its closest 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 [13] 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, application data replication and migration still need certain understanding of the protocol.

OpenDHT [12] 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 different data structures are not well supported, i.e. developers need to wrestle with the underlying implementation to get the job done.

All the above projects implement a single overlay protocol, and they are all based on the TCP protocol. However, there are some frameworks that can support constructing overlay networks based on various underlying transport protocols. For example, JXTA is a set of open protocols that facilitate the building of peer-to-peer networks [5]. It can be used to construct overlay networks, but there is no standard to guarantee the constructed overlay networks can be interchangeably used by applications. OverSim is a C++ overlay network simulation framework to evaluate new protocols [2]. Although it provides interfaces for key-based routing and a RPC framework for messaging, application data replication and migration mechanism are not specified.

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 standardize 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, Uniform Resource Identifier (URI) is used to uniformly address peers and resources on various underlying networks and overlay networks, as shown in Figure 1. This framework then consists of a transport layer, a messaging module, and an overlay transport layer.

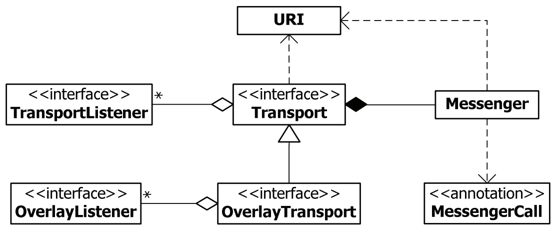


Figure 1. Overview of the framework.

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.

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

Finally, the overlay transport layer abstracts overlay networks by **`OverlayTransport`** and **`OverlayListener`** interfaces, which support mapping object into the ID space, key-based routing process, 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.

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

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 this framework, both underlying network addresses and overlay network addresses are represented using URIs. For example, a typical IP address located at TCP 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 resource’s ID in the topology. The following are example URIs for Chord and FISSIONE, respectively.

```
chord://3938476263636482923847363635252828
fission://12010210210120121202120101010210
```

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

```
tcp://10.0.0.9:2000/test
chord://3938476263636482923847363635252828/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.

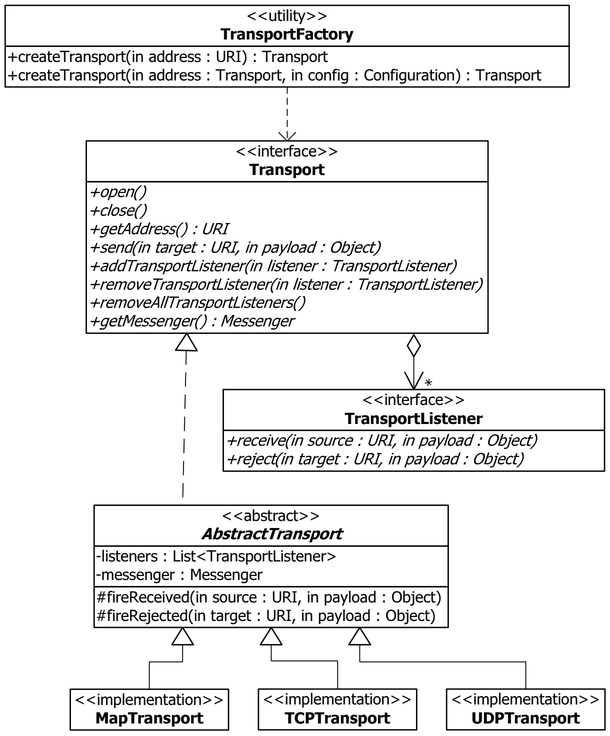
The **`Transport`** interface defines common operations for various network protocols: opening or closing the network, querying the address of a peer, sending data to a target address, managing callback listeners, etc.

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 the specified addresses.

```
Transport transport;

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



<<utility>>
TransportFactory
+createTransport(in address : URI) : Transport
+createTransport(in address : Transport, in config : Configuration) : Transport
<<interface>>
Transport
+open()
+close()
+getAddress() : URI
+send(in target : URI, in payload : Object)
+addTransportListener(in listener : TransportListener)
+removeTransportListener(in listener : TransportListener)
+removeAllTransportListeners()
+getMessenger() : Messenger
*
<<interface>>
TransportListener
+receive(in source : URI, in payload : Object)
+reject(in target : URI, in payload : Object)
<<abstract>>
AbstractTransport
-listeners : List<TransportListener>
-messenger : Messenger
#fireReceived(in source : URI, in payload : Object)
#fireRejected(in target : URI, in payload : Object)
<<implementation>>
MapTransport
<<implementation>>
TCPTransport
<<implementation>>
UDPTransport

Figure 2. Transport layer class diagram.

The `send()` method is 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. For example, 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 invoking their `reject()` method, so they can resend the data if necessary.

[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.

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, e.g. TCP, UDP and an in memory network simulator.

The `TCPTransport` class uses Java NIO to implement asynchronous communication for the TCP protocol by efficiently selecting on opening sockets channel for read and write operations. The `UDPTransport` class is a simple implementation for the UDP protocol.

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

4.3 Messaging Module

Remote method calls are useful for constructing overlay networks, because all of distributed routing algorithms, topology maintenance, and data replication and migration need asynchronous or synchronous remote invocations. Applications may also need remote procedure 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, the messaging module associates a `Messenger` object with the transport layer to provide both asynchronous and synchronous calls to remote methods annotated by the `MessengerCall` annotation. Figure 3 shows the class diagram of the messaging module.

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` class and become `CallHandler`s. Remote calls are extracted, scheduled, and will be concurrently processed by a pool of threads, and they immediately return `Future` objects that serve as the placeholder of the potential return values.

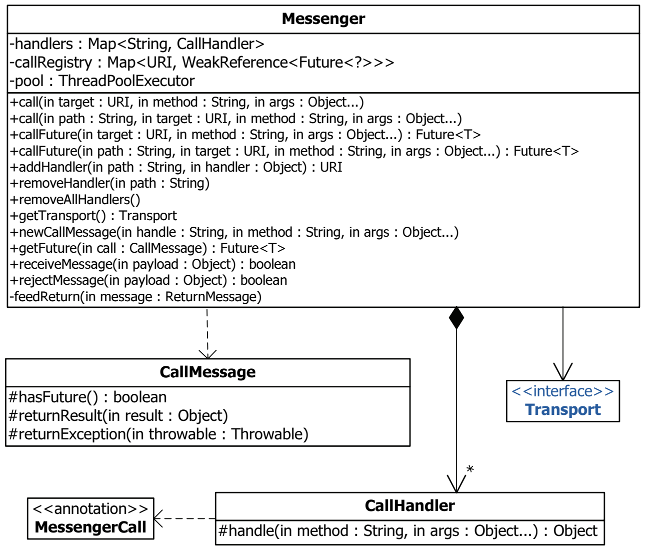
A `Messenger` object is associated with each `Transport` instance. The following code snippet acquires the associated object for later use.

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

The `addHandler()` method is used to register objects with paths. A path uniquely identifies an object registered in a messenger. For instance, two new objects are registered to "/test" and "/calc", respectively, in the following code snippet.

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

Any object can be registered to a `Messenger` object, and the methods that can be called remotely should be explicitly annotated by the `MessengerCall` annotation, such as the following `sayHello()` method.



Messenger
-handlers : Map<String, CallHandler>
-callRegistry : Map<URI, WeakReference<Future<?>>>
-pool : ThreadPoolExecutor
+call(in target : URI, in method : String, in args : Object...)
+call(in path : String, in target : URI, in method : String, in args : Object...)
+callFuture(in target : URI, in method : String, in args : Object...) : Future<T>
+callFuture(in path : String, in target : URI, in method : String, in args : Object...) : Future<T>
+addHandler(in path : String, in handler : Object) : URI
+removeHandler(in path : String)
+removeAllHandlers()
+getTransport() : Transport
+newCallMessage(in handle : String, in method : String, in args : Object...)
+getFuture(in call : CallMessage) : Future<T>
+receiveMessage(in payload : Object) : boolean
+rejectMessage(in payload : Object) : boolean
-feedReturn(in message : ReturnMessage)
CallMessage
#hasFuture() : boolean
#returnResult(in result : Object)
#returnException(in throwable : Throwable)
<<interface>>
Transport
*
<<annotation>>
MessengerCall
CallHandler
#handle(in method : String, in args : Object...) : Object

Figure 3. Messaging module class diagram

```
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, together with the 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 remote 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` objects' `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);
```

To support this asynchronous mechanism, 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, leaving garbage in the data structure. To eliminate this memory leaking, those orphaned `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 their finalization. Thus alleviate the burden of the programmers.

Unlike Java RMI, which uses interfaces to ensure the consistency of remote calls in the syntax level, our URI based remote call mechanism may induce runtime exceptions when used improperly, e.g. calling a remote method with a wrong name, or using incompatible parameters. However, our implementation of asynchronous remote calls with future support is a tradeoff between ease of use and restrictions to programmers. For example, although Java Dynamic Proxy can be used to hide the `Future` interface and provide a syntactically sound solution for asynchronous remote calls, it requires programmers to specify remote methods in a particular interface. Third party libraries, such as cglib [5], do not require the interfaces, but require the remote method's return type to be reifiable (i.e. it should be non-primitive, non-final, and has a constructor without arguments), e.g. in ProActive [4], which rule out many useful classes, e.g. `String` is a final class that cannot be extended. Our approach also has the benefit of explicit timeout control, as well as advanced composition of remote methods.

The `Messenger` class provides some advanced APIs, e.g. `newCallMessage()`, `getFuture()` and `receiveMessage()`, to customize the delivery and invocation of `CallMessage`s. For example, a `CallMessage` object 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 wrapped into other objects and delivered to arbitrary nodes, e.g. as a parameter of another remote call. Then, developers can explicitly invoke the call by using the `receiveMessage(payload)` method, which returns true when the payload is indeed an `CallMessage` object and its destination method is scheduled to be invoked, as demonstrated in the following code snippet.

```
// if payload is a remote call, invoke it
if (! messenger.receiveMessage(payload)) {
  // otherwise, it's not a remote 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 the ID space, key-based routing process, application data replication and migration. Figure 4 depicts main classes of the overlay transport layer.

The `OverlayTransport` interface extends the `Transport` interface, so it can be used in the same way as described in Section 4.2. Moreover, the `OverlayTransport` interface defines additional overlay specific operations, e.g. getting the address of a key object and managing overlay related callback listeners.

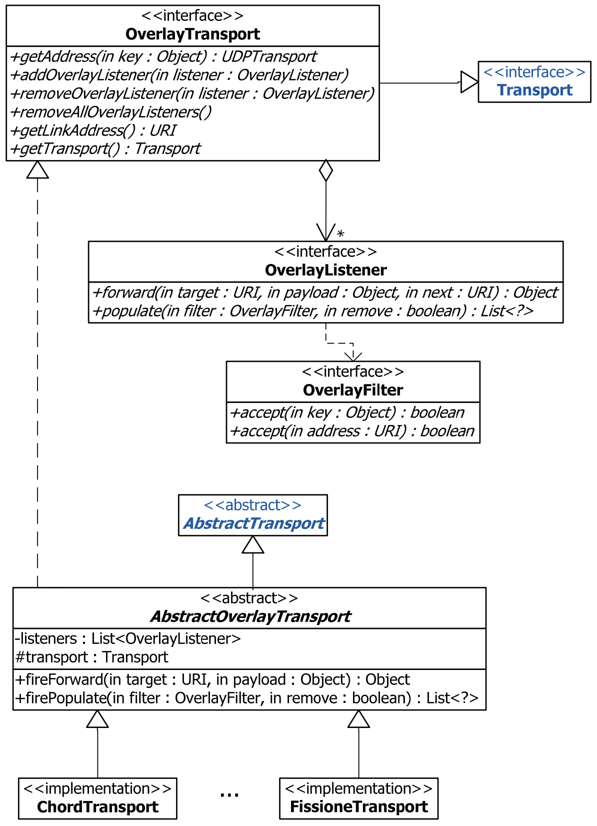

<<interface>>
OverlayTransport
+getAddress(in key : Object) : UDPTransport
+addOverlayListener(in listener : OverlayListener)
+removeOverlayListener(in listener : OverlayListener)
+removeAllOverlayListeners()
+getLinkAddress() : URI
+getTransport() : Transport
<<interface>>
Transport
*
<<interface>>
OverlayListener
+forward(in target : URI, in payload : Object, in next : URI) : Object
+populate(in filter : OverlayFilter, in remove : boolean) : List<?>
<<interface>>
OverlayFilter
+accept(in key : Object) : boolean
+accept(in address : URI) : boolean
<<abstract>>
AbstractTransport
<<abstract>>
AbstractOverlayTransport
-listeners : List<OverlayListener>
#transport : Transport
+fireForward(in target : URI, in payload : Object) : Object
+firePopulate(in filter : OverlayFilter, in remove : boolean) : List<?>
<<implementation>>
ChordTransport
...
<<implementation>>
FissioneTransport

Figure 4. Overlay transport layer class diagram.

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, many Chord protocol specific parameters, as well as the seeds.

```
OverlayTransport overlay = new ChordTransport(
    transport, 80, 2, 10000, 100000, 100000, 300000,
         seeds); // ring size = 80, successor = 2,
   // various timeout values, and the seed URI list
```

The **`open()`** method must be called for a node to join the overlay network by contacting seed peers via the underlying transport object, while the **`close()`** method will detach the current node from the overlay network. The **`getAddress(key)`** 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, via key-based routing, as shown in the following code.

```
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, to support asynchronous remote method calls on the overlay network! 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 for fault tolerance), the **`send()`**, **`call()`** and **`callFuture()`** methods may actually notify multiple peers. This represents the high availability of overlay networks. In this case, **`Future`** objects will wait for the first return messages, and discard all its subsequent 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 (remove == false) or
    // migrated (remove == true)
    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. adjusting 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 **`accept()`** 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 accept(Object key) {
    ChordID id = start.hash(key);
    return id.between(start, end) || id.equals(end);
  }

  public boolean accept(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`, which can run on various underlying networks provided by the framework.

4.5 Performance

The abstractions in the framework will surely induce performance overheads, e.g. messages may be propagated across extra layers to reach their recipients, increasing transport latency and processor load. However, certain measures are used by components in the framework to compensate these overheads in a certain degree. For instance, messages are often queued in the memory to increase the I/O performance, and thread pools are carefully managed to further boost the overall throughput. Besides, these queues and thread pools can be tuned using environment variables.

Moreover, optimized I/O operations can be developed by experts and built into the framework. For example, the Java NIO implementation of the `TCPTransport` class encapsulates the efficient and rather complex selector/channel operations, together with a configurable pool of threads, can achieve better performance and scalability than those based on the simple Java Socket library.

5. APPLICATIONS

Based on this framework, peer-to-peer overlay networks can be easily built, while providing uniform interfaces. Peer-to-peer applications can be easily developed on these interfaces. In this section, we illustrate the applications of this framework by implementing the Chord overlay network and building a typical distributed hash table application.

5.1 Chord Overlay Network

Chord is an overlay network that uses a ring topology, in which each peer has a pointer to its closest successor peer. It distributes a key to the peer whose ID is the closest successor of the key's ID among all the online peers [14]. The key-based routing process of Chord relays the finding of the closest successor of a given ID on a chain of intermediate peers that approximate to the closest one.

The code at the right column constitutes part of the `ChordTransport` class that implements the Chord overlay protocol by extending the `AbstractOverlayTransport` class.

```
public class ChordTransport
        extends AbstractOverlayTransport {
  private ChordID id, succ;
  private ChordHandler handler;
  private URI[] seeds;

  @Override
  public void open() throws IOException {
    super.open();

    Messenger messenger = transport.getMessenger();
    messenger.addHandler("/chord",
                  handler = new ChordHandler());
    id = new ChordID(...);

    for (URI seed: seeds) if (join(seed)) return;
  }

  private boolean join(URI seed) {
    Messenger messenger = transport.getMessenger();
    CallMessage ping = messenger.newCallMessage(
        "/chord", "ping");
    Future<ChordID> future =
        messenger.getFuture(ping);

    try {
      messenger.call(seed, "find_succ", id, ping);
      succ = future.get(timeout,
             TimeUnit.MILLISECONDS);
      return true;
    } catch (Exception e) {
      return false;
    }
  }

  public void send(URI target, Object payload)
      throws TransportException {
    handler.find_succ(
        ChordID.parse(target), id, payload);
  }

  // export remote methods using an inner class
  public class ChordHandler {
    @MessengerCall
    public ChordID ping() { return id; }

    @MessengerCall
    public void find_succ(ChordID source,
              ChordID target, Object payload) {
      Messenger messenger = transport.getMessenger();
      if (succ == null) { // single node ring
        succ_accept(source, payload);
      } else if (new ChordFilter(id, succ)
                   .accept(target)) { // bingo!
        messenger.call(succ.address,
          "succ_accept", source, payload);
      } else { // relay to the next node
        ChordID next = find_closest_pred(target);
        payload = fireForward(target.toURI(),
                  payload, next.address);
        messenger.call(next.address,
          "find_succ", source, target, payload);
      }
    }

    @MessengerCall
    public void succ_accept(ChordID source,
               Object payload) {
      Messenger messenger = transport.getMessenger();
      if (! messenger.receiveMessage(payload))
        fireReceived(source.toURI(), payload);
    }
  }
}
```

The **open()** method is overridden to register a **ChordHandler** object (as "/chord") to the messaging module associated with the *underlying* network, and generate its identity in the ring using some credential information (e.g. the underlying networking address). Then, it tries to join the overlay network by contacting the seed peers one by one using the **join()** method.

A Chord peer joins the network by finding the closest successor peer of its ID. Since the new peer cannot use its own routing capability yet before it joins the network, the routing is delegated to the seed peer. The **join()** method construct a **CallMessage** object, **ping**, and remotely calls the **find_succ()** method of the seed peer to route the **ping** object to its closest successor, which will hopefully report its ID back. When such a response is returned before the specified timeout, the new peer can complete the process of joining the network.

Once a peer successfully joins the network, the **send()** method can be used to conduct key-based routings. The routing process is simply delegated to the handler's **find_succ()** method, which contains the routing algorithms of Chord: if the current peer is the only node in the network, the routing process trivially ends by invoking the peer's **succ_accept()** method; if the target locates between the current peer's ID and its successor's ID, the routing process ends by invoking the **succ_accept()** method of the successor peer; otherwise, the routing process should be relayed by invoking **find_succ()** method of the closest predecessor (known by the current peer) of the target.

In the **succ_accept()** method, the received payload is either a **CallMessage** object that encapsulates remote method calls in the overlay layer, or some application data that should be propagated to the interested listeners.

As shown by the example, complex distributed algorithms can be elegantly implemented using this framework.

5.2 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 retrieve (get) the value associated with the given key.

Building DHT applications using the key-based routing process 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 across the nodes when they arrive or depart.

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

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.

```
public class DHTApp implements OverlayListener {
  private OverlayTransport overlay;
  private Messenger messenger;
  private Map store =
        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=
         store.entrySet().iterator(); i.hasNext();) {
      Entry e = i.next();
      if (filter.accept(e.getKey())) {
        list.add(messenger.newCallMessage("/dht",
              "put", e.getKey(), e.getValue()));
        if (remove) i.remove();
      }
    }
    return list;
  }

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

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

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

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.

6. CONCLUSIONS AND FUTURE WORK

We have presented a framework for constructing peer-to-peer overlay networks in Java. First, peer-to-peer 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, e.g. Chord [14] and FISSIONE [8]. Moreover, peer-to-peer applications such as distributed hash table application and decentralized content-based publish/subscribe services based on Ferry [15], have been constructed using this framework, 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 [7] to penetrate firewalls or NAT networks, as well as various overlay network protocols such as Pastry [13], Kademlia [10] etc. We are also planning to integrate application layer multicast [9] into the framework, which can improve the performance of many applications.

7. ACKNOWLEDGMENTS

This work is partially supported by the National Basic Research Program of China (973) under Grant No. 2005CB321802 and National Natural Science Foundation of China under Grants No. 90612009 and No. 60725206.

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] Caromel, D., di Costanzo, A. and Mathieu, C. Peer-to-peer for computational grids: mixing clusters and desktop machines. *Parallel Computing*, Elsevier, 33(4-5): 275-288, 2007. (http://proactive.inria.fr/)

[5] Code Generation Library (cglib) http://cglib.sourceforge.net

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

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

[12] 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/)

[13] 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)

[14] 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/)

[15] 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.