Distributed Protocol Factory 1.0 Component Specification

1. Design

This component provides a factory for accessing a distributed protocol and a default protocol for supporting distributed systems. This protocol is a data transfer protocol for ensuring all nodes in a group receive an update, or none receive the update. It is intended to be used by a variety of application types.

The component will support a factory method for loading the protocol, so that specialized protocols can be substituted for the default protocol, when necessitated by the needs of the application.

This design provides the following additional functionalities:

Message Authentication – The design provides framework authentication. This framework can be used either by the protocol itself or calling applications in the future.

Membership Authentication – The design provides a method to allow group change. The membership service provided by the design includes the authentication function. Actually, this function is not provided by the design directly. This function is provided by the JXTA framework,

Rich Services And Easy Design – Since the design utilizes the JXTA framework, lots of low level protocols dealing with peer-to-peer communications and network structure have been well defined and implemented. The design can easily utilize very useful core services provided by JXTA framework, such as: discovery service, membership service, route resolver service and etc. Thus, the design and development only need to focus on how to utilize those services to provide “synchronized” data transfer to meet the requirements.

Great Extensibility – This additional function comes with the design concept. The design clearly divides the protocol into three abstractions, each abstraction provides its own extensibility. The design presents three packages, and each of them represents a layer of abstraction. The abstractions are: message, service, and protocol. Message defines the query and response formats, which are the container of information transferred among peers in a group. Service provides a logic network layer, and provides means to send and receive the defined messages among peers. Protocol utilizes designed service to transfer and receive the messages. Thus protocol does not need to worry about data transmission at all, it only needs to focus on implementing algorithms to provide synchronized data transfer.

Support Peers In Both WAN And LAN – This is an advantage of utilizing JXTA frameworks. The protocol presented by the design is built on top of Data Transfer Service. The Data Transfer Service is built of top of Route Resolver Service provided by the JXTA frameworks. Route Resolver Service can send the message to peers located both in WAN and LAN, even behind the firewall.

None-blocking Fashion: The design supports synchronized data transfer, which means the component users do not need to wait for the results after sending the message, and they will be notified by this component if any result comes back. Thus, the use of this component can be very efficient and flexible.

1.1 Industry Standards

JAVA 1.4

JXTA 2.0

1.2 Design Patterns

Delegate – The involved classes are DataTransferService interface and its subclasses, and QueryHandler interface in net.jxta.resolver package. Design utilizes delegate pattern to delegate the function of sending queries and responses.

Listener Pattern – The design uses the listener pattern in both com.topcoder.network.synchronization package and com.topcoder.network.synchronization.service package to make the system work in a non-blocking fashion, which means after sending the data, the calling application do not need to wait for the results, and the results will be sent via a callback function. The pattern is based on these simple concepts: events, event producer, and event listener. Events are messages that are sent from one object to another. The component that sends the event is said to "fire" the event, while the component that receives the event is said to "handle" the event. An event producer is a class that fires events. It also has the ability to add and delete event listeners (components that receive events). Event listeners, also known as event consumers, "listen" for an event. In programmatic terms, a method on the event listener object is called when an event that it is listening for is fired

Factory Method Pattern – The involved classes are DistributedProtocolFactory interface and its subclasses, DataTransferSynchronizationProtocol abstract class and its subclasses. This pattern is used to create protocols.

Creation Method Pattern – This pattern is used by ApplicationOperationResult class. This class can represent both valid result and invalid result. In order to make creation process easier (save the trouble of selecting proper constructors and passing in correct parameters), this class employs creation method pattern. Although creation method pattern makes creation nonstandard, it makes easy creations.

Aggregate Enforcer Pattern – This pattern is used by all classes in com.topcoder.network.synchronization.message package. The Aggregate Enforcer pattern recommends that when an object is constructed, it must be constructed in full. That means that when a class is instantiated, all of its member variables representing the set of constituting objects must also be initialized. The idea is to make sure that an object is created in full or is not created at all. In Java, declaring a member variable as final ensures that the variable gets initialized fully as part of the object constructor. This pattern makes it easy to keep the classes employing it thread-safe.

1.3 Required Algorithms

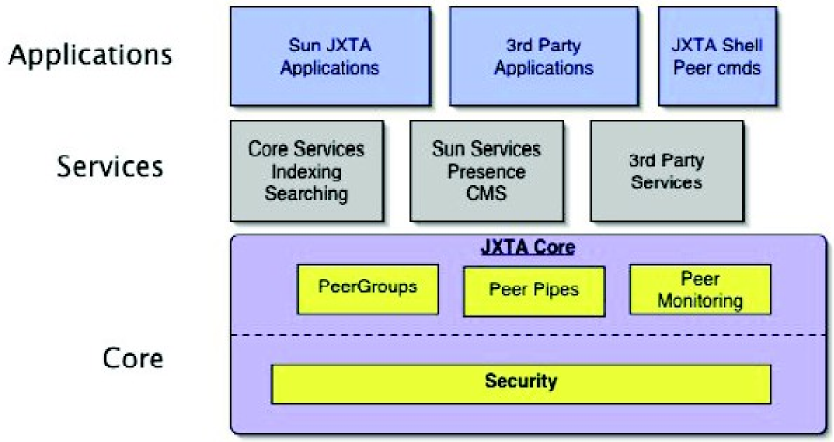
The classes provided by the component are relatively easy and do not implement very complex logic. Most algorithms are explained in the form of sample code or pseudo code in the documentation tab of each class method. Algorithms need to be further explained are listed as follows:

Utilizing JXTA Frameworks – The design utilizes the defined and well implemented peer-to-peer network structure of JXTA frameworks. This is allowed by PM in the development forum. To help developers understand how this component works, it is essential to introduce JXTA frameworks first. The important concepts are briefly explained below. More detailed info can be found at http://www.jxta.org

None-Centralized, XML based framework.

Unlike Jini, JXTA does not use a centralized server to locate services on the network. Instead, JXTA relies on XML rather than object serialization to exchange structured data and discovers services across all peers on the P2P network. (XML supports transferring binary data by using algorithms like base64 encoding.

JXTA software structure



JXTA provides core services, such as membership service, discovery service, and etc. ***This component utilizes JXTA core services to define our own Data Transfer Service and then use our own Data Transfer Service to build the required protocols.***

JXTA Peers:

JXTA has three types of peers: simple peer, Rendezvous peers, and Router peers. A simple peer is designed to serve a single end user, allowing that user to provide services from his device and consuming services provided by other peers on the network. Taken literally, a rendezvous is a gathering or meeting place; in P2P, a rendezvous peer provides peers with a network location to use to discover other peers and peer resources. One quick note here, a rendezvous peer is not a server. Any node can become a rendezvous peer. The JXTA defines, when a peer join a group, if he can not find a a rendezvous peer, it becomes a a rendezvous peer and get all necessary information from JXTA core discovery service. A router peer provides a mechanism for peers to communicate with other peers separated from the network by firewall or Network Address Translation (NAT) equipment.

JXTA Peer Groups:

Peer groups divide the P2P network into groups of peers with common goals. Peer group members can provide redundant access to a service, ensuring that a service is always available to a peer group as long as at least one member is providing the service.

JXTA services:

Services provide functionality that peers can engage to perform “useful work” on a remote peer. This work might include transferring a file, providing status information, performing a calculation, or basically doing anything that you might want a peer in a P2P network to be capable of doing. In JXTA, services can be divided into two categories. One is Peer Services, offered by a particular peer on the network to other peers. The other is Peer Group Service, offered by a peer group to members of the peer group. The core group services are Discovery service, Resolver service, membership service, access service, and pipe service.

Services can be either preinstalled into a peer or loaded from the network. The process of finding, downloading, and installing a service from the network is similar to performing a search on the Internet for a web page, retrieving the page, and then installing the required plug-in. This will be done automatically when a peer joins a new group.

The component designs Data Transfer Service as a Group Service, thus all members in the group will receive the query if any peer sends a query in the group. Here we need to make a distinction between query and response. In this component, Data Transfer Service utilizes JXTA route resolver core service to send query and response. The route resolver service guarantees that the query will be received by all peers in the group, but the response will only be sent to the query sender, which meets the requirements of this component perfectly.

JXTA Modules:

Modules provides a generic abstraction to allow a peer to instantiate a new behavior. As peers browse or join a new peer group, they may find new behaviors that they may want to instantiate. For example, when joining a peer group, a peer may have to learn a new search service that is only used in this peer group. The module abstraction includes a module class, module specification, and module implementation. Modules are used by peer groups services.

JXTA IDs

Peers, peer groups, pipes and other JXTA resources need to be uniquely identifiable. A JXTAID uniquely identifies an entity and serves as a canonical way of referring to that entity. Currently, there are six types of JXTA entities which have JXTA ID types defined: peers, peer group, pipes, contents, module classes, and module specifications. URNs are used to express JXTA IDs. URNs are a form of URI that “... are intended to serve as persistent, location- independent, resource identifiers”. Like other forms of URI, JXTA IDs are presented as text. An example JXTA peer ID is:

urn:jxta:uuid-

59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903

Design does not use Top Coder component to generate Ids, because it is much easier to use JXTA implementations to generate UUIDs.

JXTA Network Structure Maintain

JXTA membership core service utilizes other core services to maintain a group of peers, making sure that services provided by the group will be received by the peer members. ***Thus, this component does not need to worry how to maintain none-centralized network structure at all.***

JXTA Network Commuications

JXTA resolver service utilizes other core services, such as Peer Endpoint Protocl to find routes (paths) to destination ports on other peers. ***Thus, this component fully relies on the core service provided by JXTA to handle network communication protocols.***

JXTA Membership Service

JXTA has a core membership service, which can meet all “Group” related requirements of this component. ***Thus, design does not actually handle move group function at all. Design just utilizes the functions provided by membership service, such as join a group, apply for joining a group, and leave a group, and etc.***

Providing Synchronized Data Transfer – Now we come to how the design provides synchronized data transfer. The key points are explained below:

None-blocking Fashion

The design utilizes listener pattern to make the system work in a none-blocking fashion. Design actually employ listener pattern twice. One is used in DefaultDataTransferService class. This approach used here has a great advantage: the service users do not need to wait for the response after sending the query, especially when it would take a while for the query reach all peers in the group and being responded by all peers. The DataTransferSynchronizationProtocol also uses listener pattern. Thus when protocol users send a message through the protocol, they do not have to wait for the results. Instead, they will be notified when the result comes out. ***This requires all protocol user applications implement DataTransfeSynchronizationListener interface.***

Synchronized Requirements

DataTransferSynchronizationProtocol class and its subclasses incorporates all logics described here. Design interpret “Syncrhonized” as : one meaning is that either all-peers or none-peers in the group receive the message; the other meaning is that for those messages do reach other peers, the sequence of received messages should be the same among all peers. Since this design utilizes listener pattern, a more accurate description of the second meaning would be: for those messages do reach other receivers the component will notify component users to receive the messages in the same order. The design fulfills both interpretations above.

Synchronization Algorithm

The basic idea is very simple. Design divided the message sending/receiving into two phases: preparation phase, and commit phase. The procedure from query sender's point of view is as follows:

When a protocol user application use the protocol send a message, the protocol first put the message in a query and send it to ask nodes in the group to prepare.

After receiving the preparation query, the nodes would save the message in their cache, and send a response indicating whether the preparation phase is done or failed.

The node who sends the query would count how many responses indicating query is successfully received, and how many responses indicating query is not successfully received.

Situation A:
If the number of successful responses reaches a certain percentage (defined by fault tolerance ratio), then the node would send out a query for the same message to ask peers to commit the message. After receiving the commit query, all peer nodes would try to notify each listener that a new message comes, and needs to be processed. If any error occurs when messages are processed by the application or protocol, the errors are included in an response and the response is sent to the query sender.

Now the query sender counts how many responses indicating query is successfully committed, and how many responses indicating query is not successfully committed.

> Sub-situation A:
> If the number of successful responses reaches a certain percentage, (defined by fault tolerance ratio), the node will notify the calling application that the message has been sent successfully.
>
> Sub-situation B:
> If the number of failed responses reaches a certain percentage, (calculated with fault tolerance ratio), the node will notify the calling application that the message was failed to sent with all errors from the response.
>
> Sub-situation C:
> If after a certain amount of time (defined by message life time property), neither the sub-situation A Nor B occurred, the message is expired. The node would delete the message from cache, and send a query to peers to ask them to abort the message. Responses of this query is not monitored. Then the node would notify the protocol user (listener) that the message has been aborted

Situation B:
If the number of failed responses reaches a certain percentage (calculated with fault tolerance ratio), then the node would notify the calling application that the message was failed to sent with all errors from the response

Situation C:
If after a certain amount of time (defined by message life time property), neither the situation A Nor B occurred, the message is expired. The node would delete the message from cache, and send a query to peers to ask them to abort the message. Responses of this query is not monitored. Then the node would notify the protocol user (listener) that the message has been aborted

Now we will describes what a receiver should do to complement the above algorithms.

Situation A: receive a preparation query

After receiving the preparation query, the node would save the message in its cache, and sends a response indicating whether the preparation phase is done or failed.

Situation B: receive a commit query

After receiving the commit query, the node would check the cache, suppose the cache is a sorted list, sorting by the preparation query time-stamp.

```
if the query to be committed  is  at the head of the list {
        step 1: we invoke the listener, notify the listener to commit the message. Then we send a response to message sender indicating if the operation succeeds or fails.
        step 2: we check the next element in the list to see if it is ready to be committed. If it is,   we repeat step 1 for this element. If it is not ready, we check to see if the element is expired (exceeds the message life time). If it is, we delete it and check the next element until we find an entry, which is not expired, and not ready to commit.
}
if the query is not at the head of the list {
        step 1: we set the query is ready to commit. This tells that this query is ready to commit, but we are waiting for the query on the head of the list to be committed first.
        Step 2: Then we start from the head, check if the head entry is expired, If it is, we delete it. Then we check the next entry in the sorted list, if it is ready to commit, we commit it, if it is expired, we delete it. This process continues until we find an entry is not expired and not ready to commit.
}
```

Situation C: receive an abortion query

After receiving the abortion query, the node would delete the previously saved preparation query from the cache, and notify the message is aborted. No response needs to be sent for this query.

Now, the algorithms are presented completely. The algorithms described above meets the requirements discussed in the **Synchronized Requirements** section. Developers also need to know in this component:

1) dataTransferQueryEvent(event:DataTransferQueryEvent) method of DefaultDataTransferSynchronizationProtocol class implements the logics describing what a sender should do (except sending the preparation query part, which is implmented by sendSynchronizedData(data:String) method.)

2) dataTransferResponseEvent(event:DataTransferResponseEvent) method of DefaultDataTransferSynchronizationProtocol class implements the logics describing what a receiver should do.

Component Architecture – The component consists of three packages. The message package consists the classes used as message containers to hold the actual message sent by the protocol users. The service package defines the interface of DataTransferService, and provide a default implementation. Since we use listener pattern to implement the Data Transfer Service, all related events and listener interfaces are also defined in that package. The last package consists the protocol abstract class and a default implementation. Since the protocol also utilizes listener pattern to provide non-blocking services, all related events and listener interface are defined in the package too. In order to meet the requirement, some factory classes also provided in that package to create a protocol on the fly.

Group Moving – The component utilizes the membership core service to provide this function. The following pseudo code shows the ideas:

```
// resign current membership
MembershipService oldMembership = currentGroup.getMembershipService();
oldMembership.resign();

// Generate the credentials for the new Peer Group, grp holding the new PeerGroup
StructuredDocument creds = null;
AuthenticationCredential authCred = new AuthenticationCredential( grp, null, creds );

// Get the MembershipService from the new peer group
MembershipService membership = grp.getMembershipService();

// Get the Authenticator from the Authentication creds
Authenticator auth = membership.apply( authCred );

// join the group
if (auth.isReadyForJoin()){
        membership.join(auth);
} else{
        // unable to join the group
}
```

Fault Tolerance Ratio – The component provides APIs for protocol users to set fault tolerance ratio. The valid range of the fault tolerance ratio is between (0,1) inclusive. It is used in the following manner.

ratioA = the number of received successful responses / total number of peers in the group

if ratioA is greater than the fault tolerance ratio, we say the message was sent successfully

rratioB = the number of received failed responses / total number of peers in the group

if ratioB is greater than (1 - fault tolerance ratio), we say the message was failed.

1.4 Component Class Overview

Below is a brief overview of the classes in this component. Please refer to the class diagram's documentation tab for a more complete overview of each class. For the external classes used in the component, please consult the related top coder documentation for a more detailed and comprehensive description.

Package com.topcoder.network.synchronization

DistributedProtocolFactory (interface):

This interface defines the factory methods used to create new DataTransferSynchronizationProtocol

DataTransfeSynchronizationListener (interface):

An interface to encapsulate an object that listens for notification from the DataTransferSynchronizationProtocol.

DefaultDistributedProtocolFactory (class):

This class is a protocol factory. It provides APIs to allow user applications to create a default protocl

DataTransferSynchronizationEvent (class):

This class is the container for DataTransferSynchronization events. It holds three parameters, one indicates the code of the event, one contains the message and another contains all error messages both from application and protocol if there is any. This class is used by DataTransferSynchronizationListener

ResponseCounter (class):

This class is used to count how many responses have been received for the query message sent by this peer. It has four parameters, one contains the message actually sent. The others are error messages received from the peer of this query, the number of successful responses and the number of failed responses.

DataTransferSynchronizationProtocol (abstract class):

This class is the meat of the design. It defines a protocol framework used to send synchronized data to group peers. It can be extended to utilizes different algorithm to guarantee that the message is received either by all peers or none peers.

DefaultDataTransferSynchronizationProtocol (class):

This class extends DataTransferSynchronizationProtocol abstract class, and provides implementations of the abstract methods defined in the super class. It is a concrete protocol implementation that guarantees that either none-all peers in the group receives the message

ApplicationOperationResult (class):

This class models the application operation result after the protocol sends the data/info to the applications. It is used as a return object of callback method dataTransferSynchronizationEvent() in the DataTransfeSynchronizationListener class

Package com.topcoder.network.synchronization.message

DataTransferQueryMsg (abstract class):

This class defines the Data Transfer message "Query". The default behavior of this abstract class is simply a place holder for the data transfer query fields.

DataTransferResponseMsg (abstract class):

This class defines the Data Transfer message Response. The default behavior of this abstract class is simply a place holder for the data transfer response fields.

DataTransferQuery (class):

This class extends DataTransferQueryMsg abstract class, implements the getDocument method to return the xml Document representation of the query. It also provides a toString() method to return the xml string representation of the query.

DataTransferResponse (class):

This class extends DataTransferResponseMsg abstract class, implements the getDocument method to return the xml Document representation of the response. It also provides a toString() method to return the xml string representation of the response.

ErrorMessage(class):

This class models the Error Message used in the DataTransferResponse. It contains two sets of properties. The first set of properties are constants, representing the error level. The second set of properties are: type indicating the error type, and message containing the actual message.

Package com.topcoder.network.synchronization.service

DefaultDataTransferService(class):

The implementation of the DataTransferService interface. This service builds on top of the Resolver core service provided by JXTA framework to provide the function of transferring the message to the peers in the same group. This service provides two set of listeners, one for query, one for response.

DataTransferResponseEvent(class):

This class is the container for DataTransferResponse events. The source of the event is the Endpoint address of the responding peer

DataTransferQueryEvent(class):

This class is the container for DataTransferQuery events. The source of the event is the Endpoint address of the responding peer

DataTransferQueryListener(interface):

An interface to encapsulate an object that listens for notification from the DataTransferService of newly arrived query messages.

DataTransferService(interface):

An interface for the DataTransferService. This interface defines the operations that a developer can expect to use to manipulate the DataTransferService regardless of which underlying implementation of the service is being used

DataTransferResponseListener(interface):

An interface to encapsulate an object that listens for notification from the DataTransferService of newly arrived response messages.

1.5 Component Exception Definitions

NullPointerException

This represents some nullable field (String, object, etc) was passed to a function that cannot handle null values. Almost (there are some exceptions) any class that deals with a String or Object will throw this exception when a null value is encountered. The methods that throw this exception are clearly marked in the tags section of the documentation tab.

IllegalArgumentException

This exception is thrown when the parameters passed to a function is an empty string, an empty Set, or an empty Map. Almost (there are some exceptions) any class that deals with a String, a Set or a Map will throw this exception. The methods that throw this exception are clearly marked in the tags section of the documentation tab.

DataTransferServiceException

This is a custom exception. It extends the java.lang.Exception. It is thrown if any error occurs when using the DataTransferService. It wraps all thrown custom exceptions, such as exceptions thrown by ResolverService.

DataTransferMessageBuildingException

This is a custom exception. It extends the java.lang.Exception. It is thrown if any error occurs when building the document. This exception is thrown by getDocument() method of DataTransferQuery class, DataTransferQueryMsg class, DataTransferResponse class, DataTransferResponse class, and ErrorMessage class.

DataTransferSynchronizationProtocolException

It extends the java.lang.Exception. It is thrown if any error occurs when using/building the DataTransferSynchronizationProtocol. It wraps all custom exceptions thrown by calling methods, such as PeerGroupException when the default peer group can not be created.

1.6 Thread Safety.

This component is thread-safe. The thread-safety discussion is based on the thread-safety of each package.

Package com.topcoder.network.synchronization.message

This package is thread-safe, because every class in this package is thread-safe. All classes in the package are immutable. They do not modify their states at all.

Package com.topcoder.network.synchronization.service

This package is thread-safe, because every class in this package is thread-safe.

The only mutable class in this package is DefaultDataTransferService class. Its thread-safety is. ensured by the following:

1) This class will only be initialized by the JXTA framework, which invokes the following methods in order: empty constructor, init(), startApp(). Then the instance is availabe for others to use. Thus, this class only has two real mutable attributes: esigeredResponseListeners, and resigeredqueryListeners.

2) We use synchronized key word to declare methods dealing with those mutable attributes.

We also need to know the subclasses of DataTransferService interface are expected to be thread-safe, since they might work under multiple threads environment.

Package com.topcoder.network.synchronization.

This package is thread-safe, because every class in this package is thread-safe.

The ApplicationOperationResult class, DataTransferSynchronizationEvent class, and DefaultDistributedProtocolFactory class are thread safe, since they are immutable.

ResponseCounter class is thread-safe, since we use synchronized key word to declare all public methods dealing with the counter property.

The subclasses of DataTransferSynchronizationProtocol abstract class are expected to be thread-safe. It uses "synchronized" key word to declare all non-abstract methods.

DefaultDataTransferSynchronizationProtocol class is a subclass of DataTransferSynchronizationProtocol abstract class. It is thread safe because it defines all methods as “synchronized”

We also need to know that the subclasses of DistributedProtocolFactory interface are expected to be thread-safe.

2. Environment Requirements

2.1 Environment

Development language: Java1.4

Compile target: Java1.4

Multiple runtime environments

WebLogic

JBoss

JVM 1.4

2.2 Top Coder Software Components

None

Why the design does not use Simple Cache Component

The DataTransferSynchronizationProtocol class does use caches, but the design does not use Simpe Cache component. The reason is simple cache component would periodically clean all contents in the cache, we do not have control of when to clean an individual content in the cache. Most importantly, we want to clean cache in a event-driven fashion, which can not be accomplished by Simple Cache component.

Why the design does not use Service Manager Component

Again, the design needs to clean cache or invoke the discovery service to find how many peers in the group in event-driven fashion. The Service Manager component can not fulfill this purpose.

Why the design does not use Socket related components

The design utilizes JXTA frameworks to handle network communications, there is no direct contact with sockets in this component.

2.3 Third Party Components

JXTA 2.0

3. Installation and Configuration

3.1 Package Name

com.topcoder.network.synchronization

3.2 Configuration Parameters

None

3.3 Dependencies Configuration

Before utilizing the DataTransferSynchronizationProtocol, We need to publish DataTransferService in the groups that we want to use the protocol.

4. Usage Notes

4.1 Required steps to test the component

Extract the component distribution.

Follow 3.3 Dependencies Configuration

Execute ‘nant test’ within the directory that the distribution was extracted to

4.2 Required steps to use the component

Most likely, usage scenario of this component could look like as follows:

1) User application uses protocol factory to create a DefaultDataTransferSynchronizationProtocol object.
2) Adjust the protocol parameters according to our needs. It is ***strongly*** recommended that all peers in a group keep the same parameters. Although different peers can have different parameters to make the usage of the protocol more flexible.
3) User application then register itself as a listener to the DefaultDataTransferSynchronizationProtocol object.
4) Perform event handling in the dataTransferSynchronizationEvent() method.
5) Send messages at will.

Note: the user application must implement DataTransfeSynchronizationListener interface, and register itself as a listener of the protocol.

4.3 Demo

The code below demonstrates the usage scenario mentioned above. To help developers to understand how to use JXTA frameworks in order to utilize the protocol provided by this component, the demo also includes some sample code related with JXTA frameworks, but has nothing to do with the functionalities of this component. Those codes are colored in green.

// Use protocol factory to create a DefaultDataTransferSynchronizationProtocol object.

```
try {
      DistributedProtocolFactory factory = new
            DefaultDistributedProtocolFactory();
      DataTransferSynchronizationProtocol protocol =
            factory.createProtocol();
}catch(Exception e){
}
```

/* The above code shows that we create a protocol instance with default group (which is NetPeerGroup, default root group defined by JXTA), default fault tolerance ratio, which is 0.85, default 360 seconds service period, and default 60 seconds message life time.
*/

// Change to a group first, suppose the grp holds a reference to PeerGroup object, which we want
// to join. One thing we need to remember, all services are bound to each group individually, once
// we change our group, we are bound to new Data Transfer Service, that's why we can not send
// message to old group peers any more.

```
try {
      protocol.changeGroup(grp);
} catch (Exception e){
}
```

/* We can easily utilize discovery service provided by JXTA to find all peer groups, the following code shows how to do this. The codes below has nothing to do with this component.
*/

```
// Get the default group. everyone belongs to the NetPeerGroup
try {
      netPeerGroup = PeerGroupFactory.newNetPeerGroup();
} catch ( PeerGroupException e) {
}

// Get the discovery service from our peer group
discovery = netPeerGroup.getDiscoveryService();

// Add ourselves as a DiscoveryListener for Discovery events
discovery.addDiscoveryListener(this);

// Find peer groups, please refer to JXTA manual for details about
// the meaning of parameters. Basically, first null means return all
// peer groups, second parameter means we want to search for peer groups
// the last parameter 5 means we want 5 peer groups maximum in response
discovery.getRemoteAdvertisements(null,DiscoveryService.GROUP,
      null, null, 5);

// This callback function will be invoked if any peer responses.
public void discoveryEvent(DiscoveryEvent ev) {

      DiscoveryResponseMsg res = ev.getResponse();

      // now print out each discovered peer group
      PeerGroupAdvertisement adv = null;
      Enumeration en = res.getAdvertisements();
      if (en != null ) {
        while (en.hasMoreElements()) {
            adv = (PeerGroupAdvertisement) en.nextElement();
            System.out.println (" Peer Group = " + adv.getName());
        }
    }
}

// Adjust parameters
try{
    protocol.setMessageLifeTime(1000);
    protocol.setFaultToleranceRatio(0.9);
    protocol.setServicePeriod(600);
}catch(Excetion e){
}

// register ourself as a listener to the protocol.
protocol.addlistener(this);

// perform operations in the callback functions
public ApplicationOperationResult dataTransferSynchronizationEvent
    (DataTransferSynchronizationEvent event) {

    int code = event.getEventCode();

    // if a message from other peers need to be processed
    if (code ==  DataTransferSynchronizationEvent.COMMIT){
        // we perform necessary logic here
    }

    // if a message from other peers need to be aborted
    if (code ==  DataTransferSynchronizationEvent.ABORT){
        // we perform necessary logic here
    }

    // if the message we sent earlier has been received by all peers
    if (code ==  DataTransferSynchronizationEvent.OPERATION_SUCCESS){
        // we perform necessary logic here
    }

    // if the message we sent earlier failed to be delivered
    if (code ==  DataTransferSynchronizationEvent.OPERATION_FAIL){

        // we print out all error messages
        List errors = event.getErrors();
        for (int i = 0; i < errors.size(), i++){
            ErrorMessage msg = (ErrorMessage) errors.get(i);
            System.out.println(“Error Level:” + msg.getType() +
                “  Error Message” + msg.getMessage());
        }
    }
}
```

```
// Send messages at will. Suppose the data holds a string containing data to be sent.
protocol.sendSynchronizedData(data);

/* Note: It's application's responsibilities to define a format of
   data that all group peers agree upon
*/
```

4.4 Extra Documents

Design also provides several XML sample files to help developers understand the design. They are DataTransferQuery.xml and DataTransferResponse.xml. Each shows the XML format of the message of the corresponding class in com.topcoder.network.synchronization.message package.

5. Future Enhancements

In the future, send a message to multiple groups using different protocol could be added, and authentication when sending message between peers can be added.