DistRegistry

Reference Documentation

Version 0.95

1 Preface

The DistRegistry library is open source pure Java library, which is intended to support distributed registry of items (similar to distributed hash table, or DHT).

The main purpose of DistRegistry is to provide the open source, lightweight and Java-based implementation of distributed registry, which stores objects with given keys. Therefore, the main idea is quite close to using distributed hash table, but has some differences.

While DistRegistry can be used as independent library, the primary goals of its development is providing support for service discovering functionality implemented by Clustered Remoting For Spring Framework (Cluster4Spring) project which replaces built-in remoting functionality of Spring framework. It is intended to add dynamic service discovering, increase fault tolerance of the distribute system could be built with the use of Spring framework.

In addition to distributed registry, the library includes implementation of simple distributed cache as well as offers various utility classes used for various purposes, for example, in UDP based multicast communication.

This document highlights the most important issues related to DistRegistry. However, we suggest that you will also refer to JavaDoc for DistRegistry library as well as to the examples if you need more details.

2 License

3 Copyright Notice

The DistRegistry library was developed in SoftAMIS, a Ukraine based software development company specialized in Java and Web development outsourcing services.

To find more about SoftAMIS, about our services, skills and experience, please visit our site:

http://www.soft-amis.com

While several people were involved into development, debugging, testing and writing documentation for DistRegistry library, most of DistRegistry design and implementation was performed by Andrew Sazonov, the senior software architect in SoftAMIS.

Before first public release, the DistRegistry was used in several real enterprise level projects and now it is quite stable due to the long use.

4 Dependencies

DistRegistry relies heavily on Java 5 language features and therefore requires Java version 1.5 or later to be compiled and executed.

The logging functionality within DistRegistry is implemented using Apache Commons Logging (http://jakarta.apache.org/commons/logging/) therefore the *commons-logging.jar* should be available during library compilation as well as in Java classpath on runtime.

DistRegistry code base also includes reference to JGroups – a reliable multicast library (http://www.jgroups.org) as well as to Spring framework (http://www.springframework.org). However, there dependencies are only required for compiling library from source code. If during runtime no JGroups based network transport is used or if it is not intended that DistRegistry will not be used within part of Spring framework context. Accordingly, appropriate libraries are not required to run code which uses DistRegistry.

5 Major Features

Generally, DistRegistry does not represent full-fledge transactional cache solution and is not intended to compete with well-known and widely used open source and commercial solutions like JBoss Cache or Tangosol Coherence to name a few.

However, its functionality is enough to organize distributed communication and to support dynamic services discovering.

Briefly, the major features of DistRegistry library are:

- DistRegistry is Open Source implementation of distributed registry;
- It's possible to use library as base for development of more sophisticated functionality;
- Integration with Spring framework is included;
- Contains generic usage UDP multicasting functionality with can be used separately;
- Includes basic distributed cache;
- DistRegistry supports pluggable transport layer (either simple UDP multicast based or JGroups Based);
- Library is stable and ready to use;

6 Internal Architecture

In this section, we will go into detail of DistRegistry implementation.

From the high-level perspective, the major problem, which is tackled by DistRegistry library, is quite simple.

To simplify explanation, we will use sample with services discovering (at least, DistRegistry was developed to support dynamic services discovering); however, the overall approach is applicable to organization of distributed registry of any arbitrary objects which are tagged by their unique key. Internally, DistRegistry uses Java serialization to convert both keys and values to data which are transferred via network and therefore it is required that both objects used as keys and appropriate values should be serializable (and, obviously, has proper implementation of *equals()* and *hashCode()* methods).

Let‘s imagine that we have several servers that expose the set of services that could be available remotely.

Using DistRegistry, each server can publish information about its own services. Each service is denoted by some unique identifier. The identifier can be considered as a good candidate for a key under which particular service is published.

On the one hand, server may publish information about location of that service (or, more precisely, information that allows to access service). The ordinary RMI URL is a good example of such information.

On the other hand, it is not enough to publish information about services. The client interested in using remote service (in general terms, arbitrary data published by provider) should be able to discover location of that service.

To support such a scenario, the entire distributed registry consists of two parts – the first one let’s publishing information in registry and the second allows consuming published data from there.

In general, publishing part is considered to be “server-side” part of distributed registry while consuming part of registry is considered to be “client-side”. Of course, terms “client” and “server” are only applicable to internal organization of registry and are not related to overall architecture of application, because the last uses distributed registry (if necessary, the same component of distributed application may include both providing and consuming parts of the registry).

It should be noted that each publisher might store data in registry under the same key, while the data itself can vary and be specific for particular provider. As for the previous example – several identical servers may publish the same service under the same key, but it is obviously that location of that service will be specific for particular server.

Since the client is interested in obtaining information about all possible locations of the service, the architecture of distributed registry allows a client to retrieve all data items stored under the same key.

The following diagram illustrates the overall scheme of distributed registry organization.

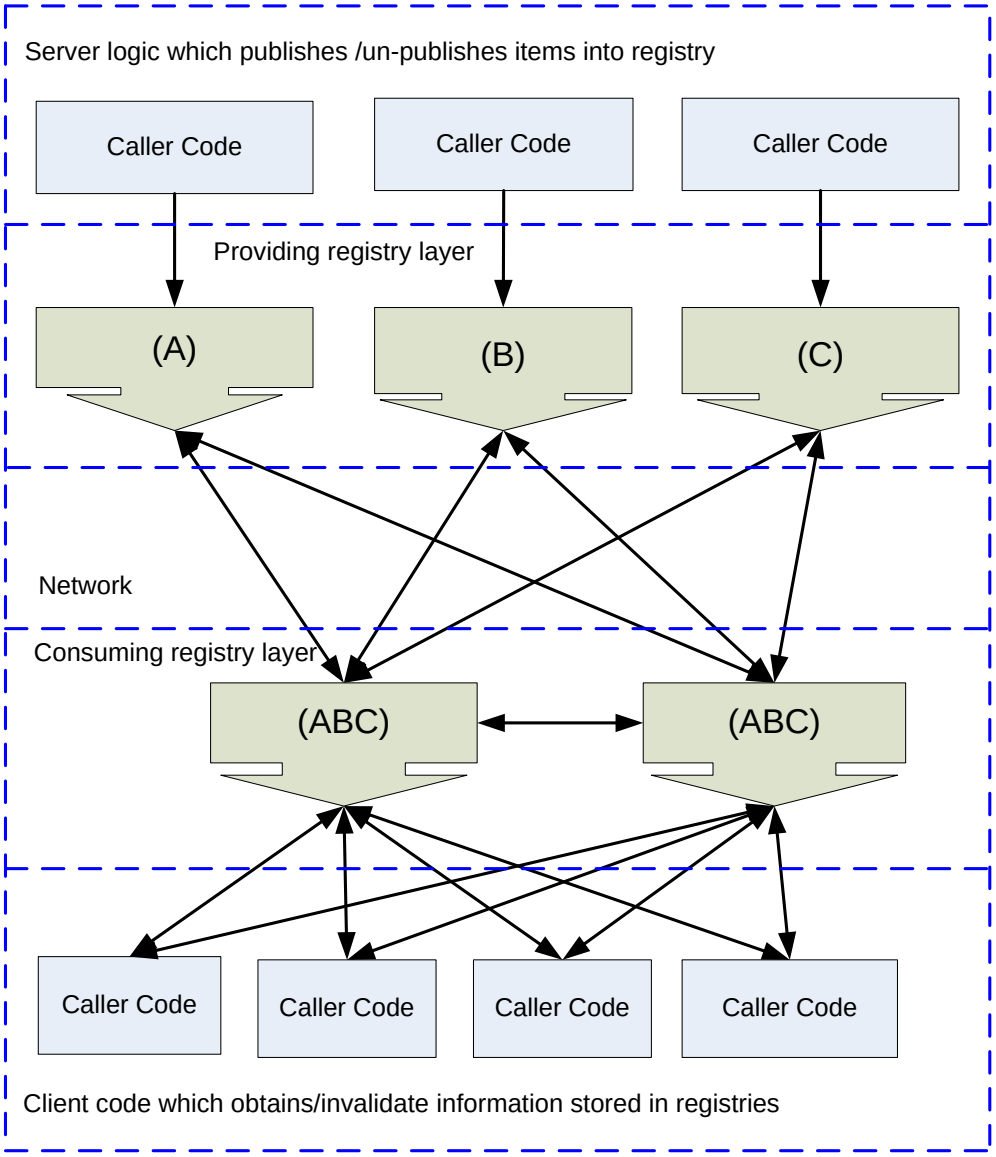

Figure 1 Overall scheme of distributed registry

From this diagram, it can be seen that there is a business logic code, which publishes information in *ProvidingRegistry*. Each providing registry is only "aware" of "own" items ("own" means items were published via it).

Data published by providing registries are propagated via network using fairly simple protocol. They are available on client side of *ConsumingRegistry*. Application code, which represents business logic, may ask *ConsumingRegistry* for available data.

Unlike *ProvidingRegistry*, the *ConsumingRegistry* contains data published by all providing registries (of course, this feature is defined by configuration of distributed registry).

ConsumingRegistry may ask *ProvidingRegistry* for all data published via them or for data published under particular key. In addition, particular *ConsumingRegistry* may "decide" that some data are no longer valid, and notify other *ConsumingRegistry* of that fact.

On the other hand, *ProvidingRegistry* may notify providing registry of data published under particular key or of all data published via particular *ProvidingRegistry* and also notify that some data published via it are not valid or available anymore and therefore should be removed from registry.

While previously we used examples with services publishing, the overall scheme of distributed registry allows working with arbitrary data if such a scheme should be supported inside the application.

7 Public API

This section of the document highlights public API of *ConsumingRegistry* and *ProvidingRegistry*. Usage of methods of these API is self-explanatory.

7.1 ProvidingRegistry

The public interface of *ProvidingRegistry* is quite simple and is illustrated by a listing below (please refer to JavaDoc comments to find descriptions of individual methods):

```
/**
 * Generic interface for ProvidingRegistry. The main purpose of provider
 * registry is to provide ability to register (or publish) items stored in
 * the distributed registry.
 * Appropriate ConsumerRegistries that shares data with this registry will
 * be notified of the items published by the ProvidingRegistry.
 *
 * @author Andrew Sazonov
 * @see ConsumingRegistry
 * @version 1.0
 * @param <K> type of keys used to identify registry items
 * @param <V> type of values stored in the registry
 */

public interface ProvidingRegistry<K extends Serializable,
                                   V extends Serializable>
{
 /**
  * Registers item stored under given key
  * @param aItemKey key of the item to be registered
  * @param aItem the item to be registered
  */
  public void registerItem(K aItemKey, V aItem);

 /**
  * Performs un-registration of the item under given key. Item will
  * be removed from the distributed registry
  * @param aItemKey item to unregister
  */
  public void unRegisterItem(K aItemKey);

 /**
  * Forces registry to issue notification for all registered items.
  * Potentially, this method can be called on some scheduled basis
  * to provider "heart-beat" of registry (so client registries will be
  * re-notified about content published via registry).
  */
  public void notifyAllRegistrations();
}
```

There is inherited interface, *DiscoverableProvidingRegistry* that is declared as it is illustrated by the listing below. It and allows obtaining information on items that were published by the particular instance of *ProvidingRegistry*.

The following listing illustrates *DiscoverableProvidingRegistry* interface.

```
/**
 * Extension to {@link ProvidingRegistry} that allows to obtain content
 * of items that was published by particular {@link ProvidingRegistry}.
 *
 * @author Andrew Sazonov
 * @version 1.0
 * @param <K> type of keys used to identify registry items
 * @param <V> type of values stored in the registry
 * @see ProvidingRegistry
 */
public interface DiscoverableProvidingRegistry<K extends Serializable,
                                               V extends Serializable>
                                    extends ProvidingRegistry<K, V>
{
/**
 * List of items that were published by this provider registry
 *
 * @return items
 */
 public List<Map.Entry<K, V>> getOwnItems();
}
```

7.2 ConsumingRegistry

The public interface of *ConsumingRegistry* contains more methods than one of *ProvidingRegistry* but still is quite simple.

```
/**
 * Generic interface for ConsumingRegistry. <code>ConsumingRegistry</code>
 * is used to obtain information about items stored in distributed registry.
 * Since there can be several ProvidingRegistries that publish different
 * items under the same key, the <code>ConsumingRegistry</code> allows to
 * obtain the set of items published by them under the particular key.
 *
 * In addition to obtaining information on current state of distributed
 * registry, the <code>ConsumingRegistry</code> may also indicate that some
 * items in the registry are invalid (therefore, if there are several
 * <code>ConsumingRegistries</code>, they may handle such a
 * notification).
 *
 * @author Andrew Sazonov
 * @see ProvidingRegistry
 * @version 1.0
 * @param <K> type of keys used to identify registry items
 * @param <V> type of values stored in the registry
 */
public interface ConsumingRegistry<K extends Serializable,
                                   V extends Serializable>
{
  /**
   * Indicates whether there were any changes in distributed registry occurred
   * due to synchronization since the last call of the <code>getItems()</code>
   * for given key.
   * @param aItemKey key for items
   * @return true if the set of data items stored under the given key was
   * changed since the last call of {@link #getItems}
   * @see #getItems
   */
  public boolean isDirty(K aItemKey);
```

```
/**
 * Returns the set of items stored in the distributed registry under given
 * key.
 * @param aItemKey key of items
 * @return set of items stored under given key
 */
public Set<V> getItems(K aItemKey);

/**
 * Returns list of available keys currently stored in registry (ones
 * which are actually stored in particular instance of consuming registry
 * or all published ones)
 * @return a set of available keys
 * @param aForceRefresh indicates whether request for available items
 * should be issued to distributed
 * registry before returning set of key
 */
public Set<K> getKeys(boolean aForceRefresh);


 /**
 * Provides ability to mark specific item  stored in the distributed
 * registry as invalid one.
 * It there are several <code>ConsumingRegistries<code> in the same
 * distributed registry, they can be notified of and be handled
 * properly. It's assumed that this call will not affect
 * ProvidingRegistries somehow.
 * @param aItemKey key of the item
 * @param aItem the item which should be marked invalid
 */
public void markItemInvalid(K aItemKey, V aItem);

/**
 * Initiates requesting of items for given key by sending appropriate
 * notifications to ProvidingRegistries
 * @param aItemKey if null, will request items for all available keys
 */
public void requestItems(K aItemKey);

/**
 * Removes listener from list of registered RegistryEventProcessors which
 * performs further processing of obtained external notifications
 * @param aListener processor to be removed
 */
public void removeRegistryEventProcessor(RegistryEventProcessor<K, V>
                                         aListener);

/**
 * Adds listener to list of registered RegistryEventProcessors which
 * performs the  further processing of obtained external notifications
 * @param aListener processor to be added
 */
public void addRegistryEventProcessor(RegistryEventProcessor<K, V>
                                      aListener);

/**
 * Closes this registry instance and performs all necessary cleanup. As
 * soon as registry is  closed, no actions may be performed within  it, so
 * calling this methods actually ends registry instance lifecycle.
 */
public void close();
```

```
  /**
   * Indicated whether registry it in closed state
   * @return <code>true</code> if registry instance is closed, false otherwise
   * @see #close()
   */
  public boolean isClosed();
}
```

DistRegistry distribution includes several examples that illustrate various aspects of distributed registry configuration and usage. Please refer to these examples to find more about possible use cases for distributed registry.

8 Implementation Details

This section of the document contains information on details of distributed registry implementation in DistRegistry library. These details are given there to provide better exposure of distributed registry internals, which can be useful for better understanding of distributed registry configuration details and customizations (if ones will be necessary).

8.1 Internal decomposition

In previous section, we have considered both *ProvidingRegistry* and *ConsumingRegistry* as solid notions. Of course, this is correct from the high-level view on architecture, but internally different classes compose both these entities.

Both *ProvidingRegistry* and *ConsumingRegistry* aggregate these classes and delegate significant part of their functionality to these classes. Since both registries are just different parts of the same distributed registry, they have quite close structure and internally utilize almost the same set of components.

Let us consider these classes in more details.

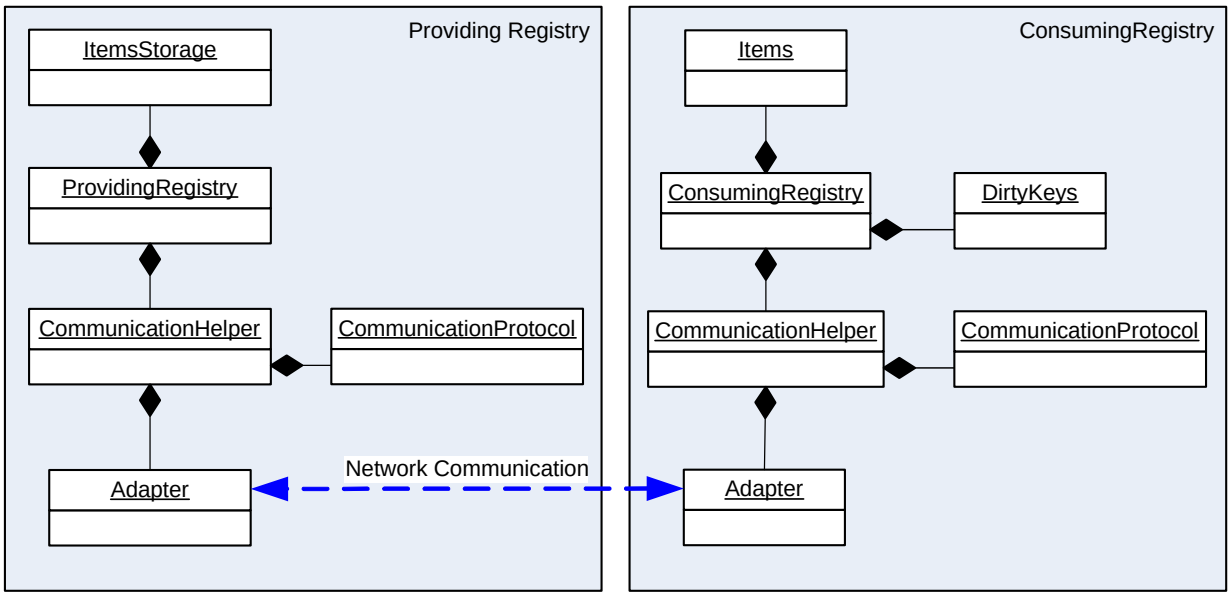

Figure 2 Overall scheme of distributed registry

First, both registries use some storage for items. *ProvidingRegistry* uses that storage to save data published directly via it only, while storage in *ConsumingRegistry* contains data item for particular key(s) published by all instances of *ProvidingRegistry*, which communicates with *ConsumingRegistry*.

The classes that implements registries include just a high-level functionality related to management of storage and communication with caller code. All details of network communication and commands, which are used within that protocol, are isolated from registries and are delegated to *CommunicationHelper*.

CommunicationHelper represents a facade for network communication protocol used by distributed registry. The protocol is fairly simple and sending of appropriate network messages is invocated by corresponding methods of the *CommunicationHelper*. In addition to issuing commands to network, *CommunicationHelper* also obtains information about incoming

messages and provides listener that allows interested parties (such as registry) get information about incoming data.

To encode and decode data from high-level registry specific format to raw data format, in which data will be transferred via network, the *CommunicationHelper* uses *CommunicationProtocol* class. That class performs encoding of data, which should be sent via network, and decoding data received from there.

And finally, the *Adapter* is used to perform low-level network communication by sending and receiving network message.

8.1.1 Pluggable networks transport

The current implementation of distributed registry allows isolating details of network communication from logic of the registry via isolated *CommunicationHelper*. Currently, there are two possible implementations of network communication available – one is based on simple UDP multicasting and another one which uses JGroups library for network communication. In general, it is possible to implement different schemes of network transport.

Depending on scheme of networking communication scheme, appropriate *CommunicationProtocol* and *CommunicationHelper* should be used by registries.

Of course, if one type of network transfer layer is used by *ProvidingRegistry* (say, JGroups based one) the same type of network transfer layer should be used by *ConsumingRegistry*.

8.1.1.1 Simple UDP

The UDP based network transport protocol specific classes are located in org.softamis.net.exchange.udp package.

The UDP multicasting (*Adapter* component) is supported by classes located in org.softamis.net.multicast package. Please note that these classes are, in general, independent from the distributed registry and can be used separately.

Internally, the following figure illustrates the format of packets sent by multicaster:

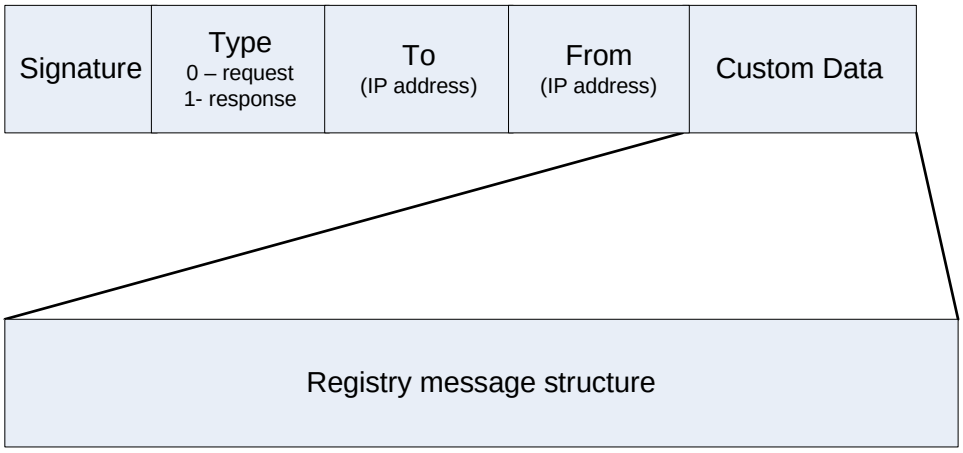

Figure 3 Structure of UDP message

As it illustrated by figure above, the structure of UPD message sent by multicaster is simple and represents rather an envelope, which carries some custom data. For distributed registry, these custom data represent specific message, which is part of internal registry communication protocol.

The *Signature* field is used to filter incoming datagram packages and selecting only ones specific messages (this is necessary if the same multicast group is used by different applications or by different components of the application). Other components of message are self-explanatory.

8.1.1.2 JGroups

The UDP based network transport protocol specific classes are located in org.softamis.net.exchange.jgroups package.

The *PullPushAdapter* from JGroups is used to implement underlying network operations.

8.1.2 Details of communication protocol

The communication protocol used within distributed registry is very simple. Actually, there are only four commands used internally. These commands are listed in the table below.

Command	Purpose
COMMAND_ITEM_REGISTERED	Notification about item registration. It is sent by *ProvidingRegistry* either as result of new item publishing or after processing COMMAND_ITEM_REQUEST command issued by *ConsumingRegistry*
COMMAND_ITEM_INVALID	Issued either by *ConsumingRegistry* to invalidate particular item data. Processed by other *ConsumingRegistry* instances.
COMMAND_ITEM_UNREGISTERED	Issued by *ProvidingRegistry* if data item is un-published. Processed by *ConsumingRegistry* instances, which removes data item from their storages.
COMMAND_ITEM_REQUEST	Issued by *ConsumingRegistry* to obtain information about data item under particular key or for all data items published. Processed by all instances of *ProvidingRegistry*.

The *CommunicationHelper* component offers high level API, which allows issuing such a network commands as well as processing them.

These commands are carried out by appropriate communication messages used by distributed registry. These messages are part (or content) of messages issued by either UDP multicaster or JGroups based messages.

The structure of communication message is also very simple and figure below illustrates it:

Signature (registry signature)	Type 0 – item registered 1-item invalid 2 – item unregistered 3 – item request	To (registry ID)	From (registry ID)	Key (item key)	Value (item value)

Figure 4 Structure of distribute registry message

The *Signature* field is used to filter messages, which are related to particular distributed registry. This is necessary if several distributed registries are used within the same application or if there are other components of the system, which use the same network protocol but transfer different type of data.

Type defines the command for message.

To and *From* fields are used internally by registry and present ID of providing or consuming registry.

The Key field represents key of data item, Value field contains that data item itself. If COMMAND_ITEM_REQUEST command is issued by *ConsumingRegistry*, that field is empty. If Key item is missed in COMMAND_ITEM_REQUEST command, such a message is considered as request for all items published.

8.1.3 Details of internal functioning

Here we briefly review some details of internal implementation of distributed registry and collaboration between internal components. We consider high-level sequence of processing incoming data from network and issuing network notifications.

To find more details of communications, please refer to source code of DistRegistry library.

8.1.3.1 Processing incoming data

Each registry within its lifecycle either issues data into network or receive them. Here we consider what happens if outer data are available and examine the sequence of processing them.

The following diagram illustrates the process:

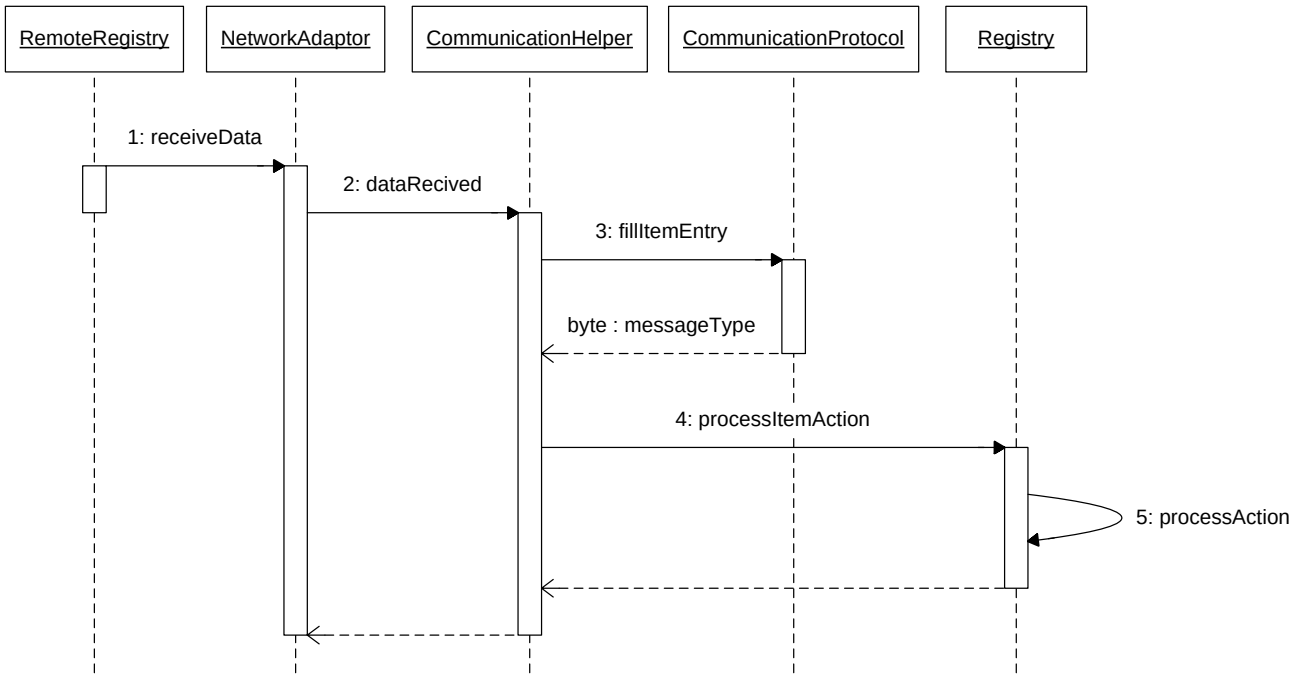

Figure 5 Sequence of incoming data processing

When some remote component of distributed registry issues network message, it first arrives to network adapter (UDP or JGroups based). If *NetworkAdapter* decides that it should process incoming data, it notifies *CommunicationHelper* of that fact (step 2). *CommunicationHelper* decodes incoming message using *CommunicationProtocol* (step 3), checks whether valid data are received and whether this message is related to distributed registry. If necessary, it notifies registry of obtained data (step 4). Registry performs all necessary operations according to incoming data.

8.1.3.2 Issuing outgoing notification

The process of issuing notification to network is quite close to the previous case and utilizes the same set of component. Actually, the sequence is close but is directed to the opposite side. The following diagram illustrates it:

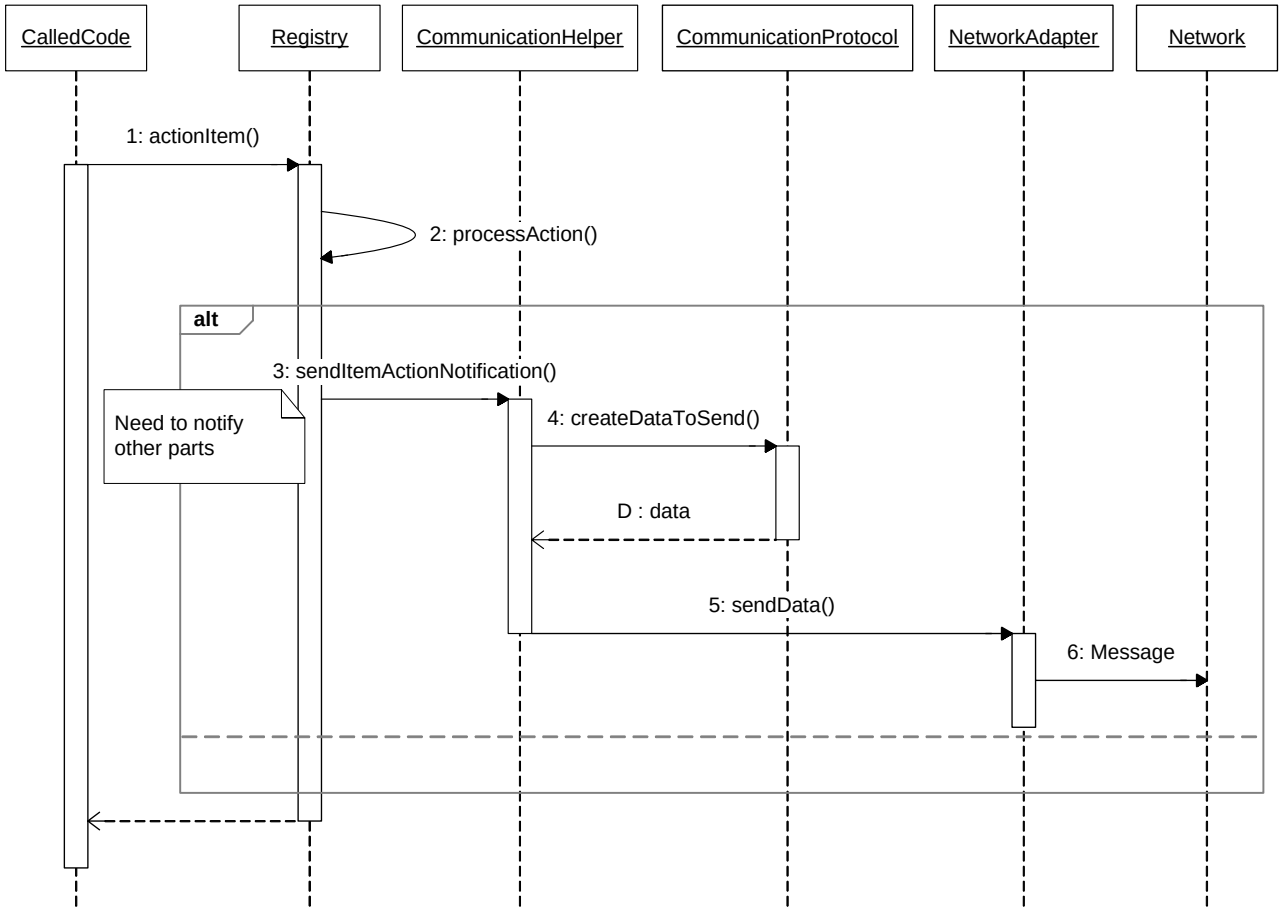

Figure 6 Sequence of incoming data processing

As it was expected, the sequence is quite straightforward. As soon as caller code (one which outside *ConsumingRegistry* or *ProvidingRegistry*) invokes some method, registry starts to process it. If registry determines that network notification is to be issued, it issues proper call to *CommunicationHelper* (step 3). *CommunicationHelper* creates appropriate registry message and uses CommunicationProtocol to encode data to format, which is ready to network (step 4).

After that, *CommunicationHelper* invokes *NetworkAdapter* to initiate sending of data via network.

8.2 Distributed cache

In some cases, the providing registry can be not applicable for needs of particular application. For example, it can be necessary to have a simple *HashMap*-like structure that stores only one value under given key but is still distributed.

For this purposes, DistRegistry includes simple implementation of distributed hash table, which is based on the same protocol as one used by distributed registry, but stores only one value under one key.

Current implementation does not make any assumptions about resolving possible conflicts (i.e. how to resolve the case if different data for the same key are stored in different locations). Instead, it simply stores data and therefore slightly different values can be stored in different instances of cache.

However, such a simple distributed cache is still quite useful in some cases despite of these limitations.

To find more about distributed cache, please refer to *org.softamis.net.cache* package.

8.3 Integration with Spring framework

In general, the distributed registry does not require Spring framework and can be used separately.

However, Spring is very popular today and offers very convenient way of assembling applications via Inversion of Control (IoC) pattern (among with other features). That is why DistRegistry includes some small extensions of base classes which are intended to simplify usage of library components within Spring container (however, it's not necessary to use extended class to use DistRegistry within Spring – extensions just slightly simplify configuration).

These extended classes do not provide any additional logic, which is directly related to, for example, distributed registry. However, they contain implementation of some interfaces related to Spring lifecycle.

To find more about such classes, please refer to *org.softamis.net.registry.spring* package.

9 Configuration Examples

This section contains several examples of configuring distributed registry. These examples assumed that Spring framework is used as IoC container and therefore represent appropriate fragments of Spring configuration files.

However, it is possible to use DistRegistry out of Spring (using either different IoC container or simply assembling all components in plain Java code). Please refer to examples included to DistRegistry distribution to find examples of configuration of distributing registry in Java.

9.1 UDP based registry

The following examples illustrate how to configure distributed registry if UDP multicasting is used as underlying networking protocol.

9.1.1 Configuring ProvidingRegistry

Configuration of providing registry is quite simple. The most configuration options there are related not to *ProvidingRegistry* itself, but rather to details of networking communications.

9.1.1.1 Verbose configuration

The following listing illustrates the most complete configuration of *ProvidingRegistry* and illustrates all properties that may be configured. Such a form of configuration is useful if, for example, the same UDP multicaster is shared between different components application.

Please refer to configuration comments below to find more information for every particular element of declaration.

```
[1]<bean name="_ProvidingRegistry.verbose.udp"
        class="org.softamis.net.registry.spring.DefaultProvidingRegistry">
[2]   <property name="communicationHelper">
[3]      <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
              init-method="init" destroy-method="close">
[4]        <property name="multicaster" ref="UDPMulticaster"/>
[5]        <property name="communicationProtocol">
[6]          <bean class="org.softamis.net.exchange.udp.UDPCommunicationProtocol">
[7]            <property name="datagramSignature" value="NIRM"/>
[8]            <property name="useCompression" value="false"/>
             </bean>
           </property>
       </bean>
    </property>
  </bean>

[9]<bean id="UDPMulticaster"
        class="org.softamis.net.multicast.DefaultMulticaster">
[10] <property name="groupName" value="230.0.0.1"/>
[11] <property name="port" value="12890"/>
[12] <property name="timeToLive" value="2"/>
[13] <property name="active" value="true"/>
  </bean>
```

Configuration details:

1. Declaration of *ProvidingRegistry* bean;

2. Here we declare *CommunicationHelper* which performs necessary network communications;
3. Since we use UDP based transport layer in this example, we specify corresponding class of helper. Also, it's important that *init()* and *destroy()* methods on communication helper will be called for initialization and closing;
4. Property which specifies instance of UDP multicaster which will be used by helper to perform network communications;
5. Property allows to declare *CommunicationProtocol* used by helper to encode/decode network data;
6. Declaration of UDP based *CommunicationProtocol*;
7. Property which specifies signature of messages will be used by distributed registry (default is “NIRM”);
8. Optional property which allows to specify whether data should be compressed before sending to network;
9. Configuration of UDP multicaster which performs issuing/receiving UDP notifications;
10. Property that specifies UDP multicast group (multicast IP address) which is used by multicaster (default is 230.0.0.1);
11. Property that specifies number of port should be used for multicasting (default value is 12890);
12. Time to live (TTL) parameter which specifies how many network nodes may pass UDP datagram (default value is 2);
13. Property that allows specifying the fact that multicaster should be in active state immediately after startup.

9.1.1.2 Short configuration

The following listing illustrates shorter configuration of *ProvidingRegistry* that requires shorter markup and creates appropriate instances of UDP multicaster and *CommunicationProtocol* internally in *UDPCommunicationHelper*.

Please refer to configuration comments below to find more information for every particular element of declaration.

```
[1]<bean name="_ProvidingRegistry.short.udp"
        class="org.softamis.net.registry.spring.DefaultProvidingRegistry">
[2]  <property name="communicationHelper">
[3]    <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
           init-method="init" destroy-method="close">
[4]      <property name="defaultGroupName" value="230.0.0.1"/>
[5]      <property name="defaultPort" value="10000"/>
[6]      <property name="defaultTimeToLive" value="20"/>
[7]      <property name="defaultMessageSignature" value="NIRM"/>
       </bean>
    </property>
  </bean>
```

Configuration details:

1. Declaration of *ProvidingRegistry* bean;

2. Here we declare *CommunicationHelper* which performs necessary network communications;
3. Since we use UDP based transport layer in this example, we specify corresponding class of helper. Also, it's important that *init()* and *destroy()* methods on communication helper will be called for initialization and closing;
4. Property that specifies UDP multicast group (multicast IP address) which is used by multicaster (default is 230.0.0.1);
5. Property that specifies number of port should be used for multicasting (default value is 12890);
6. Time to live (TTL) parameter which specifies how many network nodes may pass UDP datagram (default value is 2);
7. Property which specifies signature of messages will be used by distributed registry (default is “NIRM”);

9.1.1.3 Minimal configuration

The following listing illustrates minimal required configuration of *ProvidingRegistry* and illustrates all properties that may be configured. Most of properties have internal built-in default values.

```
<bean name="_ProvidingRegistry.minimal.udp"
      class="org.softamis.net.registry.spring.DefaultProvidingRegistry">
   <property name="communicationHelper">
      <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
            init-method="init" destroy-method="close"/>
   </property>
</bean>
```

9.1.2 Configuring ConsumingRegistry

As soon as *ProvidingRegistry* is configured, it is necessary to configure *ConsumingRegistry*. Exactly as for *ProvidingRegistry*, there are, generally, three possible examples of configuration – verbose, short and minimal (built-in defaults based) ones that are illustrated by listings below.

In general, since internally *ConsumingRegistry* contains the same components, its configuration is very close to configuration of *ProvidingRegistry*.

Please note that it is important to have the same configuration of network related options (like multicast group, multicast port) as well as message signature used by distributed registry both on *ProvidingRegistry* and *ConsumingRegistry* configuration to make sure that distributed registry works properly (in other case, *ConsumingRegistry* simply will not receive notifications from *ProvidingRegistry*).

9.1.2.1 Verbose configuration

The following listing illustrates the most complete configuration of *ConsumingRegistry* and illustrates all properties that may be configured. Such a form of configuration is useful if, for example, the same UDP multicaster is shared among different components of application.

Please refer to configuration comments below to find more information for every particular element of declaration.

```
[1] <bean name="_ConsumingRegistry.verbose.udp"
         class="org.softamis.net.registry.spring.DefaultConsumingRegistry">
[2] <property name="communicationHelper">
      <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
            init-method="init" destroy-method="close">
        <property name="communicationProtocol">
          <bean
            class="org.softamis.net.exchange.udp.UDPCommunicationProtocol">
            <property name="datagramSignature" value="NIRM"/>
            <property name="useCompression" value="false"/>
          </bean>
        </property>
        <property name="multicaster" ref="UDPMulticaster"/>
      </bean>
    </property>
[3] <property name="discoveringTimeout" value="1000"/>
[4] <property name="requestItemsOnInit" value="true"/>
[5] <property name="issueRequestForInvalidatedService" value="false"/>
   </bean>

[6]<bean id="UDPMulticaster"
         class="org.softamis.net.multicast.DefaultMulticaster">
    <property name="groupName" value="230.0.0.1"/>
    <property name="port" value="12890"/>
    <property name="timeToLive" value="2"/>
    <property name="active" value="true"/>
   </bean>
```

Configuration details:

8. Declaration of *ConsumingRegistry* bean;
9. Here we declare *CommunicationHelper* which performs necessary network communications – all configuration is the same as was illustrated before for *ProvidingRegistry*;
10. Property that specifies timeout used to wait for responses from providing registries when request for items is issued (in milliseconds). Default is 1000;
11. Property that controls whether registry should request for available items as soon as it configured (default value is true);
12. Property that controls how *ConsumingRegistry* should process invalidation notifications from other registries (default value is false). It defines policy how COMMAND_ITEM_INVALIDATED notification should be processed by *ConsumingRegistry*. If this option is set to true, the *ConsumingRegistry* will invalidate item and then issue COMMAND_ITEM_REQUEST for invalidated item key. If this option is set to false, the *ConsumingRegistry* will simply invalidate item.
13. Configuration of UDP multicaster that performs issuing/receiving UDP notifications;

9.1.2.2 Short configuration

The following listing illustrates shorter configuration of *ConsumingRegistry* that requires shorter markup and creates appropriate instances of UDP multicaster and *CommunicationProtocol* internally in *UDPCommunicationHelper*.

Please refer to configuration comments below to find more information for every particular element of declaration.

```
[1] <bean name="_ConsumingRegistry.short.udp"
          class="org.softamis.net.registry.spring.DefaultConsumingRegistry">
[2]    <property name="communicationHelper">
[3]      <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
               init-method="init" destroy-method="close">
[4]        <property name="defaultGroupName" value="230.0.0.1"/>
[5]        <property name="defaultPort" value="10000"/>
[6]        <property name="defaultTimeToLive" value="20"/>
[7]        <property name="defaultMessageSignature" value="NIRM"/>
         </bean>
       </property>
   </bean>
```

Configuration details:

1. Declaration of *ConsumingRegistry* bean;
2. Here we declare *CommunicationHelper* which performs necessary network communications;
3. Since we use UDP based transport layer in this example, we specify corresponding class of helper. Also, it's important that *init()* and *destroy()* methods on communication helper will be called for initialization and closing;
4. Property that specifies UDP multicast group (multicast IP address) which is used by multicaster (default is 230.0.0.1);
5. Property that specifies number of port should be used for multicasting (default value is 12890);
6. Time to live (TTL) parameter which specifies how many network nodes may pass UDP datagram (default value is 2);
7. Property which specifies signature of messages will be used by distributed registry (default is “NIRM”);

9.1.2.3 Minimal configuration

The following listing illustrates minimal required configuration of *ConsumingRegistry* and illustrates all properties that may be configured. Most of properties have internal built-in default values.

```
  <bean name="_ConsumingRegistry.minimal.udp"
        class="org.softamis.net.registry.spring.DefaultConsumingRegistry">
    <property name="communicationHelper">
      <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
            init-method="init" destroy-method="close"/>
    </property>
  </bean>
```

9.2 JGroups based registry

The following examples illustrate how to configure distributed registry if JGroups based multicasting is used as underlying networking protocol. In general, the overall way of configuration is very close to UDP based one, so in the following listings we highlight rather differences that are specific for JGroups based setup.

9.2.1 Configuring ProvidingRegistry

Configuration of providing registry is quite simple. The most configuration options there are related not to ProvidingRegistry itself, but rather to details of networking communications.

9.2.1.1 Verbose configuration

The following listing illustrates the most complete configuration of *ProvidingRegistry* and illustrates all properties that may be configured.

Please refer to configuration comments below to find more information for every particular element of declaration.

```
[1]<bean name="_ProvidingRegistry.verbose.jgroups"
         class="org.softamis.net.registry.spring.DefaultProvidingRegistry">
[2]   <property name="communicationHelper">
[3]      <bean class="org.softamis.net.exchange.jgroups.JGCommunicationHelper"
              init-method="init" destroy-method="close">
[4]        <property name="signature" ref="NIRG"/>
[5]        <property name="defaultGroupName" value="sa.registry"/>
[6]        <property name="defaultProperties">
             <value>"UDP(mcast_addr=228.5.5.5;mcast_port=45566;
                   ip_ttl=4;mcast_send_buf_size=150000;
                   mcast_recv_buf_size=80000)
                   :PING(timeout=2000;num_initial_members=3)
                   :MERGE2(min_interval=5000;max_interval=10000)
                   :FD_SOCK:VERIFY_SUSPECT(timeout=1500)
                   :pbcast.NAKACK(gc_lag=50;retransmit_timeout=300,600,
                   1200,2400,4800)
                   :UNICAST(timeout=600,1200,2400)
                   :pbcast.STABLE(desired_avg_gossip=20000)
                   :FRAG(frag_size=4096;down_thread=false;up_thread=false)
                   :pbcast.GMS(join_timeout=5000;
                               join_retry_timeout=2000;
                               shun=false;print_local_addr=true)"
           </value>
         </property>
[7]      <property name="communicationProtocol">
           <bean class="org.softamis.net.exchange.jgroups.JGCommunicationProtocol">
[8]          <property name="datagramSignature" value="NIRG"/>
             <property name="useCompression" value="false"/>
           </bean>
         </property>
       </bean>
    </property>
  </bean>
```

Configuration details:

1. Declaration of *ProvidingRegistry* bean;
2. Here we declare *CommunicationHelper* which performs necessary network communications;
3. Since we use JGroups there, we specify corresponding class of helper. Also, it's important that *init()* and *destroy()* methods on communication helper will be called for initialization and closing;
4. Property that specifies signature of message which will be used by adapter to select JGroups messages (default value is “NIRG”);

5. Property that specifies name of JGroups group used by communication channel;
6. Property that represents configuration spring for JGroups channel;
7. Configuration of JGroups specific *CommunicationProtocol*;
8. Signature of messages which will be used by distributed registry (default is NIRG) to select own network messages;

9.2.1.2 Minimal configuration

The following listing illustrates minimal required configuration of *ProvidingRegistry* and illustrates all properties that may be configured. Most of properties have internal built-in default values.

```
  <bean name="_ProvidingRegistry.minimal.jgroups"
        class="org.softamis.net.registry.spring.DefaultProvidingRegistry">
    <property name="communicationHelper">
      <bean class="org.softamis.net.exchange.jgroups.JGCommunicationHelper"
            init-method="init" destroy-method="close"/>
    </property>
  </bean>
```

9.2.2 Configuring ConsumingRegistry

Following examples illustrates how to configure *ConsumingRegistry* if JGroups bases networking protocol is used and underlying network transport.

Since all principles of configuration are the same as for UDP based communication and to configuring *ProvidingRegistry*, here we illustrate only minimal required configuration.

```
  <bean name="_ConsumingRegistry.minimal.jgroups"
        class="org.softamis.net.registry.spring.DefaultConsumingRegistry">
    <property name="communicationHelper">
      <bean class="org.softamis.net.exchange.jgroups.JGCommunicationHelper"
            init-method="init" destroy-method="close"/>
    </property>
  </bean>
```

9.3 Configuring distributed cache

The following example illustrates how to configure simple distributed cache. Generally, that cache uses the same internal components and therefore the process of configuring it is very close to configuring registry. Therefore, for simplicity, here we demonstrate only configuration that utilized UDP and relies on default values of properties.

Please note that configuration of all instances of distributed cache that should share the same data should be also the same to insure that these instances may synchronize their data.

Again, configuration of cache is very simple and is shown on the listing below. Please refer to configuration comments below to find more information for every particular element of declaration.

```
[1]<bean id="_SampleCache"
         class="org.softamis.net.registry.spring.DefaultDistributedCache"
         destroy-method="clear">
[2]   <property name="communicationHelper">
        <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
              init-method="init" destroy-method="close">
```

```
[3]       <property name="defaultMessageSignature" value="EX.CACHE"/>
        </bean>
      </property>
  </bean>
```

Configuration details:

9. Declaration of *DistributedCache* bean;
10. Here we declare appropriate *CommunicationHelper* (in this example, UDP based one is shown) which performs necessary network communications;
11. Property that specifies signature of messages should be processed by *CommunicationHelper*. Here we use different signature to eliminate possible clash with network messages that are issued by distributed registry. The signature of message should be specified only if it is assumed that both distributed registry and cache has similar network setup (for example, use that same multicaster, or use multicaster with the same network settings;

10 Where to find more information

To find more information about DistRegistry and to obtain newer versions of DistRegistry, please visit either SoftAMIS site

http://www.soft-amis.org/distregistry/index.html

Or visit project page on SourceForge:

http://sourceforge.net/projects/pscs

Also, if you have questions or comments regarding DistRegistry, please feel to contact us via email:

cluster4spring@soft-amis.org

Please also refer to JavaDoc, examples, and source code for DistRegistry – we hope that you will find helpful details there.

Oh, and do not forget that your feedback, ideas, suggestions and usage experience is very important for us! Please do not hesitate dropping us email if you have any questions or simply would like to share your impression with us!