HyCube library documentation

v.1.0.2

Author: Artur Olszak

12.10.2016

Contents

1. HyCube Java library

This chapter describes the *HyCube* library - Java implementation. The library (a jar file) consists of core classes and interfaces (generic - allowing creating different implementations of a DHT), and *HyCube*-specific classes, implementing the algorithms used by a *HyCube* node. The jar file contains the compatibility information (Java version) and a default configuration file (*hycube-default.cfg*). Individual configuration properties may be overwritten in the application-specific configuration files. The chapter presents the API, the architecture of the library and describes configuration of individual modules. The following sections are supposed to provide an overview and explain the main concepts being used. Details of individual classes/interfaces are provided in the *Javadoc* documentation attached to the library.

1.1. The API

The library API is provided by several node service classes:

- *HyCubeSingleQueueNodeService*, *HyCubeSingleQueueNodeServiceNonWakeable*
- *HyCubeMultiQueueNodeService*, *HyCubeMultiQueueNodeServiceNonWakeable*
- *HyCubeSchedulingMultiQueueNodeService*
- *HyCubeSimpleNodeService*, *HyCubeSimpleSchedulingNodeService*

Node services should be considered as the main API entry point, unless certain customizations are required, in which case other publicly accessible classes may be used. All node services implement the *HyCubeNodeService* interface (defining operations performed in the node context). Individual node services realize various approaches for event processing and managing threads. Because individual node services manage threads differently, they expect different configuration parameters - specified in the configuration file, as well as parameters passed to the node service initializer method call (*initialize*), and they may define additional service-specific methods (in addition to the *HyCubeNodeService* interface methods).

This section focuses on the *HyCubeNodeService* interface and the *HyCubeSimpleNodeService* node service implementation, which may be used for majority of applications. Other node services are described in detail in Section 1.5. *HyCubeSimpleNodeService* is a node service automatically determining the number of threads needed for processing events, based on the parameters specified upon the node service instance creation (*initialize* method) - described in Table 1.1.

Method	Description
Environment **environment**	The environment object represents the external environment, defines the time provider and contains node configuration read from the configuration file (details in Sections 1.3.2, 1.3.3 and 1.3.4). In the simplest case, the *DirectEnvironment* class instance (providing the system time time provider and scheduler) and the default parameter file may be used (used by default when the configuration file is not specified in the *DirectEnvironment.initialize* method).
String **nodeIdString /** ***NodeId*** **nodeId**	The node ID - a *String* representation or a *NodeID* object (*HyCubeNodeId* instance is expected with the default configuration).
String **bootstrapNodeAddress**	The bootstrap node network address. With the default configuration, the UDP/IP protocol is used, and the address format is: *IP_ADDRESS:PORT*. If the *null* value is provided, the node does not perform the JOIN procedure and forms a DHT containing only itself (other nodes may connect to it).
JoinCallback **joinCallback**	The callback object that is notified (by calling the *joinReturned* method) when the JOIN operation terminates (the node joined the DHT). In most cases, the use of an instance of *JoinWaitCallback* (providing blocking waiting for the JOIN to finish) is sufficient.
Object **callbackArg**	An argument that will be passed to the *JoinCallback.joinReturned* method.
int **blockingExtEventsNum**	The number of custom blocking events processed by nodes (details in Section 1.5). For the default configuration, value 0 should specified (this the default value used in case the value is not specified).
boolean **wakeup**	A flag determining whether blocking events should be interrupted when non-blocking operations are enqueued to be processed. The mechanism is described in detail in Section 1.4. The value *true* may be specified by default, in which case the node may be served by a single thread. Otherwise, at least two threads should be defined, as one of the threads would be blocked for most of the time by blocking waiting for incoming messages.
EventProcessingErrorCallback **errorCallback**	Specifies the error callback object that will be notified (*errorOccurred* method called) when a critical internal error occurs in a thread different than the API caller thread. When such an error is raised, the further processing should be terminated and the node instance should be discarded.
Object **errorCallbackArg**	An object passed to the *EventProcessingErrorCallback.errorOccurred* method when an error is raised.

Table 1.1 *HyCubeSimpleNodeService.initialize* method parameters

A node service represents a DHT node (connected to the DHT system), and the operations defined by *HyCubeNodeService* interface may be performed in the node context (Table 1.2).

Method	Description
Node **getNode**	Returns the *Node* class instance, representing the node
NetworkNodePointer **createNetworkNodePointer**	Creates a network node pointer object from its string representation
void **setPublicAddress**	Sets the node's public network address (in case the network address translation is used) - this address will be exposed to other nodes in messages
MessageSendInfo **send**	Sends a message to the specified recipient from the local source port to the recipient's destination port. The method expects the following arguments: the message recipient ID (*NodeId*), optional direct recipient (*String* or *NetworkNodePointer* representation of the network address), the message data (*byte[]*), the ack callback object (*AckCallback*) notified when the delivery confirmation is received, optional routing parameters (*Object[]* described in Section 1.3.13, and a boolean flag indicating whether the call should be blocking or sending the message should be enqueued and performed in the background. The recipient ID, recipient network address, source port, destination port and message data may be aggregated in a *DataMessage* class instance.
LookupCallback **lookup**	Initiates the node lookup procedure, for the specified node ID. The method arguments include the lookup node ID (*NodeId*), the lookup callback object (*LookupCallback*) notified when the lookup procedure terminates, passing the result of the operation to the *lookupReturned* method, a callback argument (*Object*) that will be passed to the *lookupReturned* method call, and optional lookup parameters (*Object[]* described in Section 1.3.14.
SearchCallback **search**	Initiates the search procedure, for the given number of closest nodes to the specified node ID. The method arguments include the search node ID (*NodeId*), the search callback object (*SearchCallback*) notified when the search procedure terminates, passing the result of the operation to the *searchReturned* method, a callback argument (*Object*) that will be passed to the *searchReturned* method call, optional set of initial node pointers (*NodePointer[]*) to which initial search requests should be sent, an optional flag indicating whether the exact match node should NOT be returned by any intermediate node (default value *false*), and optional search parameters (*Object[]* described in Section 1.3.15.
PutCallback **put**	Initiates the PUT operation, storing a resource in the DHT. The method arguments include the message recipient (*NodePointer*) - the argument is optional (if not specified, the message will be routed), the resource key (*BigInteger*), the resource object (*HyCubeResource*), a callback object (*PutCallback*) notified when the PUT operation is finished, passing the status of the operation to the *putReturned* method call, and optional put parameters (*Object[]* described in Section 1.3.18.

***RefreshPutCallback* refreshPut**	Initiates the REFRESH_PUT operation, refreshing the validity time of the resource in the DHT. The method arguments include the message recipient (*NodePointer*) - the argument is optional (if not specified, the message will be routed), the resource key (*BigInteger*), the resource descriptor (*HyCubeResourceDescriptor*), a callback object (*RefreshPutCallback*) notified when the REFRESH_PUT operation is finished, passing the status of the operation to the *refreshPutReturned* method call, and optional parameters (*Object[]*) described in Section 1.3.18.
***GetCallback* get**	Initiates the GET operation, retrieving resources from the DHT. The method arguments include the message recipient (*NodePointer*) - the argument is optional (if not specified, the message will be routed), the resource key (*BigInteger*), the get criteria (*HyCubeResourceDescriptor*), a callback object (*GetCallback*) notified when the GET operation is finished, passing the results (resources) to the *getReturned* method call, and optional get parameters (*Object[]*) described in Section 1.3.18.
***DeleteCallback* delete**	Initiates the DELETE operation, deleting a resource from a node in the DHT. The method arguments include the message recipient (*NodePointer*), the resource key (*BigInteger*), the delete criteria (*HyCubeResourceDescriptor*), a callback object (*DeleteCallback*) notified when the DELETE operation is finished, passing the status of the operation to the *deleteReturned* method call, and optional delete parameters (*Object[]*) described in Section 1.3.18.
***void* join**	Performs the JOIN procedure, connecting to the specified bootstrap node, and notifying the specified join callback object when the procedure terminates
***void* leave**	Performs the LEAVE operation - should be called before the node is disconnected from the DHT and discarded.
***LinkedBlockingQueue<ReceivedDataMessage>* registerPort**	Registers an incoming messages port and returns a queue to which the received messages will be inserted.
***void* registerMessageReceivedCallbackForPort**	Registers a callback object for a port (the callbeck object will be notified when a message is received)
***void* unregisterMessageReceivedCallbackForPort**	Unregisters the message received callback for port
***void* unregisterPort**	Unregisters the port
***void* registerMessageReceivedCallback**	Registers a callback for incoming messages (when ports are not used - configuration). The callback object is notified when a message is received.
***void* unregisterMessageReceivedCallback**	Unregisters the message received callback (when ports are not used)
***int* getMaxMessageLength**	Returns the maximum allowed message length
***int* getMaxMessageDataLength**	Returns the maximum allowed message data length
***boolean* isInitialized**	Returns a boolean value indicating whether the service instance is initialized
***boolean* isDiscarded**	Returns a boolean value indicating whether the service instance is discarded
***void* recover**	Explicit execution of the recovery procedure
***void* recoverNS**	Explicit execution of the neighborhood set recovery procedure
***void* discard**	Discards the node service instance

Table 1.2 Operations defined by *HyCubeNodeService*

1.2. Example of a node life cycle

The listing below presents an exemplary application creating a node instance using the *SimpleNodeService* service, registering an incoming messages port, sending a test message to itself, receiving the message, leaving the system and destroying the node instance.

```
public static void main(String[] args) {
  Environment environment = DirectEnvironment.initialize();
  JoinWaitCallback joinWaitCallback = new JoinWaitCallback();
  SimpleNodeService sns = HyCubeSimpleNodeService.initialize(environment,
    HyCubeNodeId.generateRandomNodeId(4, 32),
    "192.168.1.9:5000", "192.168.1.8:5000", joinWaitCallback, null, 0, true, null, null);
  LinkedBlockingQueue<ReceivedDataMessage> inMsgQueue = sns.registerPort((short)0);

  sendTestDataMessageToSelf(sns, "Test string");
  try {
    ReceivedDataMessage recMsg = inMsgQueue.take();
  } catch (InterruptedException e) {}
  System.out.println("Message received!: " + new String(recMsg.getData()));

  sns.discard();
  environment.discard();
}

public static void sendTestDataMessageToSelf(NodeService ns, String text) {
  MessageAckCallback mac = new WaitMessageAckCallback() {
    public void notifyDelivered(Object callbackArg) {
      super.notifyDelivered(callbackArg);
      System.out.println("Message DELIVERED.");
    }
    public void notifyUndelivered(Object callbackArg) {
      super.notifyUndelivered(callbackArg);
      System.out.println("Message UNDELIVERED");
    }
  };
  byte[] data = text.getBytes();
  DataMessage msg = new DataMessage(ns.getNode().getNodeId(), null, (short)0, (short)0, data);
  try {
    MessageSendInfo msi = ns.send(msg, mac, null);
    System.out.println("Message send info - serial no: " + msi.getSerialNo());
  } catch (NodeServiceException e) {}
}
```

1.3. Library architecture, core library classes and their implementations

The core of the *HyCube* library is a set of configurable classes allowing implementation of any DHT algorithm. The architecture is based on exchangeable components implementing certain interfaces, realizing individual system functions. *HyCube* library provides implementation of these components, realizing the *HyCube* algorithms.

Figure 1.1 presents the core of the library, and individual classes/interfaces are described in Section 1.3. Section 1.4 describes the realized solutions for event processing. Section 1.5 focuses on node services, which are wrapper classes for node instances, additionally providing event processing mechanisms.

1.3.1. Node and library architecture

The main class of the library is *Node*. An instance of this class represents a DHT node and exposes the node interface to node services - operations such as *join*, *sendMessage*, *lookup*, *search*, *put*, *refreshPut*, *get*, *delete*, *leave*, as well as methods registering incoming message queues (*ReceivedDataMessage* instances) and callbacks (*MessageReceivedCallback* instances). This class gathers the logic of all operations performed by nodes explicitly, as well as the processes taking place in the background. *Node* is instantiated by calling one of the variants of its static method *initializeNode*.

Most of the logic is realized by defining modules (classes) implementing certain interfaces, which are instantiated and initialized during the node initialization. Most of these modules (implementations) are defined in the configuration file (details in Section 1.3.4), but some of them are explicitly passed to the node initialization method upon creation (*initialize*). Using the abstraction of individual modules, the node calls configured implementations as a result of the API, as well as internal calls.

Because individual components require full access to the node state, and often, to other dependent components, a special object, implementing *NodeAccessor* interface, is passed to the components upon initialization *initialize* method call (by the *Node* instance). This object serves as a back-reference to the node instance, and may be used to access/modify node properties.

1.3.2. Environment

An object of a class extending abstract *Environment* class is passed to the node initialization on creation. This object represents the external environment, defines the time provider and

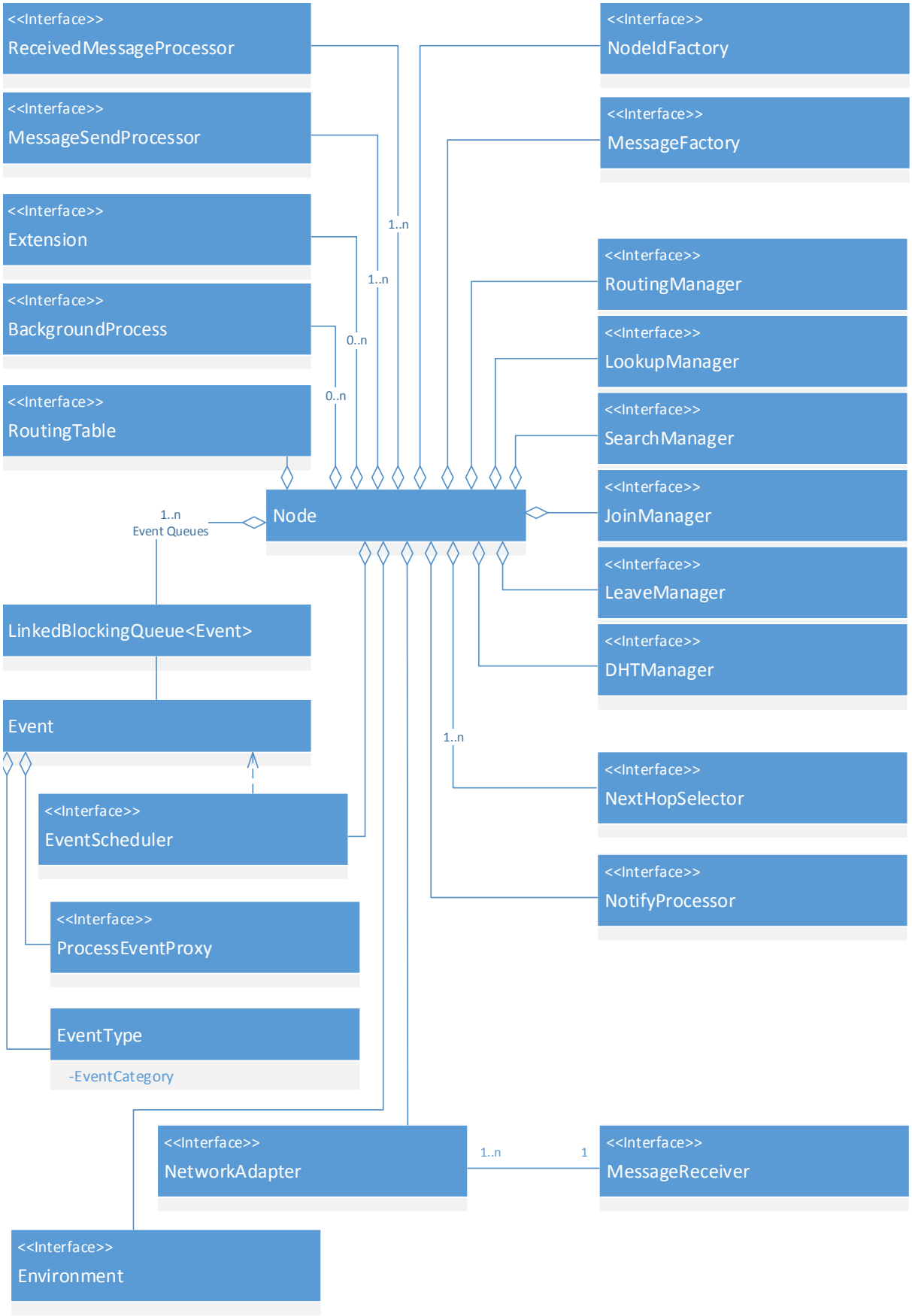

Figure 1.1 Node library architecture

contains node configuration (read from the configuration file). The information passed to the *Node* instance within the *Environment* object may be extended and used by any module (accessed through the instance of *NodeAccessor*). The default implementation of *Environment* - *DirectEnvironment* uses the system clock as the time provider, and reads configuration from a file (described in Section 1.3.4).

1.3.3. Time provider

An object implementing *TimeProvider* interface is passed to the *Node* instance as a property of the *Environment* object. The time provider object is expected to return a *long* value, representing the current time. The interpretation of the time value returned is implementation specific. The default implementation *SystemTimeProvider*, embedded in *DirectEnvironment* class, uses the system clock (*System.currentTimeMillis()* method) to retrieve the current time.

TimeProvider provides also methods for scheduling tasks (instances of *ScheduledTask*) for execution at a certain time in future. The default implementation uses the system clock (internally uses the *ScheduledThreadPoolExecutor* class).

1.3.4. Configuration

Node configuration is passed to the *Node* object on initialization, as a property of the *Environment* object. The node properties are accessed though an object implementing the *NodeProperties* interface. The methods of this interface allow access to the configuration properties for given property keys. The property keys and values are strings of characters. However, the *NodeProperties* implementations should provide conversion of values to simple types defined in the *ObjectToStringConverter.MappedType* enumeration, as well as conversion to enumerations (by default, the class *ObjectToStringConverter* is used for the conversion). Additionally, *NodeProperties* methods allow reading lists of values (of any type, including enumerations). Table 1.3 presents the data types supported, as well as their formats.

The default configuration technique employed by *HyCube* uses *ReaderNodeProperties* class, which reads configuration using an abstract reader object. The properties are, in the standard implementation of the *Environment* class (*DirectEnvironment*), read from a file (standard Java properties file format), using the *FileNodePropertiesReader* class (implementation of NodePropertiesReader). *FileNodePropertiesReader* reads properties from two files - default properties file and the application properties file, overwriting any property

Corresponding Java type	MappedType	Format
Integer	**MappedType.INT**	A signed decimal string representation, as expected by *java.lang.Integer.Integer(String s)* constructor
Short	**MappedType.SHORT**	A signed decimal string representation, as expected by *java.lang.Short.Short(String s)* constructor
Long	**MappedType.LONG**	A signed decimal string representation, as expected by *java.lang.Long.Long(String s)* constructor
Boolean	**MappedType.BOOLEAN**	A string equal to *true* or *false*, as expected by *java.lang.Boolean.Boolean(String s)* constructor
Float	**MappedType.FLOAT**	A string representation of the *float* value, as expected by *java.lang.Float.Float(String s)* constructor
Double	**MappedType.DOUBLE**	A string representation of the *double* value, as expected by *java.lang.Double.Double(String s)* constructor
Decimal	**MappedType.DECIMAL**	A string representation of a decimal number, as expected by the constructor *java.math.BigDecimal.BigDecimal(String val)*
BigInteger	**MappedType.BIGINTEGER**	A decimal string representation of a big integer, as expected by the constructor *java.math.BigInteger.BigInteger(String val)*
Date	**MappedType.DATE**	yyyy-MM-dd HH:mm:ss.SSS
String	**MappedType.STRING**	-
Enum	**Separate conversion methods are defined for enumerations**	Enumerations are converted using *Enum.valueOf(Class<T>, String)* and *Enum.toString()* methods.
List	**MappedType.LIST**	A list of elements of any of the types described above. The list elements are, by default, separated by the comma (“,”) character

Table 1.3 Configuration - property types

value redefined in the application properties file. Both properties files are expected to be located in the classpath of the application, and, if the default properties file name is not specified, the file name "*hycube_default.cfg*" is used (the name of the default configuration file delivered with the library).

The configuration architecture allows defining hierarchy of properties by separating keys at individual levels by the "." character. *NodeProperties* contains methods returning lower-level nested *NodeProperties* objects. Such an approach may be used for configuration of nested components, by passing the nested configuration (*NodeProperties*) objects to nested components. For example, for the following properties:

```
Node.PropertyKey1 = PropertyValue1
Node.PropertyKey2 = PropertyValue2
Node.Module1.PropertyKey3 = PropertyValue3
Node.Module1.PropertyKey4 = PropertyValue4
```

it is possible to access "Node.Module1.*" properties by a nested instance of *NodeProperties*, using property keys, relative to "Node.Module1", for example "PropertyKey3".

Moreover, *HyCube* configuration allows definition of nested properties (keys) using a different notation:

```
Node[Module1].PropertyKey3 = PropertyValue3
Node[Module1].PropertyKey4 = PropertyValue4
```

The effect is the same as in the previous example. However, such a notation allows easier identification of properties for subcomponents, for example:

```
Node.RoutingModuleKeys = Module1, Module2
Node.RoutingModules[Module1].PropertyKey1 = PropertyValue1
Node.RoutingModules[Module2].PropertyKey1 = PropertyValue1
```

Such properties may be accessed by retrieving nested properties until the lowest-level is reached, by specifying the full property key, or by specifying the key "Node.RoutingModules" (possibly just "RoutingModules" on the nested *NodeProperties* object) and specifying "Module1" or "Module2" as the "element name". All three methods are supported by the *HyCube* configuration API.

ReaderNodeProperties allows specifying the property determining the name space of the configuration (by default the property name is "configuration"). Such an approach makes it very easy to switch between many configuration variants stored within the same file, for example:

```
configuration = node.main
# configuration = node.simulation

node.main.Property1 = Value1
node.main. ...

node.simulation.Property1 = Value2
node.simulation. ...
```

making is possible to switch the whole configuration by changing the value of only one property - *configuration*. The *Node* instance, within the *Environment* object, receives the *NodeProperties* instance, representing a nested element defined by the configuration name space and nested *NodeProperties* instances are passed to individual components.

The character "#" at the beginning of a line denotes a comment, and causes that such a line is not processed by the configuration reader. It is also possible to use pointers (@ symbol) in the configuration to point property values (or whole nested properties sets) defined under a different key, for example:

```
Node.RoutingModuleKeys = Module1, Module2, Module3
Node.RoutingModules[Module1].PropertyKey1 = @conf1.PropertyValueX
Node.RoutingModules[Module2].PropertyKey1 = @conf1.PropertyValueX

Node.RoutingModules[Module3] = @Node.RoutingModules[Module1]

conf1.PropertyValueX = PropertyValue1
```

Whenever the symbol "@" is found at the beginning of the property value, the value (or the whole nested properties set) is first retrieved from the property that the symbol "@" points. Such a feature may become very useful when semantically the same property is a configuration parameter of multiple modules. In such a case, it is possible to define that property once, and use pointers to set the same value in configuration of all modules. The pointers mechanism is recursive, so multiple pointers may be used before the final value is obtained. In the above example, the property "PropertyValue1" for a nested *NodeProperties* object (representing "Node.RoutingModules[Module3]") would have value "PropertyValue1". Two pointers are resolved while retrieving the value. However, the pointer would be resolved only when the value or the nested property retrieved is the pointer itself. The pointer "@Node.RoutingModules[Module1]" would not be resolved in the example

above, if the property "Node.RoutingModules[Module3].PropertyValue1" was requested on an *NodeProperties* object relative to the root of the configuration.

For module (configurable objects created dynamically), a convention has been adopted to define a key (or keys) representing the module instances and configure their properties using the [] notation. Additionally, when the implementing class (of the module) is configurable, a property "Class" is introduced on the module level (defining the full class name). This property should be read by the parent module - the class name should be known to instantiate the child module. An example (configuration of the node ID factory and exemplary extensions) is presented in the listing below:

```
configuration = node

node.NodeIdFactory = HyCubeNodeIdFactory
node.NodeIdFactory[HyCubeNodeIdFactory].Class
                ↪ = net.hycube.core.HyCubeNodeIdFactory
node.NodeIdFactory[HyCubeNodeIdFactory].Dimensions = 4
node.NodeIdFactory[HyCubeNodeIdFactory].Levels = 32

node.Extensions = KeepAliveExtension, RecoveryExtension, ...
node.Extensions[KeepAliveExtension].Class
                ↪ = net.hycube.maintenance.HyCubeKeepAliveExtension
node.main.Extensions[KeepAliveExtension].PingInterval = 5000
...
node.Extensions[RecoveryExtension].Class
                ↪ = net.hycube.maintenance.HyCubeRecoveryExtension
...
```

After instantiating a module instance, as a rule, the initialization (*initialize()*) method of the created object is called, passing the configuration (a nested *NodeProperties* object) and the node accessor to the module. Module's *discard* method should be used to release all resources maintained by the module and perform any operations necessary when the module is disconnected from the node instance. This method is called by the *Node* object when the module is discarded.

The properties at the node level (used by the *Node* class) are described in Table 1.4. The properties of individual component implementations are presented in the following sections. It is also possible to define environment-level parameters by specifying the nested configuration

Property	Type	Value
MessageTTL	**Integer**	The TTL - maximum route length
MessageAckEnabled	**Boolean**	Determines whether the message delivery acknowledgments should be sent
DirectAck	**Boolean**	Determines whether ACK message are sent directly to the sender or routed
AckTimeout	**Integer**	The waiting time (milliseconds) for an ACK message, after which the original message is considered undelivered
ProcessAckInterval	**Integer**	The schedule interval for the background process processing awaiting ACK messages
ResendIfNoAck	**Boolean**	Determines whether messages should be resent when no ACK is received
SendRetries	**Integer**	Determines how many times messages should be resent if they are not delivered
NodeIdFactory	**Nested (+ Class)**	The configuration of the node ID factory module
MessageFactory	**Nested (+ Class)**	The configuration of the message factory module
RoutingTable	**Nested (+ Class)**	The configuration of the routing table structure module
NextHopSelectors	**Nested (+ Class)**	The configuration of the next hop selectors. Multiple next hop selectors may be defined.
RoutingManager	**Nested (+ Class)**	The configuration of the routing manager module
LookupManager	**Nested (+ Class)**	The configuration of the lookup manager module
SearchManager	**Nested (+ Class)**	The configuration of the search manager module
JoinManager	**Nested (+ Class)**	The configuration of the join manager module
LeaveManager	**Nested (+ Class)**	The configuration of the leave manager module
DHTManager	**Nested (+ Class)**	The configuration of the DHT manager module
NotifyProcessor	**Nested (+ Class)**	The configuration of the notify processor module
NetworkAdapter	**Nested (+ Class)**	The configuration of the network adapter module
MessageReceiver	**Nested (+ Class)**	The configuration of the message receiver module
ReceivedMessageProcessors	**Nested (+ Class)**	The configuration of the received message processors (multiple entries)
MessageSendProcessors	**Nested (+ Class)**	The configuration of the message send processors (multiple entries)
Extensions	**Nested (+ Class)**	The configuration of the extension modules (multiple entries)
BackgroundProcesses	**Nested (+ Class)**	The configuration of the background process modules (multiple entries)

Table 1.4 Node configuration properties

Property	Type	Value
Environment	**String**	*DirectEnvironment*
Environment[DirectEnvironment] ↪**.SchedulerThreadPoolSize**	**Integer**	The number of threads used by the system clock scheduler. This parameter is optional - if not specified, the default value (1) is used.

Table 1.5 DirectEnvironment configuration properties

Property	Type	Value
Class	**String**	*net.hycube.core.HyCubeNodeIdFactory*
Dimensions	**Integer**	The number of dimensions
Levels	**Integer**	The number of hierarchy levels

Table 1.6 HyCubeNodeIdFactory configuration properties

- “Environment” property at the root namespace level. Table 1.5 specifies the configuration of *DirectEnvironment*.

1.3.5. Node ID factory, Node ID

Classes implementing *NodeId* interface represent node identifiers. To allow dynamic creation of node ID instances, based on their binary or text representation, a factory class (implementing *NodeIdFactory* interface) should be defined. The methods of the factory object would return the instances of the node IDs of a specific type. The implementations of classes *NodeId* and *NodeIdFactory* define the object-byte and byte-object conversion for the node IDs, as well as comparisons and other operations on the node IDs.

To represent the node ID in *HyCube*, the class *HyCubeNodeId* and the factory class *HyCubeNodeIdFactory* (implementing the interfaces described) are used. *HyCubeNodeIdFactory* class should be configured for a certain number of dimensions and hierarchy levels (of the hierarchical hypercube), and returns instances of *HyCubeNodeId* class for the configured numbers of dimensions and hierarchy levels. The properties that should be configured for *HyCubeNodeIdFactory* module are presented in Table 1.6.

1.3.6. Routing table

Because the core of the library may be used for implementation of many different DHT systems, the structure of the routing table may vary depending on the overlay structure. The class representing the routing table should implement the *RoutingTable* interface.

Property	Type	Value
Class	**String**	*net.hycube.core.HyCubeRoutingTableImpl*
Dimensions	**Integer**	The number of dimensions
Levels	**Integer**	The number of hierarchy levels
NSSize	**Integer**	The size of the neighborhood set
RoutingTableSlotSize	**Integer**	The maximum number of nodes that may be stored in the routing table slot
UseSecureRouting	**Boolean**	Determines whether the secure routing tables should be maintained

Table 1.7 HyCubeRoutingTableImpl configuration properties

HyCubeRoutingTableImpl is a class implementing the *RoutingTable* interface, reflecting the structure of routing tables supported by nodes in *HyCube*: the primary routing table, the secondary routing table and the neighborhood set. Classes operating on the routing tables should cast the maintained routing table object to *HyCubeRoutingTableImpl*, and would then be able to access the *HyCube*-specific structure. *HyCubeRoutingTableImpl* configurable properties are presented in Table 1.7

The class *RoutingTableEntry* is used to store routing table entry information (for individual references). A routing table entry contains the node pointer (*NodePointer* class, which consists of the node ID and the node's network pointer - a network layer specific class implementing the *NetworkNodePointer* interface). In addition to a node pointer, routing table entries store the entry creation time, the distance to the node, a reference to the routing table slot containing the entry (routing table implementation specific), information whether the entry is enabled (used in next hop selection), discarded (the node should be removed). Furthermore, the routing table entry contains a map of additional data (objects) managed by individual modules. The map may be used to store the LNS/PNS indicators, cached results of extensive calculations performed for nodes, and many other. The map keys used by individual modules should be unique to avoid conflicts.

1.3.7. Message factory

A configurable message factory class should be an implementation of *MessageFactory* interface. The message factory object is responsible for creating message instances (instances of a class implementing *Message* interface). The implementations of *Message* and *MessageFactory* are responsible for message creation, object-byte and byte-object conversion,

Property	Type	Value
Class	**String**	*net.hycube.messaging.messages.HyCubeMessageFactory*
NodeIdFactory	**Nested (+ Class)**	The node ID factory module (used by the message factory) configuration (a pointer to the node ID factory configuration on the Node level may be used)
NetworkAddressByteLength	**Integer**	The number of hierarchy levels
HeaderExtensionsCount	**Integer**	The number of the header extensions
HeaderExtensionLengths	**List of Integers**	Lengths of the header extensions

Table 1.8 HyCubeMessageFactory configuration properties

as well as message header definition, and any other operations performed on the message objects.

The implementations specific to the *HyCube* protocol are *HyCubeMessage* and *HyCubeMessageFactory*. The header contains all the fields defined in Appendix **??**. Furthermore, the *HyCubeMessageFactory* may be configured to include additional header fields that may be used by modules (values are defined during object creation, or using getters/setters). The configuration parameters of *HyCubeMessageFactory* are presented in Table 1.8.

1.3.8. Extensions

The library architecture allows defining extensions (classes implementing the *Extension* interface). Extensions may be used to extend the library with any functionality or provide additional data structures maintained by nodes that may be used by the defined modules. An object of an extension is created during node initialization and may be accessed from outside the node by the node accessor or by an entry point (Section 1.3.10) exposed by node services (method *getExtensionEntryPoint* returning an *EntryPoint* object - if implemented). Extension objects are initialized (by calling the *initialize* method) before initialization of other modules, to allow individual modules to configure the extensions. However, if certain additional initialization activities should be performed after initialization of modules, they should be defined in the extensions's *postInitialize* method. Extension's *discard* method should be used to release all resources maintained by the extension and perform any operations necessary when the extension is disconnected from the node instance. This method is called by the *Node* object when the extension is discarded.

Property	Type	Value
Class	**String**	The full background process class name
ScheduleImmediately	**Boolean**	Determines whether the background process should be scheduled immediately after the initialization (*schedule* method call)

Table 1.9 BackgroundProcess configuration properties

1.3.9. Background processes

The library, by design, allows defining background processes - procedures executed in the node context in the background. Background processes may be scheduled or may be called explicitly. A background process should be an instance of a class implementing the *BackgroundProcess* interface. The interface's methods allow calling the background process, scheduling next execution, starting/stopping scheduled execution of the process and performing a check if the process is running. Classes implementing *BackgroundProcess* should also define the event type key for the background process events, and an entry point (Section 1.3.10) allowing access to the background process through the *Node* object (specifying the background process key). The *discard* method of a background process should be used to release all resources maintained by the background process instance and perform any operations necessary when the background process is disconnected from the node instance. This method is called by the *Node* object when the background process object is discarded. The common configuration properties of all background processes are defined in Table 1.9.

An abstract class *AbstractBackgroundProcess* implements several commonly used functions of background processes, like scheduling (configurable time interval), processing background events, starting/stopping scheduled process periodic execution, as well as returning an entry point (Section 1.3.10) - an instance of class *AbstractBackgroundProcess.BackgroundProcessEntryPointImpl*, implementing a proxy to basic operations performed on background process instances: starting, stopping, checking whether the process is started, and running the process. Most typical background processes may be defined by extending this class, in which case only one method, *doProcess()* - the process logic, remains to be defined. Properties required for background processes extending the class *AbstractBackgroundProcess* are defined in Table 1.10. The set of properties may be extended by the implementing classes.

Property	Type	Value
Class	**String**	The full background process class name
ScheduleImmediately	**Boolean**	Determines whether the background process should be scheduled immediately after the initialization (*schedule* method call)
ScheduleInterval	**Integer**	Schedule interval (in milliseconds)

Table 1.10 AbstractBackgroundProcess configuration properties

1.3.10. Entry points

Every module defined, as well as extensions and background processes, may define entry points - instances of classes implementing the *EntryPoint* interface (*BackgroundProcessEntryPoint* interface for background processes). Instances of the entry points, returned by appropriate getter methods (*getEntryPoint*, *getBackgroundProcessEntryPoint*), are publicly accessible through the *Node* object (for extensions and background processes, the methods expect the extension/process key as an argument), and may be used to access functions defined by modules, extensions, execute background processes or start/stop their scheduled execution through *EntryPoint* and (*BackgroundProcessEntryPoint* interfaces, or by casting to module-specific subclasses.

1.3.11. Next hop selectors

Next hop selectors are modules responsible for locating next hops in local routing tables. Multiple next hop selectors may be defined and individual next hop selectors may be accessed (for example through the node accessor object) to find next hop(s) for routing, lookup, search or other procedures. Next hop selectors should extend the abstract class *NextHopSelector*. The next hop selection methods (*findNextHop*, *findNextHops*) expect arguments of types *NodeId* (recipient node ID), *NextHopSelectionParameters* (next hop selection options, which may be extended to include algorithm-specific options), and an integer value determining the number of next hops to be returned.

Property	Type	Value
Class	**String**	*Class = net.hycube.nexthopselection.HyCubeNextHopSelector*
Dimensions	**Integer**	The number of dimensions of the hierarchical hypercube
Levels	**Integer**	The number of hierarchy levels of the hierarchical hypercube
UseRT1	**Boolean**	Determines whether the primary routing table should be used

UseRT2	**Boolean**	Determines whether the secondary routing table should be used
UseNS	**Boolean**	Determines whether the neighborhood set should be used
Metric	**Enum**	The routing metric (*Metric* enumeration)
UseSteinhausTransform	**Boolean**	Determines whether the Steinhaus transform should be used
DynamicSteinhausTransform	**Boolean**	Determines whether the Steinhaus point should be modified by nodes (variable Steinhaus metric)
RouteWithRegularMetricAfterSteinhaus	**Boolean**	Determines whether the next hop selection should continue without the use of the Steinhaus transform when no next hop is found
PrefixMismatchHeuristicEnabled	**Boolean**	Determines whether the prefix mismatch heuristic (PMH) should be applied
PrefixMismatchHeuristicMode	**Enum**	(*HyCubePrefixMismatchHeuristicMode*) - specifies the PMH mode - the PMH is applied based on the average or maximum neigh. set node distance
PrefixMismatchHeuristicFactor	**Double**	The prefix mismatch heuristic factor (λ). If equal to 0, routing will proceed without enforcing the prefix condition (the PMH will be applied every time, without checking distances).
PrefixMismatchHeuristicWhenNoNextHop	**Boolean**	Determines whether the PMH should be applied if no next hop is found
UseSteinhausTransformOnlyWithPMH	**Boolean**	Determines whether the Steinhaus transform should be applied only when PMH is applied - otherwise, routing proceeds according to the Euclidean metric
RespectNumOfCommonBitsInNextGroup	**Boolean**	Determines whether the next hop selection algorithm should respect the number of common bits in the first different digit of IDs
UseNSInFullScanWithoutPMH / UseRT1InFullScanWithoutPMH / UseRT2InFullScanWithoutPMH	**Boolean**	Determines whether all neighborhood set / primary routing table / secondary routing table nodes should be checked by the next hop selection algorithm (with PMH not applied). If *false* for the primary routing table, the routing table slots at the level corresponding to the common prefix length will be checked. If *false* for the secondary routing table, slots for all dimensions at levels $\log_2 d_{dim}$ and $\log_2 d_{dim}+1$ will be checked (d_{dim} is the distance to the destination node in dimension *dim*). If *false* for the neighborhood set, the neighborhood set will not be checked if it does not contain the destination node.
UseNSInFullScanWithPMH / UseRT1InFullScanWithPMH / UseRT2InFullScanWithPMH	**Boolean**	Determines whether all neighborhood set / primary routing table / secondary routing table nodes should be checked by the next hop selection algorithm (with PMH applied). If *false* for the primary routing table, no primary routing table slots will be checked. If *false* for the secondary routing table, slots for all dimensions at levels $\log_2 d_{dim}$ and $\log_2 d_{dim} + 1$ will be checked (d_{dim} is the distance to the destination node in dimension *dim*). If *false* for the neighborhood set, the neighborhood set will not be checked if it does not contain the destination node.
UseSecureRouting	**Boolean**	Determines whether secure routing is allowed
SkipRandomNumberOfNodesEnabled	**Boolean**	Determines whether skipping a random number of nodes in next hop selection is allowed
SkipRandomNumberOfNodesMean	**Double**	The mean of the normal distribution of the number of nodes to be skipped
SkipRandomNumberOfNodesStdDev	**Double**	The std. dev. of the normal distr. of the number of nodes to be skipped

SkipRandomNumberOfNodesAbsolute	**Boolean**	Specifies whether the absolute value of the generated number of nodes to skip should be used. Otherwise, for generated values smaller than 0, no nodes will be skipped.
SkipNodesNumMax	**Integer**	The maximum number of the nodes skipped
SkipNodesNumWhenRandomExceedsMax	**Integer**	The number of nodes that should be skipped when the generated random number exceeds the maximum value
ForceSkipRandomNumberOfNodes	**Boolean**	Determines whether the generated number of nodes should be skipped even if the returned number of next hops would be smaller than requested
SkipNodesIncludeExcactMatch	**Boolean**	Determines whether the set of skipped nodes may include the exact match

Table 1.11 HyCubeNextHopSelector configuration properties

Property	Type	Value
steinhausTransformApplied	**boolean**	Determines whether the Steinhaus transform is applied
steinhausPoint	**HyCubeNodeId**	Determines the current Steinhaus point
includeMoreDistantNodes	**boolean**	Determines whether next hop selection should include more distant nodes than the current node
skipTargetNode	**boolean**	Determines whether the exact match node (recipient ID) should be skipped in the next hop selection
includeSelf	**boolean**	Determines whether the current node (self) reference may be included in the results set
pmhApplied	**boolean**	Determines whether the prefix mismatch heuristic is applied
preventPmh	**boolean**	Determines whether the prefix mismatch heuristic should not be applied even if a message is already in the vicinity of the destination node
skipRandomNumOfNodesApplied	**boolean**	Determines whether skipping a random number of nodes in next hop selection is applied
secureRoutingApplied	**boolean**	Determines whether secure routing is applied

Table 1.12 Parameters of next hops selection (passed within HyCubeNextHopSelectorParameters)

The class *HyCubeNextHopSelector* is the implementation of the *HyCube* next hop selection algorithm (implementing the *NextHopSelector* methods). The configuration of the *HyCubeNextHopSelector* module consists of the parameters presented in Table 1.11 (the configuration parameters), and the runtime node selection parameters (values modified by nodes according to the algorithm). The latter are passed to the next hop selection methods (and modified values are returned) as an argument of type *HyCubeNextHopSelectionParameters* - the list of runtime parameters is presented in Table 1.12.

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeNotifyProcessor*
Dimensions	**Integer**	The number of dimensions of the hierarchical hypercube
Levels	**Integer**	The number of hierarchy levels of the hierarchical hypercube
NSSize	**Integer**	The maximum size of the neighborhood set
RoutingTableSlotSize	**Integer**	The maximum number of nodes stored in a routing table slot
UseRT1	**Boolean**	Determines whether the primary routing table should be used
UseRT2	**Boolean**	Determines whether the secondary routing table should be used
UseNS	**Boolean**	Determines whether the neighborhood set should be used
Metric	**Enum**	The routing metric (*Metric* enumeration)
ExcludeRT2ScopeFromRT1	**Boolean**	Determines whether nodes covered by secondary routing table slots should NOT be processed as primary routing table candidates
UseSecureRouting	**Boolean**	Determines whether secure routing tables should be maintained
UpdateNetworkAddressWhenDifferent	**Boolean**	If set to *true*, when processing a node, the network addresses will be updated for all references with the same node ID and different network addresses
RecentlyProcessedNodesRetentionTime	**Integer**	Time within the same node should not be processed more than once
RecentlyProcessedNodesCacheMaxSize	**Integer**	Maximum number of nodes stored for which the last notification time is stored
RTNodeSelector	**Nested (+ Class)**	The routing table node selector module
NSNodeSelector	**Nested (+ Class)**	The neighborhood set node selector module
SecureRTNodeSelector	**Nested (+ Class)**	The secure routing table node selector module

Table 1.13 HyCubeNotifyProcessor configuration properties

1.3.12. Notify processor

The notify processor component (a class extending the abstract class *NotifyProcessor*) is responsible for processing notifications of existence of other nodes (for example when a NOTIFY or RECOVERY_REPLY message is received). The abstract method of this class (*processNotify*) takes two arguments - the new node reference and the current time stamp.

The *HyCube* implementation of the notify processor is the class *HyCubeNotifyProcessor*. The notify processor finds appropriate routing table slot(s) (including the neighborhood set), and, within the routing table slot level, the best node(s) are determined based on two other components: routing table node selector (extending the class *HyCubeRTNodeSelector*) and neighborhood set node selector (extending *HyCubeNSNodeSelector*). Both classes *HyCubeRTNodeSelector* and *HyCubeNSNodeSelector* declare abstract methods processing the

Property	Type	Value
Class	**String**	*net.hycube.rtnodeselection.HyCubeSimpleRTNodeSelector*

Table 1.14 HyCubeSimpleRTNodeSelector configuration properties

new node in the context of existing nodes - the routing table slot reference is also passed to the method. A separate routing table node selector is defined for choosing nodes for secure routing tables. The configuration parameters of *HyCubeNotifyProcessor* are presented in Table 1.13. The following routing table node selectors were implemented:

- *HyCubeSimpleRTNodeSelector* - adds a new node to a routing table slot only when the slot is not full. The properties of this node selector are presented in table 1.14.
- *HyCubeLnsRTNodeSelector* - realizes the LNS technique adopted by *HyCube*, connected with *HyCube*'s keep-alive mechanism. The properties of this node selector are presented in table 1.15.
- *HyCubeSecureRTNodeSelector* - realizes the *HyCube* secure node selection algorithm. The properties of this node selector are presented in table 1.16.

and the following neighborhood set node selectors:

- *HyCubeDistanceNSNodeSelector* - realizes the neighborhood set node selection based only on distances (selects closest nodes). The properties of this node selector are presented in table 1.17.
- *HyCubeBalancedRingNSNodeSelector* - realizes the neighborhood set node selection based on distances and ensuring uniform distribution of nodes in terms of directions on a logical ring (half of nodes would be successors, half would be predecessors). The properties of this node selector are presented in table 1.18.
- *HyCubeBalancedOrthantsNSNodeSelector* - realizes the neighborhood set node selection based on distances, tending to achieve equal numbers of neighbors in individual orthants of the system of coordinates with the center at the address (ID) of the node whose neighborhood set is considered. The configuration properties of this node selector are presented in Table 1.19.

Property	Type	Value
Class	**String**	*net.hycube.rtnodeselection.HyCubeLnsRTNodeSelector*
LnsIndicatorRteKey	**String**	Specifies the key under which the LNS indicator value is stored in routing table entries. This value should be the same as the value of the parameter "PingResponseIndicatorRteKey" of the keep-alive extension (Section 1.3.22), as both modules operate on the same values.
InitialLnsIndicatorValue	**Double**	Specified the initial value of the LNS indicator (for newly added nodes)
LnsIndicatorReplaceThreshold	**Double**	Specifies the LNS replace threshold - routing table references with values of the LNS indicator lower than this threshold may be replaced. This parameter value should be a pointer to the "PingResponseIndicatorReplaceThreshold" property of the *HyCubeKeepAliveExtension* extension (Section 1.3.22).
KeepAliveExtensionKey	**String**	Specifies the keep-alive extension key (Section 1.3.22)
UseKeepAliveExtensionLnsIndicatorCache	**Boolean**	Determines whether the LNS indicator value should be cached for nodes removed from routing tables and used when checking these nodes again

Table 1.15 HyCubeLnsRTNodeSelector configuration properties

Property	Type	Value
Class	**String**	*net.hycube.rtnodeselection.HyCubeSecureRTNodeSelector*
Metric	**Enum**	The routing metric (*Metric* enumeration)
Dimensions	**Integer**	The number of dimensions of the hierarchical hypercube
Levels	**Integer**	The number of hierarchy levels of the hierarchical hypercube
XorNodeIdChangeAfter	**Integer**	Determines after how many node checks the secret Node ID should be regenerated
DistFunRteKey	**String**	Specifies the key under which the distance function data is stored in routing table entries

Table 1.16 HyCubeSecureRTNodeSelector configuration properties

Property	Type	Value
Class	**String**	*net.hycube.rtnodeselection.HyCubeDistanceNSNodeSelector*

Table 1.17 HyCubeDistanceNSNodeSelector configuration properties

Property	Type	Value
Class	**String**	*net.hycube.rtnodeselection.HyCubeBalancedRingNSNodeSelector*
SemiringNoRteKey	**String**	Specifies the key under which the semi-ring number of the neighbors is stored in routing table entries

Table 1.18 HyCubeBalancedRingNSNodeSelector configuration properties

Property	Type	Value
Class	**String**	*net.hycube.rtnodeselection.HyCubeBalancedOrthantsNSNodeSelector*
Dimensions	**Integer**	The number of dimensions of the hierarchical hypercube
OrthantNoRteKey	**String**	Specifies the key under which the orthant number of the neighbors is stored in routing table entries

Table 1.19 HyCubeBalancedOrthantsNSNodeSelector configuration properties

1.3.13. Routing manager

The routing manager is a module (a class implementing *RoutingManager* interface) responsible for routing messages. The method *routeMessage* takes two arguments - *MessageSendProcessInfo* (information about the message to be sent) and *wait* (a boolean flag determining whether the sending should be performed in the foreground or the background). The *MessageSendProcessInfo* object contains the message object (*Message*), the direct recipient to which the message should be sent (*NodePointer*), information whether the message should be processed before sending (by message send processors), and routing parameters (an algorithm specific *Object[]* array) specified for the message being sent.

The class *HyCubeRoutingManager* is the implementation of the *RoutingManager* that routes messages based on the next hop selector module, updates message's TTL and hop count values, as well as provides anonymous and registered routes functionalities. The properties of *HyCubeRoutingManager* are presented in Table 1.20. The expected routing parameters (specified within the *MessageSendProcessInfo* object) are presented in Table 1.21. The class *HyCubeRoutingManager* provides helper methods for creating and retrieving the runtime parameters - operating on the *Object[]* array. If the default value of a runtime parameter (or the value is missing - null) is passed to the routing manager, and the appropriate message header field of the message passed is given a non-default value, the non-default value is used. If the value is specified by both, the message header field and the value passed within the *Object[]* runtime parameters, the value of the runtime parameter is used.

1.3.14. Lookup manager

The lookup manager module (a class implementing *LookupManager* interface) is responsible for performing lookup requests. Methods performing the lookup take the lookup node ID as an argument, as well as optional lookup parameters (an algorithm specific *Object[]*

Property	Type	Value
Class	**String**	*net.hycube.routing.HyCubeRoutingManager*
NextHopSelectorKey	**String**	The next hop selector key (among possible multiple next hop selectors defined)
UseSteinhausTransform	**Boolean**	Determines whether the Steinhaus transform should be enabled for routed messages
AllowRegisteredRoutes	**Boolean**	Determines whether registered routes are allowed (otherwise messages sent with "Register route" header option set to *true* will be dropped)
RegisteredRoutesRetentionTime	**Integer**	Specifies the time for which the registered route information should be maintained
AllowAnonymousRoutes	**Boolean**	Determines whether anonymous routes are allowed (otherwise messages sent with "Anonymous route" header option set to *true* will be dropped)
ConcealTTL	**Boolean**	Determines whether the TTL message field should be concealed for all messages (the method is specified by the parameters "DecreaseTTLProbability" and "IncreaseTTLByRandomNum")
DecreaseTTLProbability	**Double**	Determines the probability with the nodes along the routes should decrease the TTL values
IncreaseTTLByRandomNum	**Boolean**	Determines whether the TTL should be increased by nodes along the routes by a random number (normal distribution)
IncreaseTTLRandomMean	**Double**	Specifies the mean of the distribution (normal distribution) for generating the TTL increase.
IncreaseTTLRandomStdDev	**Double**	Specifies the std. deviation of the distribution (normal distribution) for generating the TTL increase.
IncreaseTTLRandomAbsolute	**Boolean**	Specifies whether the absolute value of the generated random TTL increase should be used. Otherwise, for generated values smaller than 0, the TTL will not be increased.
IncreaseTTLRandomModulo	**Double**	Specifies the maximum TTL increase. When the randomly generated TTL increase is greater, the increase will be recalculated modulo the value of this parameter.
ConcealHopCount	**Boolean**	Determines whether the hop count message field should be concealed (set to its maximum value) for all messages
EnsureSteinhausPointAnonymity	**Boolean**	Determines whether the initial Steinhaus point ID for a message should be modified when the sender node is one of the most distant nodes to the recipient (in terms of distances between identifiers in the hierarchical hypercube)
SteinhausPointAnonymityDistanceFactor	**Double**	Determines the ratio of closer neighborhood set nodes to the destination (than the sending node), above which (inclusive) the Steinhaus point anonymity will be enabled.

Table 1.20 HyCubeRoutingManager configuration properties

Parameter name	Type	Value
secureRouting	**Boolean**	A flag indicating whether secure routing tables should be used (by all nodes along the route) for next hops selection for the message being sent (default value *false*)
skipRandomNextHops	**Boolean**	A flag indicating whether random numbers of nodes should be skipped (by all nodes along the route) in next hop selection for the message being sent (default value *false*)
regusterRoute	**Boolean**	A flag indicating whether the route should be registered by nodes (default value *false*)
routeBack	**Boolean**	A flag indicating whether the message should be routed back along a registered route (default value *false*)
routeId	**Integer**	The route ID, applicable when the message is routed back along a registered route (default value 0)
anonymousRoute	**Boolean**	A flag indicating whether the message should be routed anonymously (forced anonymous route, "Anonymous route" header option set to *true*). The default value is *false*. If the value is *true*, every node along the route should replace the message's original sender node ID and network address with its own node ID and network address. Additionally the option forces concealing the "TTL" and the "Hop count" header fields, and modifies the initial Steinhaus point if the sending node ID is one of the most distant nodes to the recipient in the hierarchical hypercube (the new value is the second best next hop found in the routing tables). This option can be set to *true* together with "registerRoute" or "routeBack" option, in which case, the message will be routed along a registered route, and concealing the "TTL", "Hop count" and "Steinhaus point" fields will be forced.

Table 1.21 HyCubeRoutingManager runtime routing parameters

Property	Type	Value
Class	**String**	*net.hycube.lookup.HyCubeLookupManager*
NextHopSelectorKey	**String**	The next hop selector key (among possible multiple next hop selectors defined)
LookupCallbackEventKey	**String**	Lookup callback event key
LookupRequestTimeoutEventKey	**String**	Lookup request timeout event key
DefaultBeta	**Integer**	Default value of the parameter *beta* (β) - the maximum number of nodes returned by intermediate nodes
DefaultGamma	**Integer**	Default value of the parameter *gamma* (γ) - the number of temporary nodes stored during the lookup
Metric	**Enum**	The routing metric (*Metric* enumeration)
UseSteinhausTransform	**Boolean**	Determines whether the Steinhaus transform should be enabled in the next lookup hop selection
LookupRequestTimeout	**Integer**	Lookup request timeout (milliseconds)

Table 1.22 HyCubeLookupManager configuration properties

array). Additionally, an object of type *LookupCallback* is passed to the the lookup method calls, and an optional callback argument of type *Object*. The callback object's *lookupReturned* method is called when the lookup procedure terminates, and the result, as well as the callback argument specified, are passed to the method call. The classes implementing the *LookupManager* interface should also define the event types for callback events and for request timeout events (Section 1.4). An entry point to the lookup manager may also be defined by implementing the method *getEntryPoint()* returning an object of type *EntryPoint*.

Class *HyCubeLookupManager* is an implementation of the *LookupManager* interface, realizing the lookup procedure of *HyCube*. Additionally, *HyCubeLookupManager* implements methods processing received lookup requests and responses (called by received message processors, described in Section 1.3.20). The configuration parameters of *HyCubeLookupManager* are presented in Table 1.22. The expected runtime lookup parameters (elements of the *Object[]* array - the argument of the lookup method call) are presented in Table 1.23. The class *HyCubeLookupManager* provides helper methods for creating and retrieving the runtime parameters - operating on the *Object[]* array.

Parameter name	Type	Value
beta	**Integer**	The maximum number of nodes returned by intermediate nodes. If not specified (or equal to 0), the default value is used.
gamma	**Integer**	The number of temporary nodes stored during the lookup. If not specified (or equal to 0), the default value is used.
secureLookup	**Boolean**	A flag indicating whether secure routing tables should be used for next hop selection (default value *false*)
skipRandomNextHops	**Boolean**	A flag indicating whether random numbers of nodes should be skipped (by nodes receiving LOOKUP messages) in next hop selection (default value *false*)

Table 1.23 HyCubeLookupManager runtime lookup parameters

1.3.15. Search manager

The search manager module (a class implementing *SearchManager* interface) is responsible for performing search requests. Methods initiating the search procedure take the search node ID as an argument, k - the number of nodes to be found, as well as optional: search parameters (an algorithm specific *Object[]* array), an optional set of references to initial search nodes (nodes to which initial requests are sent), and a flag determining whether the exact match node should be skipped in searches (local next hop selection) - *ignoreTargetNode*. Additionally, an object of type *SearchCallback* is passed to the the search method calls (and an optional callback argument of type *Object*). The callback object's *searchReturned* method is called when the search procedure terminates, and the result, as well as the callback argument specified, are passed to the method call. The classes implementing the *SearchManager* interface should also define the event types for callback events and for request timeout events (event processing described in Section 1.4). An entry point to the search manager may also be defined by implementing the method *getEntryPoint()* returning an object of type *EntryPoint*.

Class *HyCubeSearchManager* is the implementation of the *SearchManager* interface, realizing the search procedure of *HyCube*. Additionally, *HyCubeSearchManager* implements methods processing received search requests and responses (called by received message processors, described in Section 1.3.20). The configuration parameters of *HyCubeSearchManager* are presented in Table 1.24, and the expected runtime search parameters (elements of the *Object[]* array - the argument of the search method call) are presented in Table 1.25. The class *HyCubeSearchManager* provides helper methods for creating and retrieving the runtime parameters - operating on the *Object[]* array.

Property	Type	Value
Class	**String**	*net.hycube.search.HyCubeSearchManager*
NextHopSelectorKey	**String**	The next hop selector key (among possible multiple next hop selectors defined)
SearchCallbackEventKey	**String**	Search callback event key
SearchRequestTimeoutEventKey	**String**	Search request timeout event key
DefaultAlpha	**Integer**	Default value of the parameter *alpha* (α) - the number of closest nodes to which search requests are sent - parallelism factor
DefaultBeta	**Integer**	Default value of the parameter *beta* (β) - the maximum number of nodes returned by intermediate nodes
DefaultGamma	**Integer**	Default value of the parameter *gamma* (γ) - the number of temporary nodes stored during the search
Metric	**Enum**	The routing metric (*Metric* enumeration)
UseSteinhausTransform	**Boolean**	Determines whether the Steinhaus transform should be enabled in the next hop selection
SearchRequestTimeout	**Integer**	Search request timeout (milliseconds)

Table 1.24 HyCubeSearchManager configuration properties

Parameter name	Type	Value
alpha	**Integer**	The number of closest nodes to which search requests are sent (parallelism factor). If not specified (or equal to 0), the default value is used.
beta	**Integer**	The maximum number of nodes returned by intermediate nodes. If not specified (or equal to 0), the default value is used.
gamma	**Integer**	The number of temporary nodes stored during the search. If not specified (or equal to 0), the default value is used.
secureSearch	**Boolean**	A flag indicating whether secure routing tables should be used for next hop selection (default value *false*)
skipRandomNextHops	**Boolean**	A flag indicating whether random numbers of nodes should be skipped (by nodes receiving SEARCH messages) in next hop selection (default value *false*)

Table 1.25 HyCubeSearchManager runtime search parameters

1.3.16. Join manager

The join manager module (a class implementing the *JoinManager* interface) is responsible for realization of node joins. Methods performing the join take the following arguments: the bootstrap node network address (*String*), optional join parameters (an algorithm specific *Object[]* array), an object of type *JoinCallback*, and an optional callback argument of type *Object*). The callback object's *joinReturned* method is called when the join procedure terminates, and the callback argument specified is passed to the method call. Classes implementing *JoinManager* interface should also define the event type for the callback event (Section 1.4). An entry point to the join manager may also be defined by implementing the method *getEntryPoint()* returning an object of type *EntryPoint*. The ID of the joining node is expected to be set before the node join procedure is initiated.

In the *HyCube* library, two different join managers have been implemented: *HyCubeSearchJoinManager* (realizing the search join procedure) and *HyCubeRouteJoinManager* (realizing the route join procedure). Both join managers implement the join request initiation, as well as the logic of processing received join requests and responses (called by received message processors, described in Section 1.3.20). The configuration parameters of *HyCubeSearchJoinManager* (search join technique) are presented in Table 1.26, and the expected runtime join parameters (elements of the *Object[]* array - the argument of the join method call) are presented in Table 1.27. The configuration of *HyCubeRouteJoinManager* (route join technique) is presented in Table 1.28 and the runtime join parameters are presented in Table 1.29. Both join managers provide helper methods for creating and retrieving the runtime parameters - operating on the *Object[]* array.

1.3.17. Leave manager

The leave manager module (a class implementing the *LeaveManager* interface) is responsible for running procedures performed when a node leaves the DHT. The method *leave* (no arguments) should implement the leaving logic.

In the *HyCube* implementation (*HyCubeLeaveManager* class), the leaving node sends all neighborhood set references to all nodes in its neighborhood set. *HyCubeLeaveManager* also implements a method processing received LEAVE messages (the method is called by a received message processor - the message processors are described in Section 1.3.20). A node receiving a LEAVE message removes the leaving node from the routing tables, and (depending on

Property	Type	Value
Class	**String**	*net.hycube.join.searchjoin.HyCubeSearchJoinManager*
NextHopSelectorKey	**String**	The next hop selector key (among possible multiple next hop selectors defined)
JoinCallbackEventKey	**String**	Join callback event key
JoinRequestTimeoutEventKey	**String**	Join request timeout event key
JoinAlpha	**Integer**	The value of the parameter *alpha* (α) - the number of closest nodes to which search requests are sent - the parallelism factor used by the join algorithm
DefaultBeta	**Integer**	The value of the parameter *beta* (β) - the maximum number of nodes returned by intermediate nodes used by the join algorithm
DefaultGamma	**Integer**	The value of the parameter *gamma* (γ) - the number of temporary nodes stored during the lookup used by the join algorithm
Metric	**Enum**	The routing metric (*Metric* enumeration)
UseSteinhausTransform	**Boolean**	Determines whether the Steinhaus transform should be enabled in the next hop selection
JoinRequestTimeout	**Integer**	Join request timeout (milliseconds)
SendClosestInInitialJoinReply	**Boolean**	Determines whether the closest nodes to the joining node ID should be returned in responses to initial join requests
IncludeNSInInitialJoinReply	**Boolean**	Determines whether the neighborhood set references should be returned in responses to initial join requests
IncludeRTInInitialJoinReply	**Boolean**	Determines whether the routing table references should be returned in responses to initial join requests
IncludeSelfInInitialJoinReply	**Boolean**	Determines whether a reference to self should be returned in responses to initial join requests
MarkInitialJoinReplySenderAsResponded	**Boolean**	Determines whether the nodes to which initial requests were sent should be marked as requested (no additional join requests would be sent to those nodes in the first search phase)
RecoveryNSAfterJoin	**Boolean**	Determines whether nodes should perform a neighborhood set recovery procedure after joining
RecoveryAfterJoin	**Boolean**	Determines whether nodes should perform a full recovery procedure after joining
RecoveryExtensionKey	**String**	Specifies the key of the recovery extension that is used to perform the recovery after joining
DiscoverPublicNetworkAddress	**Boolean**	Determines if the requested node should return the connecting node's public network address, which is then stored by the connecting node

Table 1.26 HyCubeSearchJoinManager configuration properties

Parameter name	Type	Value
secureSearch	**Boolean**	A flag indicating whether secure routing tables should be used for next hop selection (default value *false*)
skipRandomNextHops	**Boolean**	A flag indicating whether random numbers of nodes should be skipped (by nodes receiving JOIN messages) in next hop selection (default value *false*)

Table 1.27 HyCubeSearchJoinManager runtime join parameters

the configuration) processes the list of the nodes received as routing table candidates (notify processor). The configuration parameters of *HyCubeLeaveManager* are presented in Table 1.30.

1.3.18. DHT manager

The DHT manager module (a class implementing the *DHTManager* interface) is responsible for performing operations on resources: inserting (PUT) resources into the DHT, refreshing (REFRESH_PUT, retrieving (GET) and deleting (DELETE), as well as performing the same operations on resources in the local storage, called when appropriate requests are received. Additionally, the class should define (implementing appropriate getters) callback event types and an entry point.

The class *HyCubeRoutingDHTManager* implements the PUT, REFRESH_PUT, GET and DELETE procedures of *HyCube*, including processing received request and response messages, as well as resource operations on the local storage. The interface *HyCubeDHTManager*, extending the interface *DHTManager*, specifies additional methods, using specific types used by *HyCubeRoutingDHTManager* - resources are represented by *HyCubeResource* objects, and resource descriptors/criteria are represented by *HyCubeResourceDescriptor* objects (*HyCubeRoutingDHTManager* implements this interface). The methods inserting, refreshing, retrieving and deleting resources from the DHT expect the following arguments: the resource key (*BigInteger*), optional node pointer (the node to which the request should be sent directly), an object representing the resource (PUT), or resource criteria (used for specifying resources for REFRESH_PUT, GET and DELETE), a callback object, and a callback argument that would be passed to the callback method when the request terminates. The methods operating on the local storage expect the following arguments: the resource key, the requesting node ID, and the object representing resource/criteria (implementation specific). The methods of the DHT manager (implementing operations PUT, REFRESH_PUT, GET, DELETE, as well

Property	Type	Value
Class	**String**	*net.hycube.join.routejoin.HyCubeRouteJoinManager*
NextHopSelectorKey	**String**	The next hop selector key (among possible multiple next hop selectors defined)
PMHDisabledForJoinMessages	**Boolean**	Determines whether the prefix mismatch heuristic should be disabled for JOIN messages
UseSteinhausTransform	**Boolean**	Determines whether the Steinhaus transform should be enabled in the next hop selection
JoinCallbackEventKey	**String**	Join callback event key
JoinTimeoutEventKey	**String**	Join timeout event key
WaitAfterFinalJoinReplyTimeoutEventKey	**String**	Event key for join timeout after receiving the final join reply (after this timeout, the procedure terminates even if not all responses have been received from intermediate nodes)
JoinTimeout	**Integer**	Join timeout (milliseconds)
WaitTimeAfterFinalJoinReply	**Integer**	Join timeout (milliseconds) after receiving the final join reply (after this timeout, the procedure terminates even if not all responses have been received from intermediate nodes)
IncludeNSInJoinReply	**Boolean**	Determines whether the neighborhood set references should be returned in join responses
IncludeRTInJoinReply	**Boolean**	Determines whether the routing table references should be returned in join responses
IncludeSelfInJoinReply	**Boolean**	Determines whether a reference to self should be returned in join responses
IncludeNSInJoinReplyFinal	**Boolean**	Determines whether the neighborhood set references should be returned in final join responses
IncludeRTInJoinReplyFinal	**Boolean**	Determines whether the routing table references should be returned in final join responses
IncludeSelfInJoinReplyFinal	**Boolean**	Determines whether a reference to self should be returned in final join responses
RecoveryNSAfterJoin	**Boolean**	Determines whether nodes should perform a neighborhood set recovery procedure after joining
RecoveryAfterJoin	**Boolean**	Determines whether nodes should perform a full recovery procedure after joining
RecoveryExtensionKey	**String**	Specifies the key of the recovery extension that is used to perform the recovery after joining
DiscoverPublicNetworkAddress	**Boolean**	Determines if the requested node should return the connecting node's public network address, which is then stored by the connecting node

Table 1.28 HyCubeRouteJoinManager configuration properties

Parameter name	Type	Value
secureSearch	**Boolean**	A flag indicating whether secure routing tables should be used for next hop selection (default value *false*)
skipRandomNextHops	**Boolean**	A flag indicating whether random numbers of nodes should be skipped in next hop selection by all nodes routing the JOIN message (default value *false*)

Table 1.29 HyCubeRouteJoinManager runtime join parameters

Property	Type	Value
Class	**String**	*net.hycube.leave.HyCubeLeaveManager*
BlockingSendLeave	**Boolean**	Determines whether the leave manager should send LEAVE messages in the blocking mode (waiting for the messages to be physically sent)
WaitAfterSendLeaveTime	**Integer**	Specifies the time (in milliseconds) that nodes should wait after sending LEAVE messages (before returning)
ProcessReceivedNodes	**Boolean**	Determines whether the leave manager should process references received within LEAVE messages

Table 1.30 HyCubeLeaveManager configuration properties

as their corresponding local storage operations) have an optional argument - an *Object[]* array, specifying runtime parameters. In the *HyCubeRoutingDHTManager* implementation, the runtime parameters are used only by procedures PUT (Table 1.31), REFRESH_PUT (Table 1.32), GET (Table 1.33), DELETE (Table 1.34). The methods operating on the local storage do not expect any additional runtime parameters in the *Object[]* array. The class *HyCubeRoutingDHTManager* provides helper methods for creating and retrieving the runtime parameters - operating on the *Object[]* array. The configuration properties of the class *HyCubeRoutingDHTManager* are presented in Table 1.35.

The class *HyCubeRoutingDHTManager* also implements the functions performed by background processes defined for resource management:

- *DHTBackgroundProcess* - The background process run every specified time interval, checking resources' refresh times and deleting expired resources .
- *ReplicationBackgroundProcess* - The background process run every specified time interval, performing replication - sending replication information to neighbors.

Both background processes extend the abstract class *AbstractBackgroundProcess* and their configuration includes only parameters expected by *AbstractBackgroundProcess*.

Parameter name	Type	Value
exactPut	**Boolean**	Indicates whether the PUT request is an exact PUT request sent to a node that should process it and not route it any further. If this flag is set to *true*, the direct recipient of the request must be specified. The default value is *false*.
secureRouting	**Boolean**	Indicates whether secure routing tables should be used for next hop selection (def. value *false*)
skipRandomNextHops	**Boolean**	Indicates whether random numbers of nodes should be skipped in next hop selection by all nodes routing the PUT message (default value *false*)
registerRoute	**Boolean**	Indicates whether the request should be routed using a registered route (in which case, the response would be sent back along the same path). The default value is *false*.
anonymousRoute	**Boolean**	Indicates whether the request should be routed anonymously (forced anonymous route, "Anonymous route" header option set to *true*). The default value is *false*. If the value is *true*, every node along the route should replace the message's original sender node ID and network address with its own node ID and network address. Additionally the option forces concealing the "TTL" and the "Hop count" header fields, and modifies the initial Steinhaus point if the sending node ID is one of the most distant nodes to the recipient in the hierarchical hypercube (the new value is the second best next hop found in the routing tables). This option can be set to *true* together with the "registerRoute" option, in which case, the message will be routed along a registered route, concealing the "TTL", "Hop count" and "Steinhaus point".

Table 1.31 HyCubeRoutingDHTManager runtime PUT parameters

Parameter name	Type	Value
exactRefreshPut	**Boolean**	Indicates whether the REFRESH_PUT request is an exact REFRESH_PUT request sent to a node that should process it and not route it any further. If this flag is set to *true*, the direct recipient of the request must be specified. The default value is *false*.
secureRouting	**Boolean**	Indicates whether secure routing tables should be used for next hop selection (def. value *false*)
skipRandomNextHops	**Boolean**	Indicates whether random numbers of nodes should be skipped in next hop selection by all nodes routing the REFRESH_PUT message (default value *false*)
registerRoute	**Boolean**	Indicates whether the request should be routed using a registered route (in which case, the response would be sent back along the same path). The default value is *false*.
anonymousRoute	**Boolean**	Indicates whether the request should be routed anonymously (forced anonymous route, "Anonymous route" header option set to *true*). The default value is *false*. If the value is *true*, every node along the route should replace the message's original sender node ID and network address with its own node ID and network address. Additionally the option forces concealing the "TTL" and the "Hop count" header fields, and modifies the initial Steinhaus point if the sending node ID is one of the most distant nodes to the recipient in the hierarchical hypercube (the new value is the second best next hop found in the routing tables). This option can be set to *true* together with the "registerRoute" option, in which case, the message will be routed along a registered route, concealing the "TTL", "Hop count" and "Steinhaus point".

Table 1.32 HyCubeRoutingDHTManager runtime REFRESH_PUT parameters

Parameter name	Type	Value
exactGet	**Boolean**	Indicates whether the GET request is an exact GET request sent to a node that should process it and not route it any further. If this flag is set to *true*, the direct recipient of the request must be specified. The default value is *false*.
findClosestNode	**Boolean**	Indicates whether the GET request should be routed to the closest node to the resource key, or the resource(s) should be returned by the first node on the route that stores it (def. value *false*).
secureRouting	**Boolean**	Indicates whether secure routing tables should be used for next hop selection (def. value *false*)
skipRandomNextHops	**Boolean**	Indicates whether random numbers of nodes should be skipped in next hop selection by all nodes routing the GET message (default value *false*)
registerRoute	**Boolean**	Indicates whether the request should be routed using a registered route (in which case, the response would be sent back along the same path). The default value is *false*.
anonymousRoute	**Boolean**	Indicates whether the request should be routed anonymously (forced anonymous route, "Anonymous route" header option set to *true*). The default value is *false*. If the value is *true*, every node along the route should replace the message's original sender node ID and network address with its own node ID and network address. Additionally the option forces concealing the "TTL" and the "Hop count" header fields, and modifies the initial Steinhaus point if the sending node ID is one of the most distant nodes to the recipient in the hierarchical hypercube (the new value is the second best next hop found in the routing tables). This option can be set to *true* together with the "registerRoute" option, in which case, the message will be routed along a registered route, concealing the "TTL", "Hop count" and "Steinhaus point".

Table 1.33 HyCubeRoutingDHTManager runtime GET parameters

Parameter name	Type	Value
exactDelete	**Boolean**	Indicates whether the DELETE request is an exact DELETE request sent to a node that should process it and not route it any further. If this flag is set to *true*, the direct recipient of the request must be specified. The default value is *false*.
secureRouting	**Boolean**	Indicates whether secure routing tables should be used for next hop selection (def. value *false*)
skipRandomNextHops	**Boolean**	Indicates whether random numbers of nodes should be skipped in next hop selection by all nodes routing the DELETE message (default value *false*)
registerRoute	**Boolean**	Indicates whether the request should be routed using a registered route (in which case, the response would be sent back along the same path). The default value is *false*.
anonymousRoute	**Boolean**	Indicates whether the request should be routed anonymously (forced anonymous route, "Anonymous route" header option set to *true*). The default value is *false*. If the value is *true*, every node along the route should replace the message's original sender node ID and network address with its own node ID and network address. Additionally the option forces concealing the "TTL" and the "Hop count" header fields, and modifies the initial Steinhaus point if the sending node ID is the most distant nodes to the recipient in the hierarchical hypercube (the new value is the second best next hop found in the routing tables). This option can be set to *true* together with the "registerRoute" option, in which case, the message will be routed along a registered route, concealing the "TTL", "Hop count" and "Steinhaus point".

Table 1.34 HyCubeRoutingDHTManager runtime DELETE parameters

The *HyCubeRoutingDHTManager* module defines three nested modules:

- *DHTStorageManager* - Realizes the local resource storage (implementing interface *DHTSTorageManager*), depending on the implementation, it may store resources in memory, save them to disk or take any other approach. The default implementation (class *HyCubeSimpleDHTStorageManager*) maintains the resources being stored in memory. Table 1.36 specifies the configuration of this module.
- *ResourceAccessController* - Realizes access control mechanism (implementing interface *HyCubeResourceAccessController*) - determines whether nodes are allowed to perform operations on resources. The default implementation (class *HyCubeSimpleResourceAccessController*) allows all operations to be performed by any node. Table 1.37 presents the configuration of this module.
- *ResourceReplicationSpreadManager* - A module determining to how many nodes resources should be replicated, based on the analysis of incomings requests - implementing the interface *HyCubeDHTStorageManager*. The module may be used to increase the replication radius for popular resources. Methods of this module are called when requests are processed, and another two methods return the number of nodes to which the resource should be replicated, or the multiplier - relative to the default value. The default implementation (class *HyCubeSimpleResourceReplicationSpreadManager*) does not process requests, and returns the default number of replication nodes for all resources.

Property	Type	Value
Class	**String**	*net.hycube.dht.HyCubeRoutingDHTManager*
XxxxCallbackEventTypeKey	**String**	The event type key for PUT / REFRESH_PUT / GET / DELETE operation callback events. * Xxxx = Put / RefreshPut / Get / Delete
XxxxRequestTimeoutEventTypeKey	**String**	The event type key for PUT / REFRESH_PUT / GET / DELETE request timeout events. * Xxxx = Put / RefreshPut / Get / Delete
XxxxRequestTimeout	**Integer**	The timeouts for PUT / REFRESH_PUT / GET / DELETE requests * Xxxx = Put / RefreshPut / Get / Delete
ResourceStoreTime	**Integer**	Resource validity time (milliseconds) - after that time resources should be deleted from node's local storage (if not refreshed)
Metric	**Enum**	The metric defining distances between nodes/resource keys (*Metric* enum.)
CheckIfResourceReplicaBeforeStoring	**Boolean**	Determines whether nodes should check (before storing resources) whether they are one of the closest nodes to the resource key
ResourceStoreNodesNum	**Integer**	The number of nodes that should store individual resources (k_{store})
ReplicationNodesNum	**Integer**	Specifies to how many nodes the resources should be replicated (k_{rep})

Replicate	**Boolean**	Determines whether the replication mechanism is enabled
AnonymousReplicate	**Boolean**	Determines whether REPLICATE messages should be sent anonymously ("Anonymous route" header flag set to *true*).
MaxReplicationNSNodesNum	**Integer**	Specifies the maximum number of nodes (from the neighborhood set) to which replication information is sent
MaxReplicationSpreadNodesNum	**Integer**	Specifies the maximum number of replication nodes
AssumeNsOrdered	**Boolean**	If *true* (the node selection algorithm maintains the neighborhood set ordered), efficiency of the DHT manager may be improved.
DensityCalculationQuantileFuncThreshold	**Double**	Determines how many neighborhood set nodes are taken into account in the DHT density calculation
EstimatedDistanceCoefficient	**Double**	Adjusts the probability of nodes accepting resources by multiplying the estimated radius by the parameter value
EstimateDensityBasedOnLastNodeOnly	**Boolean**	Estimates the network density based only on the most distant node from the neighborhood set nodes being considered
XxxxResponseAnonymousRoute	**Boolean**	Determines whether PUT / REFRESH_PUT / GET / DELETE responses should be sent anonymously ("Anonymous route" header flag set to *true*). * Xxxx = Put / RefreshPut / Get / Delete
ReplicationGetRegisterRoute	**Boolean**	Determines whether GET requests initiated as an effect of the replication should be sent using a registered route
ReplicationGetAnonymousRoute	**Boolean**	Determines whether GET requests initiated as an effect of the replication should be sent anonymously ("Anonymous route" header flag set to *true*)
IgnoreExactXxxxRequests	**Boolean**	Determines whether nodes should ignore exact match PUT / REFRESH_PUT / GET / DELETE requests (anonymity) * Xxxx = Put / RefreshPut / Get / Delete
DHTStorageManager	**Nested**	Storage management module (provides the storage functionality)
ResourceAccessController	**Nested**	DHT access controller module - determines whether nodes are allowed to perform operations on resources
ResourceReplicationSpreadManager	**Nested**	A module determining to how many nodes resources should be replicated, based on the analysis of incoming requests

Table 1.35 HyCubeRoutingDHTManager configuration properties

Whenever an anonymous ("Anonymous route" message header option) request (PUT, REFRESH_PUT, GET, DELETE) is received, the response should also be sent anonymously. However, GET requests sent anonymously, if the route is not registered, should be dropped by any node receiving it. Because the original request sender is unknown, it would be not possible to return the result to the sender. For PUT, REFRESH_PUT and DELETE requests sent anonymously without the route being registered, the requests should processed, but no responses would sent, as the original request senders are unknown.

Property	Type	Value
Class	**String**	*net.hycube.dht.HyCubeSimpleDHTStorageManager*
StoreMultipleCopies	**Boolean**	Determines whether multiple copies for the same *resourceId* values should be stored (different *resourceUrl* values)
MaxResourcesNum	**Integer**	Specifies the maximum number of resources stored by the node
MaxKeySlotSize	**Integer**	Specifies the maximum number of resources stored for any resource key
MaxResourceSlotSize	**Integer**	Specifies the maximum number of resources stored for any unique pair of *resourceId* and *resourceUrl*

Table 1.36 HyCubeSimpleDHTStorageManager configuration properties

Property	Type	Value
Class	**String**	*net.hycube.dht.HyCubeSimpleResourceAccessController*

Table 1.37 HyCubeSimpleResourceAccessController configuration properties

Property	Type	Value
Class	**String**	*net.hycube.dht.HyCubeSimpleResourceReplicationSpreadManager*

Table 1.38 HyCubeSimpleResourceReplicationSpreadManager configuration properties

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeRecoveryExtension*
RecoveryManager	**Nested**	Recovery manager module

Table 1.39 HyCubeRecoveryExtension configuration properties

1.3.19. Recovery manager

The recovery manager is a module responsible for providing recovery functionality in *HyCube*. The recovery manager is not directly attached to the node instance, but uses the extension mechanism. The extension *HyCubeRecoveryExtension* is defined, which maintains the recovery manager module (*HyCubeRecoveryManager*) within it. The recovery manager implements methods initiating recovery, neighborhood recovery and ID recovery procedures (ID recovery is a recovery based on sending recovery requests to closest nodes to a given ID), as well as methods processing received recovery requests and responses, called by received message processors (Section 1.3.20). The recovery extension configuration (Table 1.39) consists only of the configuration of the nested recovery manager - *HyCubeRecoveryManager* (Table 1.40). The "Class" property is not required for the *HyCubeRecoveryManager* module, because the extension always creates an instance of *HyCubeRecoveryManager* class.

HyCubeRecoveryExtension defines a method returning an entry point - an instance of class *HyCubeRecoveryExtensionEntryPoint*, which may be used to invoke the recovery mechanism from outside the library.

In addition to the recovery manager module, a background process (class *HyCubeRecoveryBackgroundProcess*), directly bound to the recovery manager, was created. The background process runs the recovery procedure (or neighborhood set recovery procedure) based on the recovery plan. Table 1.41 presents the configuration properties of *HyCubeRecoveryBackgroundProcess*.

1.3.20. Received message processors, message send processors

Received message processors (classes implementing the interface *ReceivedMessageProcessor*) are modules processing messages received by a node. When a message is received, the node calls *processMessage* method of each defined message processor. The method returns a boolean value indicating whether the message should be passed to the next message processor, or further processing should be abandoned.

Property	Type	Value
SendRecoveryToNS / SendRecoveryToRT1 / SendRecoveryToRT2	**Boolean**	Determines whether recovery (full recovery) requests should be sent to the nodes in the neighborhood set / primary routing table / secondary routing table
SendNotifyToNS / SendNotifyToRT1 / SendNotifyToRT2	**Boolean**	Determines whether NOTIFY messages should be sent to the nodes in the neighborhood set / primary routing table / secondary routing table (all recovery procedure variants)
ProcessRecoveryAsNotify	**Boolean**	Determines whether RECOVERY requests should be also processed as notifications (NOTIFY messages)
ReturnNS / ReturnRT1 / ReturnRT1	**Boolean**	Determines whether nodes should return their neighborhood set / primary routing table / secondary routing table references in RECOVERY_REPLY messages (full recovery)
RecoveryNSReturnNS / RecoveryNSReturnRT1 / RecoveryNSReturnRT2	**Boolean**	Determines whether nodes should return their neighborhood set / primary routing table / secondary routing table references in RECOVERY_REPLY messages (neighborhood set recovery)
RecoveryIdReturnNS / RecoveryIdReturnRT1 / RecoveryIdReturnRT2	**Boolean**	Determines whether nodes should return their neighborhood set / primary routing table / secondary routing table references in RECOVERY_REPLY messages (ID recovery)
SendNotifyToRecoveryReplyNodes	**Boolean**	Determines whether a NOTIFY message should be sent to nodes to which references were received in RECOVERY_REPLY messages
RecoveryEventKey	**String**	Specifies the recovery event key
RecoveryNodesMax	**Integer**	If greater than 0, specifies the maximum number of nodes to which the recovery requests are sent (all neighborhood set nodes + remaining number of random primary and routing table nodes)
MinRecoveryInterval	**Integer**	Specifies the minimum interval between recovery procedure runs (ms)
MinNotifyNodeInterval	**Integer**	Specifies the minimum interval between sending NOTIFY messages to the same node
NotifyNodesCacheMaxSize	**Integer**	Specifies the maximum number of node notification times stored
NotifyNodesMax	**Integer**	If greater than 0, specifies the maximum number of nodes to which the NOTIFY messages are sent (all neighborhood set nodes + remaining number of random primary and routing table nodes - considering only nodes that were not recently notified (MinNotifyNodeInterval))
NotifyRecoveryReplyNodesMax	**Integer**	If SendNotifyToRecoveryReplyNodes is *true*, the parameter specifies the max. number of such nodes to which notifications are sent - random selection (0 indicates that NOTIFY should be sent to all nodes)
RecoveryReplyNodesToProcessMax	**Integer**	If greater than 0, specifies the maximum number of references received within a RECOVERY_REPLY message that should be processed - according to the sequence in the message
RecoveryReplyNodesMax	**Integer**	If greater than 0, specifies the maximum number of references returned by nodes within RECOVERY_REPLY messages (all neighborhood set nodes + remaining number of random primary and routing table nodes)

Table 1.40 HyCubeRecoveryManager configuration properties

Property	Type	Value
Class	**String**	*net.hycube.dht.HyCubeDHTBackgroundProcess*
ScheduleImmediately	**Boolean**	Determines whether the background process should be immediately scheduled after initialization
RecoveryExtensionKey	**String**	Recovery extension key (allow binding the recovery background process to the recovery extension/manager)
EventTypeKey	**String**	The recovery background process event type key
ScheduleInterval	**Integer**	Background process schedule interval (milliseconds)
SchedulePlan	**List (Enum)**	The recovery plan - a list of recovery types (*HyCubeRecoveryType* enumeration), determining a sequence of recovery variants (for individual recovery runs, successive recovery types from the recovery plan are performed - following a cyclical pattern). The values allowed are: FULL_RECOVERY, RECOVERY_NS.

Table 1.41 HyCubeRecoveryBackgroundProcess configuration properties

Message send processors (classes implementing the interface *MessageSendProcessor*) are modules processing messages before sending. Before the message is passed to the routing manager and sent, the method *processSendMessage* of each processor is called. The method expects an argument of type *MessageSendProcessInfo* containing the message object, the direct recipient of the message, and a flag indicating whether the message should be processed by message send processors. The method also returns a boolean value indicating whether further processing should be continued. If any of the processors returns the value *false*, the message eventually will not be sent.

If no message send processors are defined, the messages being sent will be immediately sent without any additional processing. However, if no received message processors are defined, received messages will not be processed at all (dropped) - the logic is based on received message processors, which may then pass the message (or the information received within the message) to individual modules implementing further processing.

Two message processors - *HyCubeReceivedMessageProcessor* (received message processor - implementing *ReceivedMessageProcessor*) and *HyCubeMessageSendProcessor* (message send processor - implementing *MessageSendProcessor*) allow dispatching messages to nested message processors based on message types (recursive definition of message processors). *HyCubeReceivedMessageProcessor* additionally allows defining maximum numbers of messages processed in a specified time interval (limiting the total number of messages processed and limiting the numbers of messages for individual message types).

Property	Type	Value
Class	**String**	*net.hycube.messaging.processing ↪.HyCubeReceivedMessageProcessor*
MessageTypes	**List (Enum)**	Specifies the message types (*HyCubeMessageType* enum.) that should be processed by this message processor. For a top-level message processor, the list should include all message types.
LimitMaxProcessedMessagesRate.Num, **LimitMaxProcessedMessagesRate.Time**	**Integer**	The maximum number of messages processed within the specified time interval (ms)
LimitMaxProcessedMessagesRate.LimitForTypes	**List (Enum)**	A list of message types (*HyCubeMessageType* enum.) for which type-level limits should be applied. For individual message types, limits are defined by parameters: "LimitMaxProcessedMessagesRate[TYPE].Num" and "LimitMaxProcessedMessagesRate[TYPE].Time"
LimitMaxProcessedMessagesRate[TYPE].Num, **LimitMaxProcessedMessagesRate[TYPE].Time**	**Integer**, **Integer**	The maximum number of messages processed within the specified time interval (ms) for messages of type TYPE (*HyCubeMessageType* enum.)
ProcessRouteBackMessagesByNodesOnRoute	**List (Enum)**	Determines whether messages sent back along a registered route should be processed by the nested message processors, if the processing node is not the registered route start
ReceivedMessageProcessors	**Nested**	A list of nested message processors. For each of these message processor the property "MessageTypes" should be defined, specifying a list of message types that should be processed (*HyCubeMessageType* enumeration).

Table 1.42 HyCubeReceivedMessageProcessor configuration properties

HyCubeReceivedMessageProcessor is also responsible for detecting messages routed back along a registered route and, if necessary, passing them to the routing manager (which handles routing messages back along registered routes). The configuration parameters of *HyCubeReceivedMessageProcessor* are presented in Table 1.42, and the configuration of *HyCubeMessageSendProcessor* is presented in Table 1.43.

Several received message processors and several message send processors have been implemented to process messages of individual types:

- *HyCubeReceivedMessageProcessorData* - Realizes processing of received data messages - inserts the received message to an appropriate received messages queue and, if registered, calls the received message callback object. Depending on the configuration, upon receiving a message, this message processor may also call the ACK manager in order to send an ACK message to the message sender. *HyCubeReceivedMessageProcessorData* additionally

Property	Type	Value
Class	**String**	*net.hycube.messaging.processing.HyCubeMessageSendProcessor*
MessageTypes	**List (Enum)**	Specifies the message types (*HyCubeMessageType* enum.) that should be processed by this message processor. For a top-level message processor, the list should include all message types.
MessageSendProcessors	**Nested**	A list of nested message processors. For each of these message processor the property "MessageTypes" should be defined, specifying a list of message types that should be processed (*HyCubeMessageType* enumeration).

Table 1.43 HyCubeMessageSendProcessor configuration properties

provides detection of message duplicates. Table 1.44 specifies the configuration of this message processor.

- *HyCubeReceivedMessageProcessorAck* - Processes received ACK messages and calls the ACK callback object (if registered upon sending the DATA message). Table 1.45 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorPing* - Processes received PING and PONG messages. Upon receiving a PING message, the message processor sends back a PONG message. When a PONG message is received, the processor updates the node liveness information (ping response indicator) stored in instances of *RoutingTableEntry* class. Table 1.46 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorLookup* - Transfers processing of received LOOKUP and LOOKUP_REPLY messages to the lookup manager module. Table 1.47 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorSearch* - Transfers processing of received SEARCH and SEARCH_REPLY messages to the search manager module. Table 1.48 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorSearchJoin* - Transfers processing of received JOIN and JOIN_REPLY messages to the search join manager module (if the search join technique is used). Table 1.49 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorRouteJoin* - Transfers processing of received JOIN and JOIN_REPLY messages to the route join manager module (if the route join technique is used). Table 1.50 specifies the configuration of this message processor.

- *HyCubeReceivedMessageProcessorRecovery* - Transfers processing of received RECOVERY and RECOVERY_REPLY messages to the recovery manager module. Table 1.51 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorNotify* - Passes the node reference (sender of the received NOTIFY message) to the notify processor module. Table 1.52 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorLeave* - Transfers processing of received LEAVE messages to the leave manager module. Table 1.53 specifies the configuration of this message processor.
- *HyCubeReceivedMessageProcessorDHT* - Transfers processing of received DHT requests/responses to the DHT manager module. Table 1.54 specifies the configuration of this message processor.
- *HyCubeMessageSendProcessorData* - Transfers processing of the DATA message being sent to the ACK manager module, which registers the message in the list of messages awaiting acknowledgment. Table 1.55 specifies the configuration of this message processor.
- *HyCubeMessageSendProcessorPing* - Processes PING messages being sent - registers the PING message in the list of messages awaiting responses (PONG). Table 1.56 specifies the configuration of this message processor.

1.3.21. Message delivery acknowledgments and resending

The interface for message delivery acknowledgment mechanism is provided by the core of the library. The *Node*'s *sendDataMessage* methods take two additional arguments (ACK callback object implementing the interface *MessageAckCallback* and an *Object* - the callback argument passed to the callback method when the ACK message is received). This information, as well as additional information required to resend the message if ACK is not received (timeout), are stored in a special object of type *AckProcessInfo* and passed to the message send processors within an object of type *DataMessageSendProcessInfo* (extending the *MessageSendProcessInfo* class). The message send processor *HyCubeMessageSendProcessorData* is responsible for processing DATA messages being sent (casting the message send info object to *DataMessageSendProcessInfo*). The processors is bound (configuration) to an extension object (ACK extension - *HyCubeAckExtension* class), which maintains the ACK manager module - an instance of

Property	Type	Value
Class	**String**	*net.hycube.messaging.data.HyCubeReceivedMessageProcessorData*
MessageTypes	**List (String)**	DATA
AckEnabled	**Boolean**	Determines whether ACK mechanism is enabled
AckExtensionKey	**String**	ACK extension key
ProcessDataMessageIfCannotRoute	**Boolean**	Determines whether DATA messages should be processed by nodes even if the recipient ID is not reached (but the message cannot be routed any further)
PreventDuplicates	**Boolean**	Determines whether message duplicates should be detected
PreventAnonymousDuplicates	**Boolean**	Determines whether message duplicate detection should be enabled for anonymous messages ("Anonymous route" header option) - every node routing an anonymous message updates the original message sender, which may cause messages sent by different nodes to be detected as duplicates
PreventDuplicatesIncludeCRC	**Boolean**	Determines whether message duplicates detection should also compare the CRC message header field
PreventDuplicatesRetentionPeriod	**Integer**	The time (ms) for which duplicates information is stored
PreventDuplicatesCacheMaxSize	**Integer**	The maximum number of entries (messages received) stored for duplicates detection

Table 1.44 HyCubeReceivedMessageProcessorData configuration properties

Property	Type	Value
Class	**String**	*net.hycube.messaging.ack.HyCubeReceivedMessageProcessorDataAck*
MessageTypes	**List (String)**	DATA_ACK
AckExtensionKey	**String**	ACK extension key

Table 1.45 HyCubeReceivedMessageProcessorAck configuration properties

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeReceivedMessageProcessorPing*
MessageTypes	**List (String)**	PING, PONG
KeepAliveExtensionKey	**String**	Keep-alive extension key

Table 1.46 HyCubeReceivedMessageProcessorPing configuration properties

Property	Type	Value
Class	**String**	*net.hycube.lookup.HyCubeReceivedMessageProcessorLookup*
MessageTypes	**List (String)**	LOOKUP, LOOKUP_REPLY

Table 1.47 HyCubeReceivedMessageProcessorLookup configuration properties

Property	Type	Value
Class	**String**	*net.hycube.search.HyCubeReceivedMessageProcessorSearch*
MessageTypes	**List (String)**	SEARCH, SEARCH_REPLY

Table 1.48 HyCubeReceivedMessageProcessorSearch configuration properties

Property	Type	Value
Class	**String**	*net.hycube.join.searchjoin.HyCubeReceivedMessageProcessorSearchJoin*
MessageTypes	**List (String)**	JOIN, JOIN_REPLY

Table 1.49 HyCubeReceivedMessageProcessorSearchJoin configuration properties

Property	Type	Value
Class	**String**	*net.hycube.join.routejoin.HyCubeReceivedMessageProcessorRouteJoin*
MessageTypes	**List (String)**	JOIN, JOIN_REPLY

Table 1.50 HyCubeReceivedMessageProcessorRouteJoin configuration properties

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeReceivedMessageProcessorRecovery*
MessageTypes	**List (String)**	RECOVERY, RECOVERY_REPLY
RecoveryExtensionKey	**String**	Recovery extension key

Table 1.51 HyCubeReceivedMessageProcessorRecovery configuration properties

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeReceivedMessageProcessorNotify*
MessageTypes	**List (String)**	NOTIFY
ValidateNotifyMessageSender	**Boolean**	Determines whether only LEAVE messages sent directly should be processed

Table 1.52 HyCubeReceivedMessageProcessorNotify configuration properties

Property	Type	Value
Class	**String**	*net.hycube.leave.HyCubeReceivedMessageProcessorLeave*
MessageTypes	**List (String)**	LEAVE
ValidateLeaveMessageSender	**Boolean**	Determines whether only LEAVE messages sent directly should be processed

Table 1.53 HyCubeReceivedMessageProcessorLeave configuration properties

Property	Type	Value
Class	**String**	*net.hycube.dht.HyCubeReceivedMessageProcessorDHT*
MessageTypes	**List (String)**	PUT, PUT_REPLY, GET, GET_REPLY, DELETE, DELETE_REPLY, REFRESH_PUT, REFRESH_PUT_REPLY, REPLICATE

Table 1.54 HyCubeReceivedMessageProcessorDHT configuration properties

Property	Type	Value
Class	**String**	*net.hycube.messaging.ack.HyCubeMessageSendProcessorDataAck*
MessageTypes	**List (String)**	DATA
AckExtensionKey	**String**	ACK extension key

Table 1.55 HyCubeMessageSendProcessorData configuration properties

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeMessageSendProcessorPing*
MessageTypes	**List (String)**	PING
KeepAliveExtensionKey	**String**	Keep-alive extension key

Table 1.56 HyCubeMessageSendProcessorPing configuration properties

Property	Type	Value
Class	**String**	*net.hycube.messaging.ack.HyCubeAckExtension*
AckManager ↪.ApplySecureRoutingAfterNotDeliveredCount	**Integer**	Determines after how many attempts (ACK not received) secure routing should be applied
AckManager ↪.ApplySkippingNextHopsAfterNotDeliveredCount	**Integer**	Determines after how many attempts (ACK not received) a random number of nodes in next hop selection should be skipped by nodes on the route
AckManager.ValidateAckSender	**Boolean**	Determines whether the ACK sender should be validated (to be the same node as the one to which the original message was addressed). The value should be set to *false* when anonymous registered routing is enabled, or when processing of DATA messages by closest nodes, not being the message recipient, is allowed.

Table 1.57 HyCubeAckExtension configuration properties

HyCubeAckManager class. The message processor passes processing of the DATA message being sent to the ACK manager, which registers the message in the list of messages awaiting acknowledgment. When the ACK message (corresponding to the DATA message sent) is received, the entry is removed from the list and the ACK callback object is called (*notifyDelivered* method) to signal the acknowledgment was received. Depending on configuration (on *Node* level), when the ACK is not received within a specified time, the message may be resent (the number of attempts is also configurable). If all resend attempts fail (no ACK is received), the ACK callback is called (*notifyUndelivered* method) to signal that the message delivery failed (was not confirmed). The list of sent messages awaiting acknowledgments is checked (for entries for which the timeout elapsed) by the ACK manager, which is triggered periodically by the ACK background process (*HyCubeAwaitingAcksBackgroundProcess* class). The configuration properties of *HyCubeAckExtension* (incl. nested properties of *HyCubeAckManager*) are presented in Table 1.57, and properties of *HyCubeAwaitingAcksBackgroundProcess* are presented in Table 1.58.

If a DATA message is sent with the "Secure routing" and/or "Skip random next hops" header options set to *true*, the ACK message sent to the DATA message sender has the same values of both header options.

Property	Type	Value
Class	**String**	*net.hycube.messaging.ack.HyCubeAwaitingAcksBackgroundProcess*
ScheduleImmediately	**Boolean**	Determines whether the background process should be scheduled immediately after the initialization
AckExtensionKey	**String**	ACK extension key
EventTypeKey	**String**	The background process event type key
ScheduleInterval	**Integer**	The schedule interval (ms) for the background process

Table 1.58 HyCubeAwaitingAcksBackgroundProcess configuration properties

1.3.22. Keep-alive mechanism

HyCube's keep-alive mechanism (working together with the routing table node selection technique of *HyCube*) is based on node "liveness" (ping response indicator) values (stored in *RoutingTableEntry* objects for individual references). The mechanism is based on exchanging keep-alive messages between nodes. The *HyCubePingBackgroundProcess* background process periodically sends keep-alive (PING) message to neighbors, and nodes receiving these messages are supposed to send keep-alive responses (PONG messages) to the PING sender. When a PING message is sent, the *HyCubeMessageSendProcessorPing* adds that message to the list of PING messages awaiting replies. The liveness values are updated by the received message processor *HyCubeReceivedMessageProcessorPing* upon receiving keep-alive responses (PONG messages) or by the *HyCubeAwaitingPongsBackgroundProcess* background process, when the keep-alive response is not received after a specified timeout (the *HyCubeAwaitingPongsBackgroundProcess* background process runs periodically, checking whether the timeout elapsed for individual PING messages sent). The list of PING messages, for which PONG responses are expected is stored in a common extension object, *HyCubeKeepAliveExtension*, which is accessed by both message processors and background processes. *HyCubeKeepAliveExtension* also stores the common parameters of the keep-alive mechanism, that are accessible for other modules. Furthermore, the *HyCubeKeepAliveExtension* object serves as a cache for the liveness values of nodes that were removed from routing tables (the values are stored for the time specified in the configuration and are used when a removed node is again being considered as a routing table candidate).

The configuration of the message processors (*HyCubeMessageSendProcessorPing* and *HyCubeReceivedMessageProcessorPing*) is presented in Section 1.3.20.

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubePingBackgroundProcess*
ScheduleImmediately	**Boolean**	Determines whether the background process should be scheduled immediately after the initialization
KeepAliveExtensionKey	**String**	Keep-alive extension key
EventTypeKey	**String**	The background process event type key
ScheduleInterval	**Integer**	The schedule interval (ms) for the background process. The value should be a pointer to the "PingInterval" configuration parameter of the *HyCubeKeepAliveExtension* extension module.

Table 1.59 HyCubePingBackgroundProcess configuration properties

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeAwaitingPongsBackgroundProcess*
ScheduleImmediately	**Boolean**	Determines whether the background process should be scheduled immediately after the initialization
KeepAliveExtensionKey	**String**	Keep-alive extension key
EventTypeKey	**String**	The background process event type key
ScheduleInterval	**Integer**	The schedule interval (ms) for the background process. The value should be a pointer to the "ProcessPongInterval" configuration parameter of the *HyCubeKeepAliveExtension* extension module.

Table 1.60 HyCubeAwaitingPongsBackgroundProcess configuration properties

The configuration properties of the *HyCubePingBackgroundProcess* and *HyCubeAwaitingPongsBackgroundProcess* background processes is presented in Tables 1.59 and 1.60, and the configuration of the *HyCubeKeepAliveExtension* module is presented in Table 1.61.

1.3.23. Network adapter and message receiver

The network adapter module represents the the network (transport) layer of the node. A *HyCube* node may operate upon any underlying network protocol, and the *Node*'s protected method *pushMessage*, which is supposed to send messages through the physical network (transport layer), passes the message and the direct recipient pointer to the network adapter module (implementing the *NetworkAdapter* interface) by calling its *sendMessage* method. The network adapter maintains the network interface address as well as the public node network address (if different). The interface address is determined by the address string specified

Property	Type	Value
Class	**String**	*net.hycube.maintenance.HyCubeKeepAliveExtension*
PingInterval	**Integer**	The ping time interval (ms). Determines how often PING messages are sent to neighbors.
PongTimeout	**Integer**	Specifies the keep-alive timeout (ms) - the waiting time for PONG messages, after which the keep-alive is considered failed.
ProcessPongInterval	**Integer**	Specifies how often the *HyCubeAwaitingPongsBackgroundProcess* background process should be triggered.
PingResponseIndicatorRteKey	**String**	Specifies the key under which the ping response indicator value is stored in routing table entries. This value should be the same as the value of the parameter "LnsIndicatorRteKey" of the *HyCubeLnsRTNodeSelector* module, as both modules operate on the same values.
InitialPingResponseIndicatorValue	**Double**	Specifies the initial value of the ping response indicator - L_{init}
MaxPingResponseIndicatorValue	**Double**	Specifies the maximum value of the ping response indicator - L_{max}
PingResponseIndicatorUpdateCoefficient	**Double**	Specifies the ping response indicator update coefficient - p parameter
PingResponseIndicatorDeactivateThreshold	**Double**	Specifies the ping response indicator threshold value under which the node is deactivated (not considered in next hop selection) - $L_{deactivate}$
PingResponseIndicatorReplaceThreshold	**Double**	Specifies the LNS replace threshold - routing table references with values of the LNS indicator lower than this threshold may be replaced - $L_{replace}$
PingResponseIndicatorRemoveThreshold	**Double**	Specifies the ping response indicator threshold value under which the node is removed from the routing table(s) - L_{remove}
PingResponseIndicatorRetentionTime	**Integer**	Specifies the time (ms) for which the values of the ping response indicator are stored after a node is removed from the routing table. The value 0 indicates that the values for removed nodes should not be cached at all.

Table 1.61 HyCubeKeepAliveExtension configuration properties

upon the node initialization, and the public address, initially equal to the interface address, might be discovered during the joining procedure. Both values are set by appropriate modules and their interpretation depends on the network layer used. Three address representations should be supported by the network adapter: a *String* representation, *byte[]* representation and the *NetworkNodePointer* representation - the implementation of this interface is network adapter dependent. The *NetworkAdapter* interface specifies methods converting network addresses from *String* and *byte[]* representations to the *NetworkNodePointer* representation, and conversion from *NetworkNodePointer* to string and binary representations is provided by implementations of *NetworkNodePointer* interface. All three representations may be used by any modules, maintaining the abstraction of the actual network adapter in use.

The *NetworkAdapter* interface declare the method *messageReceived*, which is supposed to be called when a message is received by the node. The method should be called by a separate module - the message receiver (implementing *MessageReceiver* interface), which is responsible for receiving incoming messages and passing them (and the direct sender network address) to the network adapter. Upon receiving a message, the network adapter is supposed to pass the message received and the direct sender to the *Node* instance by calling *Node.messageReceived* method, which would then pass the message to the defined received message processors. The message receiver implementation should be compatible with the network adapter used and may use functionalities implemented by the network adapter.

By design, a single message receiver is able to deliver messages to multiple network adapter instances (multiple node instances served by the same message receiver). The *MessageReceiver* interface specifies methods for registering and unregistering network adapters. The message receiver may listen for messages coming from multiple sources and dispatch them to appropriate network adapters, based for example of the network addresses of individual network adapters (of individual nodes). Thus, the message receiver is not initialized automatically by the *Node* instance, but should be created outside, and the node's network adapter should then be registered for the message receiver. After the message receiver is initialized, it should be explicitly started (by calling the *startMessageReceiver* method) to start receiving messages. The method *receiveMessage* is responsible for receiving messages (possibly a blocking call), and may possibly pass multiple messages to the network adapter at a time (or no messages if no message was received).

The network adapter module may implement message fragmentation and reassembly (splitting messages to smaller messages not exceeding the defined maximum message size). In such a case, the network adapter, before sending the message, should split the message into fragments and send all fragments within a single *sendMessage* call. When the message receiver passes a message fragment to the network adapter, the network adapter is responsible to gather all remaining message fragments, and call the *Node.messageReceived* method only when the complete message is assembled.

Within the *HyCube* library, several network adapter and message receiver implementations, based on the UDP/IP protocol, are delivered. The list below briefly characterizes individual implementations:

- *UDPNetworkAdapter* - A network adapter providing abstraction of the UDP/IP protocol, providing message fragmentation. The implementation is based on the Java Socket API. The socket created is publicly accessible and may be used by message receivers to receive messages. Table 1.62 specifies the configuration of this network adapter.
- *UDPSelectorNetworkAdapter* - A network adapter providing abstraction of the UDP/IP protocol, providing message fragmentation. The implementation is based on the Java NIO API and uses a channel (*DatagramChannel* object) to send messages. The channel is publicly accessible and may be used by a message receiver to receive messages. With the use of a selector (*Selector* object), message receivers are able to listen for messages on multiple channels, serving multiple network adapters. This is the default network adapter implementation used in *HyCube*. Table 1.63 specifies the configuration of this network adapter.
- *UDPMessageReceiver* - A message receiver based on the UDP/IP protocol. The implementation is based on the Java Socket API. The message receiver expects network adapters registered to be instances of *UDPNetworkAdapter* and uses the sockets maintained by them to receive messages. Table 1.64 specifies the configuration of this message receiver.
- *UDPSelectorMessageReceiver* - A message receiver based on the UDP/IP protocol. The implementation is based on the Java NIO API. The message receiver expects network adapters registered to be instances of *UDPSelectorNetworkAdapter* and uses the channels maintained by them to receive messages using the selector mechanism. For all registered network adapters, only one instance of selector is used, which allows listening for

messages on all channels using a single *select()* call (one thread). Table 1.65 specifies the configuration of this message receiver.

- *UDPWakeableSelectorMessageReceiver* - A message receiver based on the UDP/IP protocol. The implementation is based on the Java NIO API. The message receiver expects network adapters registered to be instances of *UDPSelectorNetworkAdapter* and uses the channels maintained by them to receive messages using the selector mechanism. For all registered network adapters, only one instance of selector is used, which allows listening for messages on all channels using a single *select()* call (one thread). *UDPWakeableSelectorMessageReceiver* additionally allows waking-up a blocking call of the *receiveMessage* method when needed (implements the *Wakeable* interface), which is a very useful feature for efficient event processing (details of the "wakeable" objects design in Section 1.4). This is the default message receiver used by *HyCube*. Table 1.66 specifies the configuration of this message receiver.

UDPNetworkAdapter and *UDPSelectorNetworkAdapter* use a nested message fragmenter module (a class implementing the *MessageFragmenter* interface), which is responsible for message fragmentation. The interface provides a method for fragmenting messages, taking a message as an argument and returning an array of messages - fragments, and a method responsible for reassembling message fragments into a complete message, taking a message fragment as an argument and returning the assembled message, if all fragments were received, or *null* if no complete message was assembled (storing the fragments received is managed by the fragmenter). The library provides an implementation of the message fragmenter - the class *HyCubeMessageFragmenter*, which provides simple message fragmentation based on the fragment length specified in the configuration of the module. *HyCubeMessageFragmenter* requires 4 bytes reserved in the message header extension field (2 bytes for the fragment number and 2 bytes for the total number of fragments). The configuration of the module is presented in Table 1.67.

1.4. Event processing

This section discusses the architecture of event processing implemented in the *HyCube* library. A node instance performs many operations as a consequence of certain actions (external triggers like receiving message, explicit execution by other operations, scheduled operations,

Property	Type	Value
Class	**String**	*net.hycube.transport.UDPNetworkAdapter*
OSSendBufferSize	**Integer**	The buffer size allocated by the operating system for sending UDP datagrams
OSReceiveBufferSize	**Integer**	The buffer size allocated by the operating system for receiving UDP datagrams
ReceiveTimeout	**Integer**	The timeout (ms) (maximum blocking time) of the receive operation on the socket
MaxMessageLength	**Integer**	The maximum allowed message length
ThrowWhenMaxMessageLengthExceeded	**Boolean**	Determines whether an exception should be thrown when the length of the message being sent exceeds the defined maximum value, or whether such messages should be silently dropped
FragmentMessages	**Boolean**	Determines whether message fragmentation is enabled
MessageFragmenter	**Nested**	Specifies the configuration of the message fragmenter module (nested)

Table 1.62 UDPNetworkAdapter configuration properties

Property	Type	Value
Class	**String**	*net.hycube.transport.UDPSelectorNetworkAdapter*
OSSendBufferSize	**Integer**	The buffer size allocated by the operating system for sending UDP datagrams
OSReceiveBufferSize	**Integer**	The buffer size allocated by the operating system for receiving UDP datagrams
ReceiveTimeout	**Integer**	The timeout (ms) (maximum blocking time) of the receive operation on the socket
MaxMessageLength	**Integer**	The maximum allowed message length
ThrowWhenMaxMessageLengthExceeded	**Boolean**	Determines whether an exception should be thrown when the length of the message being sent exceeds the defined maximum value, or whether such messages should be silently dropped
FragmentMessages	**Boolean**	Determines whether message fragmentation is enabled
MessageFragmenter	**Nested**	Specifies the configuration of the message fragmenter module (nested)

Table 1.63 UDPSelectorNetworkAdapter configuration properties

Property	Type	Value
Class	**String**	*net.hycube.transport.UDPMessageReceiver*
MessageFactory	**Nested**	The configuration (nested) of the message factory module used by the message receiver to create message objects from message bytes received

Table 1.64 UDPMessageReceiver configuration properties

Property	Type	Value
Class	**String**	*net.hycube.transport.UDPSelectorMessageReceiver*
MessageFactory	**Nested**	The configuration (nested) of the message factory module used by the message receiver to create message objects from message bytes received

Table 1.65 UDPSelectorMessageReceiver configuration properties

Property	Type	Value
Class	**String**	*net.hycube.transport.UDPWakeableSelectorMessageReceiver*
MessageFactory	**Nested**	The configuration (nested) of the message factory module used by the message receiver to create message objects from message bytes received

Table 1.66 UDPWakeableSelectorMessageReceiver configuration properties

Property	Type	Value
Class	**String**	*net.hycube.messaging.fragmentation.HyCubeMessageFragmenter*
HeaderExtensionIndex	**Integer**	The index of the message header extension reserved for the message fragmenter
FragmentLength	**Integer**	The maximum byte length of a single message. If the message length exceeds this value, the message will be split into messages of maximum length of FragmentLength
FragmentsRetentionPeriod	**Integer**	The amount of time (ms) for which received message fragments are stored by the message fragmenter. The complete message might not be reassembled if the time difference between receiving the first and the last fragment exceeds the value of this parameter.
PreventFragmentDuplicates	**Boolean**	Determines whether the message fragments duplicates received should be ignored (*true*) or the fragments received before should be overwritten (*false*)

Table 1.67 HyCubeMessageFragmenter configuration properties

or operations initiated from outside the library by calling the API methods). By design, the operations are not performed in the calling thread, but such events are inserted to event queues - instances of *LinkedBlockingQueue* (defined for individual event types) and processed by the "event processor" (an object implementing the *EventQueueProcessor* interface). The event processor is responsible for processing events from event queues. The event processor should be created independently of the node instance, and the event queues (passed to the *Node* initializer method) should be registered in the event processor object, after which the event processor should be started (*start* method). The event queues may be shared among possible multiple node instances, and a single queue may be used for multiple event types. In most cases (a single node instance running), the events may be successfully served by a single queue.

Individual implementations of event processors may use a single processing thread, multiple threads or adjust the number of threads used depending on the number of events in the queues. Events are represented by objects of class *Event* and the event operation is defined either by overloading the method *Event.process()* in the classes extending *Event* class, or by the process event proxy (an object of type *ProcessEventProxy*) to which processing of the event is passed. The process event proxy is specified for the event instance upon initialization. The *Event.process()* method calls the proxy's *processEvent* method passing the event instance to it - this is the default behavior. Additional objects required for event processing may be stored in the *Event* object by specifying a *Object[] eventArgs* array in the event's constructor, or may be explicitly included in the classes extending the *Event* class. Two additional properties are defined in the *Event* class, that may be used by inserting and processing modules: *timestamp* and *priority*.

Every event has a property *eventType*, which defines to which queue the event is inserted. The event type may also be used by the process event proxy object to determine how the event should be processed. The event type is defined by the *eventType* property of class *EventType*. The *EventType* class defines two properties: *eventCategory* (*EventCategory* enumeration) and *eventTypeKey* (*String*). The event category defines the family of the event types, while event type key specifies the event type among possible multiple event types within the same event category. Upon *Node* initialization, the event queues are specified for individual event types - a *Map<EventType, LinkedBlockingQueue<Event»* objects is passed to the *Node.initialize* method. If for, any event type, the event type key is an empty string, this queue will be the default event queue for all event types for the specified event category. Table 1.68 presents the

event categories supported by the *HyCube* library and classes extending *Event* class specific for individual events.

Event of types: *processReceivedMessageEvent*, *pushMessageEvent*, *pushSystemMessageEvent*, *processAckCallbackEvent*, *processMsgReceivedCallbackEvent* are created by the *Node* instance and a special process event proxy class (*Node.NodeProcessEventProxy*) was created to pass processing these types of events back to the *Node* instance, which is responsible for further processing.

Event scheduler

In many situations, instead of processing an event immediately by the event processor, it would be necessary to postpone executing an event - to schedule its future execution. The *Node.initialize* method expects an argument of type *EventScheduler*, which defines the event scheduler object used for scheduling events by the node and all modules. The interface specifies three methods allowing scheduling an event to be executed at the defined time or with a certain delay. The scheduler implementations should insert the specified *Event* object to the specified queue at the time defined in the method call. The class *TimeProviderEventScheduler* is an event scheduler implementation that relies on the scheduling methods of a time provider (*TimeProvider* instance), specified in the constructor. The *DirectEnvironment* environment (default) defines an event scheduler based on the system clock (*ScheduledThreadPoolExecutor*).

Message receiver events

receiveMessageEvent is a special event category. Events of this category call the message receiver's *receiveMessage* method and should be explicitly inserted to the appropriate queue, after the message receiver is properly initialized and the network adapter(s) are already bound to the message receiver. The *MessageReceiver* interface contains a method *startMessageReceiver* and an overloaded version taking an integer argument *numEventsToEnqueue*. The no-argument method implementations should insert the receiver event to the message receiver queue, while the integer argument of the second method specifies the number of the events that should be inserted to the queue - for certain message receivers it is possible to define multiple threads processing an event queue, in which case multiple events may be processed at the same time. However, the message receiver should, in such a case, be aware of the possibility of multiple simultaneous calls of the *receiveMessage* method.

Event category	Description
undefinedEvent	Undefined event type. May be used in certain situations when the event type is not relevant.
receiveMessageEvent	An event executing the message receiver call event. Class *MessageReceiverProcessEventProxy* should be used to process events of this type. The implementation's *processEvent* methods calls the *receiveMessage* method of the message received specified in the constructor of *MessageReceiverProcessEventProxy*. The events of this category are represented by *Event* class instance (do not require any additional parameters).
processReceivedMessageEvent	An event processing a received message. Class *ProcessReceivedMessageEvent* should be used to represent events of this event category.
pushMessageEvent	An event pushing a message to the network layer (physically sending the message). Class *PushMessageEvent* should be used to represent events of this event category.
pushSystemMessageEvent	An event pushing a system message to the network (transport) layer, physically sending the message. Class *PushMessageEvent* should be used to represent events of this event category.
processAckCallbackEvent	An event executing received ACK callbacks. Class *MessageAckCallbackEvent* should be used to represent events of this event category.
processMsgReceivedCallbackEvent	An event executing received message callbacks. Class *MessageReceivedCallbackEvent* should be used to represent events of this event type.
executeBackgroundProcessEvent	An event running a background process. The event type key specifies the event types for individual background processes. Class *Event* is used to represent events of this event category, and *BackgroundProcessEventProxy* is the process event proxy class processing events of this category. The background process object is bound to the process event proxy on initialization (constructor argument). The *Event*'s property *eventArgs[0]* specifies whether the *process* or *schedule* method of the background process should be called (*BackgroundProcessEventProxy.BackgroundProcessEventProxyOperation* enumeration).
extEvent	Extended (custom) event - may be any module specific events. The event type key specifies the event type. Modules defining custom event types, may use any arbitrarily chosen class to represent events (extending *Event*) and any process event proxy implementations.

Table 1.68 Event categories

The message receivers should ensure that, after the *receiveMessage* returns, another message receiver event is inserted to the queue, so that new messages may be received. That is why, as a design rule, the implementations of the *receiveMessage* should enqueue the message receiver event after receiving a message, which would ensure that the number of events processed and the events queued, specified in the *startMessageReceiver* call, remains the same.

Wakeable events and notifying queues

As it was already mentioned, the message receiver event (calling *MessageReceiver.receiveMessage* method) may be blocking, in which case the processing thread would not be possible to process events waiting in the queue. In such a case, it would be useful to allow the blocking calls, but provide a mechanism of "waking up" the blocking operation, and allow processing enqueued events, however, inserting another message receiver event to the queue. Similar situations may occur for any custom blocking event type. In the *HyCube* library, the *Wakeable* interface has been introduced, which defines one method *wakeup()*. When called, the object is expected to interrupt the blocking operation being performed, or, if the operation is not yet being performed (for instance, the event is still in the event queue), prevent the operation from blocking when it is executed. The implementation is module specific, but the event processors (Section 1.4) and node services (Section 1.5) expect the described behavior of wakeable events, and the implementations should take into account the possibility of multiple simultaneous *wakeup* calls, if performing the blocking operation simultaneously by multiple threads is allowed. The *UDPWakeableSelectorMessageReceiver* is an example of a wakeable receiver, for which it is possible to wake up a blocking *select* operation by calling the *wakeup* method (or perform it in the non-blocking mode) if the *wakeup()* method was called before the selection.

In order to work with possible multiple blocking and non-blocking events in the queue, the wakeable instances should be managed by a so called "wakeable manager" object, implementing the *WakeableManager* interface. An instance of a wakeable manager is expected to be provided upon creation of any wakeable message receiver (implementing *WakeableMessageReceiver* interface). The wakeable manager object is responsible for managing multiple blocking operations, allowing registering new running operations and removing them after the blocking operation is finished, as well as waking up one of the wakeable objects (and removing them from the list of managed wakeable objects) when the *wakeup* method is called explicitly.

Calls to the wakeable manager methods (*addWakeable*, *removeWakeable*, *wakeup*) and operations performed by the wakeable objects (adding events to the queue, waking up, internal wakeable object state changes) should be atomic. To make sure that no race conditions exist, *WakeableManager* exposes a lock object that is used to synchronize the operations the *addWakeable*, *removeWakeable*, *getWakeables* and *wakeup* operations. This lock may be used to demarcate critical sections including operations on the queues.

The implementation of a wakeable manager provided, the class *EventQueueWakeableManager*, is capable of managing wakeable objects (blocking operations) for an event queue processed by a given number of threads (*availableSlots* constructor argument). When the blocking events are processed, they should be registered as wakeable objects within the wakeable manager, and removed when the processing is finished. The *wakeup* call is interpreted as inserting a non-blocking event to the event queue. The wakeable manager ensures that no non-blocking event processing would have to be delayed because of a blocking operation taking place. The manager registers the numbers of non-blocking events enqueued (*wakeup* calls) after every wakeable event (*addWakeable* call), and when a wakeable event is removed, all non-blocking events enqueued after it (but before the next wakeable event), are also removed. Thus, upon every *wakeup* call, it is able to determine the total number of events currently enqueued for processing. The *wakeup* call would wake up an appropriate number of wakeable objects to ensure that processing of the event would not be delayed by any blocking (wakeable) operation, taking into account the number of available processing threads - *availableSlots*. If there is a certain number of blocking (wakeable) events that should be rescheduled every time after processing the event, the number of threads processing the queue should be at lest equal to the number of these blocking events, to prevent delays in processing blocking events.

The “wakeup” mechanism needs to be integrated with the operations on the event queue, to allow automatic calls to the wakeable manager when the events are inserted to the event queue. For that purpose, the *NotifyingQueue<T>* and *NotifyingBlockingQueue<T>* interfaces has been created, as well as two implementations: *NotifyingLinkedBlockingQueue<T>* and *NotifyingPriorityBlockingQueue<T>* (extending *LinkedBlockingQueue<T>* and *PriorityBlockingQueue<T>*), to be used as event queues. The classes implementing *NotifyingQueue* interface are supposed to call all registered listener objects (*NotifyingQueueListener<T>*) every time an element is inserted to the queue, by

calling the *itemInserted* or *itemsInserted* method (when multiple elements are inserted at a time), passing the items inserted to the method call. The methods inserting elements to the queue have also overloaded versions taking a boolean argument *notify*, determining whether the listener objects should be called on insertions, allowing changing the default notification behavior. This arguments determines whether the listeners should be called after the insertion. *NotifyingLinkedBlockingQueue<T>* and *NotifyingPriorityBlockingQueue<T>* extend the *LinkedBlockingQueue<T>* and *PriorityBlockingQueue<T>* classes and intercept the operations inserting elements to the queue, executing the listener method after the element is inserted. All operations on such queues should be synchronized (operations on the queue and the listener calls) - no race conditions should be present.

To link a notifying queue to an *EventQueueWakeableManager* instance, an instance of *WakeableManagerQueueListener<T>* class may be used, which is supposed to be registered as the notifying queue listener, and would call the wakeup manager's *wakeup* method whenever an element is added to the queue. When a blocking (wakeable) event is inserted to the queue, the *EventQueueWakeableManager* implementation expects the queue NOT to notify the wakeable manager. Therefore, the overloaded notifying queue's method (with the argument *notify=false*), should be called in such cases. The wakeable object should be registered using the *addWakeable* method call upon inserting events to the queue. To allow proper synchronization of *wakeup* and *addWakeable* operations with the operations performed by the notifying queue, the wakeable manager and the wakeable object, all these objects should perform synchronization using the lock object provided by the wakeable manager (the lock object of the notifying queue is defined in the queue object constructor, and the wakeable manager object is passed to the wakeable message receiver's initialize method).

Wakeable managers may also define the maximum time for which a wakeable object is allowed to block in the next call (implementation of the *getNextMaxSleepTime* method), which might be useful when the wakeable manager cooperates with the event scheduler - it may, for example, limit the blocking time to the time left to the execution of the earliest scheduled event. In such a case, no additional scheduler thread would be needed to process the event queue (to wake up the blocking operation when the scheduled event should be inserted to the event queue). With the use of such a mechanism, it is possible to process the event queue (including blocking events) as well as scheduled events, using a single thread.

Event queue processors

Event processors are objects processing events from defined event queues. The arguments of event processor initializer methods define the queues they should operate on, parameters specifying how individual queues should be processed, as well as an error callback object and the error callback argument - the error callback will be called when processing an event throws an exception, passing the callback argument to the callback method. Such a mechanism is necessary to allow processing exceptions thrown while processing events, because the events are not processed in the API caller's thread. Such exceptions should, however, be thrown outside the event processing only when they denote critical errors. Exceptions that are not critical should be handled by the event processing logic. In case the error callback object is notified, the node instance should be immediately terminated.

Several event queue processors have been implemented in the *HyCube* library. Individual implementations differ in the way they manage threads processing event queues:

- *SimpleThreadEventQueueProcessor* - A simple implementation of an event queue processor, employing a certain number of threads (*Thread* objects) processing events from the defined queues (the numbers of threads are defined for individual queues).
- *SimpleThreadPoolEventQueueProcessor* - A simple implementation of an event queue processor, employing a thread pool (using fixed number of threads) for each of the defined queues (the numbers of threads are defined for individual queues).
- *ThreadPoolEventQueueProcessor* - An implementation of an event queue processor, employing a thread pool (using a certain number of threads) for each of the defined queues. The implementation dynamically adjusts the number of threads employed for processing individual queues. The maximum number of threads, as well as the keep alive time is defined for every thread pool. The keep alive time is the maximum time that excess idle threads will wait for new tasks before terminating. Parameters of thread pools processing individual queues are defined by providing objects of type *ThreadPoolInfo* passed to the processor's *initialize* method.
- *EventQueueSchedulerProcessor* - An implementation of an event queue processor, employing a thread pool (using a certain number of threads) for each of the defined queues. The implementation dynamically adjusts the number of threads employed for processing individual queues. The maximum number of threads, as well as the keep alive time is defined for every thread pool. The keep alive time is the maximum time that excess idle

threads will wait for new tasks before terminating. *EventQueueSchedulerProcessor* class also implements the *EventScheduler* interface and provides event scheduler functionality. Parameters of thread pools processing individual queues are defined by providing objects of type *EventQueueProcessingInfo* passed to the processor's *initialize* method. The *EventQueueProcessingInfo* objects specifies the thread pool parameters (*ThreadPoolInfo*), the list of event types expected in the queue, and a boolean *wakeable* flag, specifying if the wakeup mechanism should be applied for processing the queue (in which case the queue is expected to be an instance of *NotifyingQueue*). *EventQueueSchedulerProcessor* class defines the (internal) class *EventQueueProcessorRunnable* (implementing *Runnable* interface). Instances of the internal class *EventQueueProcessorRunnable* are run by threads processing individual queues, retrieving and processing events from the queue. The class *EventQueueProcessorRunnable* additionally implements the *WakeableManager* interface, and its instances serve as wakeable managers for individual queues. The functionality of the wakeable manager described in Section 1.4 is extended by providing the maximum blocking time for wakeable objects, determined based on the number of the processing threads, scheduled event execution times, and planned wakeup times of the events currently being performed. The time returned ensures that, for every scheduled event *e* within k first scheduled events ($k = availableSlots$), the number of threads that would be available for scheduled execution (not sleeping/blocking at the scheduled execution time) is not less than the number of events scheduled, up to the event *e*. That would ensure that first k scheduled events (earliest) can be executed in parallel (by the maximum number of available threads). The same calculations are performed to determine the maximum blocking time of the *BlockingQueue.poll* method call on the event queue object. *EventQueueSchedulerProcessor* takes into account the possibility of scheduling new events for execution in the future while the processing threads are already performing blocking operations or sleeping (waiting for the events to be inserted to the event queue). Upon scheduling a new event, if needed, an appropriate number of threads are woken to allow processing the event in time. Such an approach ensures that no additional scheduler thread is needed to process the event queue (to wake up blocking operations at the time when the scheduled event should be inserted to the event queue). This implementation is the only one provided that allows processing all the events by a single thread. Although other event processors (presented before) may be configured to use a single processing

thread, additional thread(s) would have to be used to insert scheduled events to the event queues (for example the *ScheduledThreadPoolExecutor* instance used by the scheduling functionality of *SystemTimeProvider*). There is, however, one important restriction on the *EventQueueSchedulerProcessor* event processor. Because it relies on specifying maximum blocking times for blocking operations, which are passed to Java built-in classes that interpret the time as the system clock time (*BlockingQueue.poll*, *selector.select*, ...), the time provider specified for the node instance (within the *Environment* instance passed to the node initializer) should be based on the system time (possibly shifted, but the time values returned should grow proportionally to the time values returned by *System.currentTimeMillis()* call).

1.5. Node services

To run a node instance, the programmer is supposed to create and initialize the environment, time provider, scheduler, message receiver and event processor instances before creating the *Node* instance. To simplify the process of creating a node, special classes - node services have been created. Node services are classes responsible for creating the node and the necessary supporting objects, based on the parameters specific to individual node service implementations. In most cases, the node services should be considered as the library API, unless certain customizations, not supported by the implementations provided, are supposed to be made.

The *NodeService* interface exposes operations that may be performed in the node context by the API programmers. The methods directly relate to the methods exposed by the *Node* class. All node services (running a single node) implement this interface. The interface specifies only methods performing generic operations, defined by the core of the library. *HyCube*-specific operations (operations realized by *HyCube*-specific modules) are defined in the interface *HyCubeNodeService*, extending *NodeService*. Two important classes implementing these interfaces are provided: *NodeProxyService* and *HyCubeNodeProxyService*. *NodeProxyService* (implementing *NodeService*) is a proxy service passing the method calls to the corresponding *Node* instance. *HyCubeNodeProxyService* extends *NodeProxyService* with the methods specified in the *HyCubeNodeService* interface - overloaded methods based on *HyCube*-specific types, as well as methods calling entry points of *HyCube*-specific modules.

Property	Type	Value
NodeProxyService	**String**	**HyCubeNodeProxyService** - Determines the key for the nested configuration of the node proxy service.
NodeProxyService[HyCubeNodeProxyService]↪.RecoveryExtensionKey	**String**	Recovery extension key

Table 1.69 HyCubeNodeProxyService configuration properties

HyCubeNodeProxyService should be configured in the configuration file by specifying nested node proxy service configuration - property *NodeProxyService* at the root level of the configuration name space. Table 1.69 specifies the configuration of *HyCubeNodeProxyService*.

The library provides a number of abstract node service classes that implement managing the message receiver, event queues and the event processor, and methods provided by the *NodeService* interface. Internally, they use an instance of *NodeProxyService* to pass requests to the *Node* instance. However, the abstract classes do not define the creation of the node proxy service object. They use an object returned by the abstract method *initializeNodeProxyService*. The implementation of this method in *HyCube*-specific classes creates and returns an object of class *HyCubeNodeProxyService* (which initializes the *Node* instance). Individual node services provide processing events using different event processors and message receivers. The message receiver configuration is read from the configuration file, and the event processor is configured either by specifying the parameters as arguments passed to the node service's *initialize* method call, or by specifying the values in the configuration file, in which case the *initializeFromConf* method should be used to create the service instance. In the configuration file, the node services are configured by nested properties of the property *NodeService* (*NodeService[NodeServiceKey]*) at the root level of the configuration name space, and the *NodeServiceKey* is passed to the *initializeFromConf* method of the service.

The listing below presents the *HyCube* node services provided by the library:

- ***SingleQueueNodeService*** (abstract)

  HyCubeSingleQueueNodeService (*HyCube*-specific)

  A node service using a single event queue to process all event types by a thread pool defined in a *ThreadPoolInfo* object passed to the *initialize* method or in the configuration file (configuration specified in Table 1.70). The wakeup mechanism is used to wake up

wakeable events (with the default *HyCube* configuration, only message receiver events). The message receiver is expected to be an instance of *WakeableMessageReceiver*.

- ***MultiQueueNodeService*** (abstract)

  HyCubeMultiQueueNodeService (*HyCube*-specific)

  A node service using multiple event queues to process events. The parameters and lists of event types for individual queues are defined by an *EventQueueProcessingInfo[]* array passed to the *initialize* method or in the configuration file (configuration specified in Table 1.71). The wakeup mechanism is used to wake up wakeable events (with the default *HyCube* configuration, only message receiver events). The message receiver is expected to be an instance of *WakeableMessageReceiver*.

- ***SchedulingMultiQueueNodeService*** (abstract)

  HyCubeSchedulingMultiQueueNodeService (*HyCube*-specific)

  A node service using multiple event queues to process events. The parameters and lists of event types for individual queues are defined by an *EventQueueProcessingInfo[]* array passed to the *initialize* method or in the configuration file (configuration specified in Table 1.72). Internally, *EventQueueSchedulerProcessor* is used as the event processor and the event scheduler. The wakeup mechanism is used to wake up wakeable events. The message receiver is expected to be an instance of *WakeableMessageReceiver*.

- ***SingleQueueNodeServiceNonWakeable*** (abstract)

  HyCubeSingleQueueNodeServiceNonWakeable (*HyCube*-specific)

  A node service using a single event queue to process all event types by a thread pool defined by a *ThreadPoolInfo* object passed to the *initialize* method or in the configuration file (configuration specified in Table 1.73). The wakeup mechanism is not used by this node service. Therefore, the number of threads serving queues that may contain blocking events (with the default *HyCube* configuration, only message receiver events) should be large enough to prevent blocking operations from causing delays in processing non-blocking events.

- ***MultiQueueNodeServiceNonWakeable*** (abstract)

  HyCubeMultiQueueNodeServiceNonWakeable (*HyCube*-specific)

  A node service using multiple event queues to process events. The parameters and lists of event types for individual queues are defined by an *EventQueueProcessingInfo[]* array passed to the *initialize* method or in the configuration file (configuration specified in Table

1.74). The wakeup mechanism is not used by this node service. Therefore, the number of threads serving queues that may contain blocking events (with the default *HyCube* configuration, only message receiver events) should be large enough to prevent blocking operations from causing delays in processing non-blocking events.

- ***SimpleNodeService*** (abstract)

  HyCubeSimpleNodeService (*HyCube*-specific)

  A node service using one or two event queues to process events. The node service creates the minimum number of threads required to prevent blocking events from delaying processing of other events. The parameters passed to the *initialize* method, or specified in the configuration file (configuration presented in Table 1.75), define the number of custom blocking events (*blockingExtEventsNum*) that may be enqueued at the same time (in addition to the message receiver event - with the default configuration, there are no additional blocking events), and a flag (*wakeupMessageReceiver*) determining whether the wakeup mechanism should be used to wake the message receiver. If *wakeupMessageReceiver* is set to *true*, the node service creates one event queue and the queue is processed by a thread pool consisting of $blockingExtEventsNum + 1$ threads. If *wakeupMessageReceiver* is set to *false*, two event queues are created: a queue for the message receiver events processed by a single thread, and a queue for other event types, processed by $blockingExtEventsNum + 1$ threads. Because the wakeup mechanism is used to wake up wakeable events, the message receiver is expected to be an instance of *WakeableMessageReceiver*.

- ***SimpleSchedulingNodeService*** (abstract)

  HyCubeSimpleSchedulingNodeService (*HyCube*-specific)

  A node service using one or two event queues to process events. The node service creates the minimum number of threads required to prevent blocking events from delaying processing of other events. The parameters passed to the *initialize* method, or specified in the configuration file (configuration presented in Table 1.76), define the number of custom blocking events (*blockingExtEventsNum*) that may be enqueued at the same time (in addition to the message receiver event - with the default configuration, there are no additional blocking events), and a flag (*wakeup*) determining whether the wakeup mechanism should be used. If *wakeup* is set to *true*, the node service creates one event queue and the queue is processed by a thread pool consisting of $blockingExtEventsNum + 1$

Property	Type	Value
ThreadPool.PoolSize	**Integer**	The maximum number of threads in the thread pool processing the event queue
ThreadPool.KeepAliveTimeSec	**Integer**	The maximum time (s) that excess idle threads will wait for new tasks before terminating

Table 1.70 SingleQueueNodeService configuration properties

Property	Type	Value
Queues	**List (String)**	The list of event queues (names). Individual queues are configured referring to the names specified in the list.
Queues[QueueName].ThreadPool.PoolSize	**Integer**	The maximum number of threads in the thread pool processing the event queue *QueueName*
Queues[QueueName].ThreadPool.KeepAliveTimeSec	**Integer**	The maximum time (s) that excess idle threads will wait for new tasks before terminating
Queues[QueueName].Wakeable	**Boolean**	Determines whether the event queue should be bound to the wakeable manager instance
Queues[QueueName].EventTypes	**List (String)**	Defines the event types (names) that the event queue is defined for. Individual event types (for names) are specified by the nested parameters *EventCategory* and *EventTypeKey*
Queues[QueueName].EventTypes[ET].EventCategory	**Enum**	The event category (*EventCategory* enum.) for the event type *ET*
Queues[QueueName].EventTypes[ET].EventTypeKey	**String**	The event type key for the event type *ET*

Table 1.71 MultiQueueNodeService configuration properties

threads. If *wakeup* is set to *false*, two event queues are created: a queue for the message receiver events processed by a single thread, and a queue for other event types, processed by $blockingExtEventsNum + 1$ threads. Internally, *EventQueueSchedulerProcessor* is used as the event processor and the event scheduler. Because the wakeup mechanism is used to wake up wakeable events, the message receiver is expected to be an instance of *WakeableMessageReceiver*.

1.6. Multiple node services

In addition to the node services running single node instances, the library provides two multiple-node services, which may be used to manage multiple node instances with a common event processor, processing the events of all nodes, and a common message receiver,

Property	Type	Value
Queues	**List (String)**	The list of event queues (names). Individual queues are configured referring to the names specified in the list.
Queues[QueueName].ThreadPool.PoolSize	**Integer**	The maximum number of threads in the thread pool processing the event queue *QueueName*
Queues[QueueName].ThreadPool.KeepAliveTimeSec	**Integer**	The maximum time (s) that excess idle threads will wait for new tasks before terminating
Queues[QueueName].Wakeable	**Boolean**	Determines whether the event queue should be bound to the wakeable manager instance
Queues[QueueName].EventTypes	**List (String)**	Defines the event types (names) that the event queue is defined for. Individual event types (for names) are specified by the nested parameters *EventCategory* and *EventTypeKey*
Queues[QueueName].EventTypes[ET].EventCategory	**Enum**	The event category (*EventCategory* enum.) for the event type *ET*
Queues[QueueName].EventTypes[ET].EventTypeKey	**String**	The event type key for the event type *ET*

Table 1.72 SchedulingMultiQueueNodeService configuration properties

Property	Type	Value
ThreadPool.PoolSize	**Integer**	The maximum number of threads in the thread pool processing the event queue
ThreadPool.KeepAliveTimeSec	**Integer**	The maximum time (s) that excess idle threads will wait for new tasks before terminating

Table 1.73 SingleQueueNodeServiceNonWakeable configuration properties

Property	Type	Value
Queues	**List (String)**	The list of event queues (names). Individual queues are configured referring to the names specified in the list.
Queues[QueueName].ThreadPool.PoolSize	**Integer**	The maximum number of threads in the thread pool processing the event queue *QueueName*
Queues[QueueName].ThreadPool.KeepAliveTimeSec	**Integer**	The maximum time (s) that excess idle threads will wait for new tasks before terminating
Queues[QueueName].Wakeable	**Boolean**	Determines whether the event queue should be bound to the wakeable manager instance
Queues[QueueName].EventTypes	**List (String)**	Defines the event types (names) that the event queue is defined for. Individual event types (for names) are specified by the nested parameters *EventCategory* and *EventTypeKey*
Queues[QueueName].EventTypes[ET].EventCategory	**Enum**	The event category (*EventCategory* enum.) for the event type *ET*
Queues[QueueName].EventTypes[ET].EventTypeKey	**String**	The event type key for the event type *ET*

Table 1.74 MultiQueueNodeServiceNonWakeable configuration properties

Property	Type	Value
BlockingExtEventsNum	**Integer**	The number custom blocking events (*blockingExtEventsNum*) that may be enqueued at the same time in addition to the message receiver event (default value 0). With the default configuration, there are no additional blocking events, and the default value 0 should be used.
Wakeup	**Boolean**	Determining whether the wakeup mechanism should be used

Table 1.75 SimpleNodeService configuration properties

Property	Type	Value
BlockingExtEventsNum	**Integer**	The number custom blocking events (*blockingExtEventsNum*) that may be enqueued at the same time in addition to the message receiver event (default value 0). With the default configuration, there are no additional blocking events, and the default value 0 should be used.
Wakeup	**Boolean**	Determining whether the wakeup mechanism should be used

Table 1.76 SimpleSchedulingNodeService configuration properties

dispatching received messages to individual nodes. Such node services are useful when a single machine runs many node instances (for example for simulation purposes). With the use of a single event processor, large numbers of nodes may be processed by a small number of processing threads, using a single message receiver instance.

Multiple node services implement the *MultipleNodeService* interface, specifying methods initializing message receivers (returning *MessageReceiver*) and initializing node services (returning *NodeService*), as well as methods discarding massage receivers and nodes. The interface *HyCubeMultipleNodeService* extends *MultipleNodeService*, overriding the return type of the methods initializing node services to *HyCube*-specific *HyCubeNodeService* class. The abstract class *AbstractMultipleNodeService* provides implementation of managing the message receiver and node service instances and specifies two abstract methods: *initializeNodeProxyService* which is supposed to create the node proxy service (creating a *Node* instance), and *discard* which is supposed to discard the multiple node service instance (discarding the instances of individual node services). Two abstract classes has been created: *MultiQueueMultipleNodeService* and *SchedulingMultipleNodeService* (extending *AbstractMultipleNodeService*), which implement managing the event queues, managing the event processor and the *discard* method. *MultiQueueMultipleNodeService* internally uses *ThreadPoolEventQueueProcessor*,

Property	Type	Value
Queues	**List (String)**	The list of event queues (names). Individual queues are configured referring to the names specified in the list.
Queues[QueueName].ThreadPool.PoolSize	**Integer**	The maximum number of threads in the thread pool processing the event queue *QueueName*
Queues[QueueName].ThreadPool.KeepAliveTimeSec	**Integer**	The maximum time (s) that excess idle threads will wait for new tasks before terminating
Queues[QueueName].Wakeable	**Boolean**	Determines whether the event queue should be bound to the wakeable manager instance
Queues[QueueName].EventTypes	**List (String)**	Defines the event types (names) that the event queue is defined for. Individual event types (for names) are specified by the nested parameters *EventCategory* and *EventTypeKey*
Queues[QueueName].EventTypes[ET].EventCategory	**Enum**	The event category (*EventCategory* enum.) for the event type *ET*
Queues[QueueName].EventTypes[ET].EventTypeKey	**String**	The event type key for the event type *ET*

Table 1.77 MultiQueueMultipleNodeService configuration properties

while *SchedulingMultipleNodeService* uses *EventQueueSchedulerProcessor*. However, these abstract classes do not define the creation of the node proxy service object. The abstract method *initializeNodeProxyService* is implemented in two classes extending *MultiQueueMultipleNodeService* and *SchedulingMultipleNodeService*: *HyCubeMultiQueueMultipleNodeService* and *HyCubeSchedulingMultipleNodeService* - the implementations of the method return an instance of the *HyCubeNodeService*. These classes may be used for creating multiple instances of *HyCube* nodes, processed by the same event processor, and one (or more) message receivers. The event queues and the event processor are configured by the parameters and lists of event types for individual queues defined in a *EventQueueProcessingInfo[]* array passed to the *initialize* method of the service, or in the configuration file, in which case the method *initializeFromConf* should be used to initialize the service instance - the node service key is passed to the *initializeFromConf* method call. The configuration of *MultiQueueMultipleNodeService* and *SchedulingMultipleNodeService* (*HyCubeMultiQueueMultipleNodeService* and *HyCubeSchedulingMultipleNodeService*) is specified in Tables 1.77 and 1.78. Because the wakeup mechanism is used to wake up wakeable events, the message receiver is expected to be an instance of *WakeableMessageReceiver*.

Property	Type	Value
Queues	**List (String)**	The list of event queues (names). Individual queues are configured referring to the names specified in the list.
Queues[QueueName].ThreadPool.PoolSize	**Integer**	The maximum number of threads in the thread pool processing the event queue *QueueName*
Queues[QueueName].ThreadPool.KeepAliveTimeSec	**Integer**	The maximum time (s) that excess idle threads will wait for new tasks before terminating
Queues[QueueName].Wakeable	**Boolean**	Determines whether the event queue should be bound to the wakeable manager instance
Queues[QueueName].EventTypes	**List (String)**	Defines the event types (names) that the event queue is defined for. Individual event types (for names) are specified by the nested parameters *EventCategory* and *EventTypeKey*
Queues[QueueName].EventTypes[ET].EventCategory	**Enum**	The event category (*EventCategory* enum.) for the event type *ET*
Queues[QueueName].EventTypes[ET].EventTypeKey	**String**	The event type key for the event type *ET*

Table 1.78 SchedulingMultipleNodeService configuration properties

2. Changelog

1.0

- Initial release

1.0.1

- Changed the build automation to Maven
- Integration with Travis CI
- Fixed compatibility with Java 1.7 and 1.6

1.0.2

- Corrected integration with Travis CI and deployment to Maven Central