Clustered Remoting
For Spring Framework (Cluster4Spring)

Reference Documentation

Version 0.85-stable

1 Preface

The "**Cluste**red **R**emoting **For S**pring Framework" (or Cluster4Spring) is a replacement for remoting subsystem included into Spring framework.

While implementation of remoting in Spring is great, it has several limitations that are quite important and must be taken into consideration when building large enterprise-level distributed system.

Briefly, these limitations relate to the point-to-point model of remoting supported by Spring – generally speaking, the client may use only one instance of remote service. It is obvious that having only such a scheme of remoting, it is hard to develop fault-tolerant systems and implement some kinds of load balancing.

Another feature, which is currently missing in remoting subsystem offered by Spring framework, is lack of the ability to dynamically discover remote services.

The main purpose of Cluster4Spring is to extend remoting system of Spring framework and overcome limitations mentioned above.

Currently (in version 0.85), the scope of Cluster4Spring includes only remoting related issues and does not include any functionality that can be used for data replication between various instances of server. However, it is possible that such functionality will be added later, if necessary.

This document highlights the most important issues, which relate to Cluster4Spring. However, we suggest that you will also refer to JavaDoc for Cluster4Spring library as well as to the examples if you need more details.

2 License

3 Copyright Notice

The Cluster4Spring library was developed in SoftAMIS, a Ukraine based software development company specialized on Java and Web development outsourcing services.

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

http://www.soft-amis.com

While several people were involved into development, debugging, testing and writing documentation for Cluster4Spring library, most of Cluster4Spring design and implementation was performed by Andrew Sazonov, senior software architect in SoftAMIS.

Before first public release, Cluster4Spring was used at production stage for more than a year in several enterprise level projects and due to this it is quite stable now.

4 Dependencies

Cluster4Spring heavily relies on Java 5 language features, therefore it requires Java version 1.5 or later to be compiled and executed.

The logging functionality within Cluster4Spring is implemented using Apache Commons Logging (http://jakarta.org/commons/logging) therefore the *commons-logging.jar* must be available during library compilation as well as in Java classpath on runtime.

Cluster4Spring includes ready to use implementation of dynamic services discovering mechanism that is based on DistRegistry library and therefore appropriate binaries for that library (located in *softamis-net.jar*) must be available to compile Cluster4Spring from source. Both source code and compiled binaries for DistRegistry are included into Cluster4Spring distribution.

However, if some application does not use remoting scheme with dynamic services discovering or uses different implementation of distributed services registry than DistRegistry, the *softamis-net.jar* is not required on runtime.

Obviously, Cluster4Spring requires libraries from Spring framework. Cluster4Spring is compatible with Spring 1.2.x and Spring 2.x.

5 Major Features

Clustered Remoting for Spring Framework (Cluster4Spring) represents a replacement of the remoting subsystem of Spring framework and provides possibilities to build more stable and fault tolerant systems with dynamic discovering of remote services.

Briefly, the major features of Cluster4Spring library are:

- Support of one-to-one scheme of remoting (similar to one currently supported by Spring);
- Support of one-to-many scheme of remoting, which assumes that one client selects remote service for invocation from one of predefined locations;
- Support of one-to-many scheme of remoting with dynamic discovering of remote services;
- Several built-in policies for selecting remote service for invocation are included (they are applied if service is available in several locations);
- Built-in functionality for handling remoting exceptions of method invocations that provides ability to re-invoke remote service using different service location automatically;
- Steep learning curve since ideological implementation of Cluster4Spring bears a close resemblance to the implementation of remoting in Spring;
- Non-intrusive for existing applications – it is simply enough to change appropriate remoting-related Spring configuration files to start using Cluster4Spring;
- Provides a convenient way to add custom interceptors both on client and server side;
- Flexible and modular architecture which is ready for further extensions and customizations;
- Library is stable and ready to use;

Current version of Cluster4Spring includes core logic that is agnostic to actual remoting protocol and implementation of remoting via RMI. However, it is planned that later version of Cluster4Spring will include support of additional remoting protocols.

6 Overall figure of remoting

Cluster4Spring was designed to be very similar to the existing remoting subsystem offered by Spring framework to simplify transition and adoption of it.

Due to it, it uses the same scheme as Spring framework (and, in general, any remoting implementation):

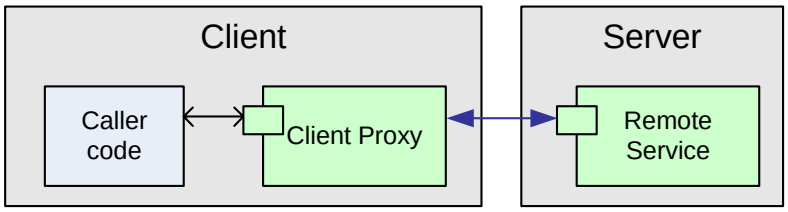

Figure 1 Overall scheme of remoting

There is remote service implementation that resides on the server. On the client side there is a proxy of that remote service which has the same interface as remote service. Caller code invokes methods of proxy and responsibility of proxy implementation is to perform necessary remote communications, to invoke methods of remote service and return results to the caller code.

Cluster4Spring utilizes the same transparent approach to remoting as Spring does – remoting is transparent for caller code and caller code is only aware of remote service interface. Since all magic of remote calls is hidden in proxy, it is possible to use different remoting implementations without affecting application code simply by using various configurations of Spring context.

In general, internally Cluster4Spring remoting includes two layers:

- Remoting core - includes core remoting functionality which is agnostic to particular underlying protocol;
- Protocols layer – contains implementation code that is specific to support of particular remoting protocol.

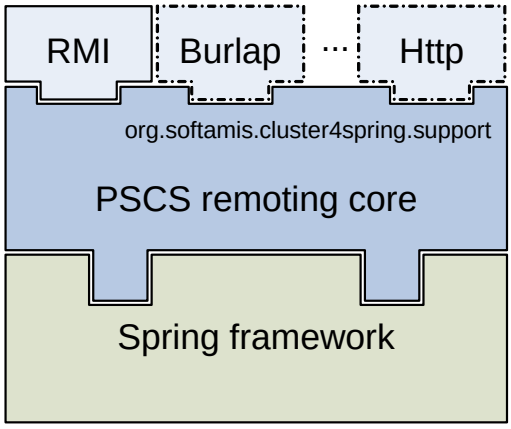

Figure 2 Relation between remoting protocols and Spring core functionality

While current version (0.85) of Cluster4Spring includes core logic and implementation for RMI remoting protocol, additional remoting protocols supported by Spring will be added later.

7 Supported remoting schemes

One of the major features of Cluster4Spring is support of different remoting schemes. These schemes are:

- One-to-one;
- One-to-many static;
- One-to-many with dynamic discovering;

All these schemes are completely transparent for application code and only are defined by the appropriate configuration of remoting-related functionality of application described in Spring configuration files.

Selection of particular remoting scheme depends on the specifics and needs of particular application.

Here we consider possible remoting schemes in more detail.

7.1 One-to-one

This is the simplest scheme and it is the only one supported by remoting subsystem of Spring framework.

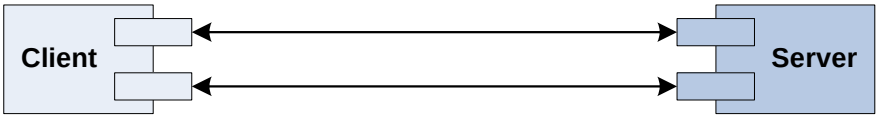

Figure 3 One-to-one remoting scheme

The diagram above illustrates one-to-one remoting scheme. Such scheme assumes that there is one known location of remote service and client can access only that remote instance using that predefined and known service location.

7.1.1 Quick configuration example

This is a quick example of minimal configuration for client-side proxy factory bean for one-to-one remoting scheme if RMI is used as underlying remoting protocol:

```
<bean name="TestService"
      class="org.softamis.cluster4spring.rmi.RmiSingleUrlProxyFactoryBean">
  <property name="serviceInterface" value="org.softamis.cluster4spring.example.Service"/>
  <property name="serviceUrl" value="rmi://localhost:1099/TestService"/>
</bean>
```

It is enough to have Cluster4Spring proxy on client side and use ordinary Spring exporter for remoting service on server to use such scheme of remoting (however, exporter which is included into Cluster4Spring provides extended functionality compared to that of Spring).

7.2 Static one-to-many

This scheme is more complex yet more powerful. It can be used to create distributed systems with high-level fault tolerance and (if appropriate policy for selecting servers is used) to divide the load among several servers.

Such scheme assumes that there is one client and several servers with known locations that are specified as part of remote proxy configuration. Based on appropriate policy, client may select one of these services for actual invocation of remote service methods.

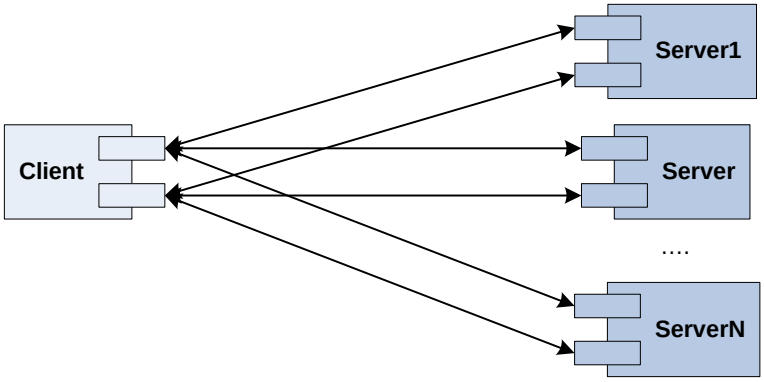

Figure 4 One-to-many remoting scheme with static services

The figure above illustrates static one-to-many remoting scheme.

In addition, if some server becomes inaccessible or some network issues occur while performing remote call to it, the client proxy (optionally, if specified by configuration) may select another instance of remote service (in other words, another server) and re-invoke the corresponding method from it.

7.2.1 Quick configuration example

This is a quick example of the minimal configuration for client-side proxy factory bean for static one-to-many remoting scheme if RMI is used as underlying remoting protocol:

```
<bean name="TestService"
      class="org.softamis.cluster4spring.rmi.RmiUrlListProxyFactoryBean">
  <property name="serviceInterface" value="org.softamis.cluster4spring.example.Service"/>
  <property name="serviceURLs">
     <list>
       <value>rmi://host1:1097/TestService</value>
       <value>rmi://host2:1098/TestService</value>
       <value>rmi://host3:1099/TestService</value>
     </list>
  </property>
</bean>
```

It can be enough to have Cluster4Spring proxy on client side and use ordinary Spring exporter for remoting service on server to use such scheme of remoting.

7.3 One-to-many with dynamic services discovering

In general, such scheme can be considered to be a further evolution of one-to-many scheme.

Exactly as to static one-to-one scheme, such a scheme of remoting also includes one (or more) client that can invoke remote service from one of available locations. However, unlike the previous scheme, available locations of remote service are dynamically discovered.

The figure below illustrates dynamic one-to-many remoting scheme.

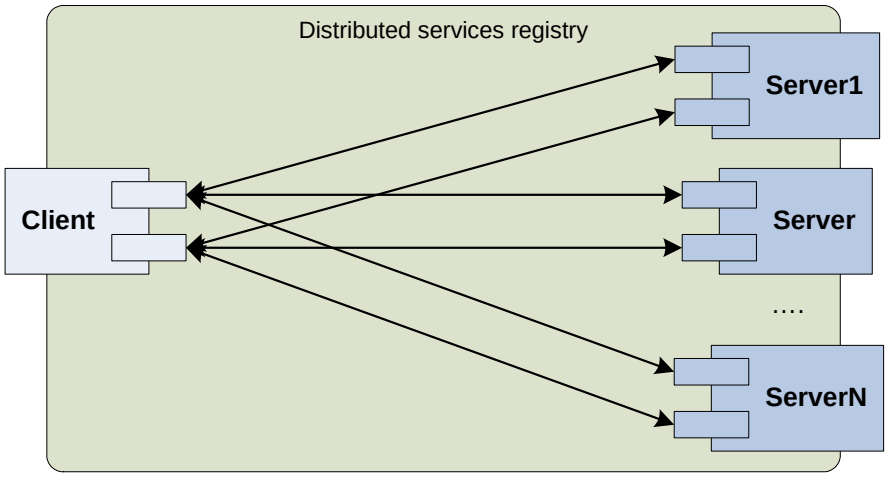

Figure 5 One-to-many remoting with dynamic services discovering

Dynamic discovering of services is performed with the use of appropriate distribute registry which contains information on all locations of particular remote service. To let the service be dynamically discovered, server publishes information on it in distributed registry, so the client is able to access that registry and locate all instances of remote service there.

If such scheme of remoting is used, it is necessary to have Cluster4Spring proxy on client side and exported and publisher on server side (these notions will be described below).

7.3.1 Quick configuration example

This is a quick example of minimal configuration for client-side proxy factory bean for dynamic one-to-many remoting scheme if RMI is used as underlying remoting protocol:

```
<bean name="TestService"
      class="org.softamis.cluster4spring.rmi.RmiDiscoveringProxyFactoryBean">
  <property name="serviceInterface" value="org.softamis.cluster4spring.example.Service"/>
  <property name="serviceName" value="TestService"/>
  <property name="clientServicesRegistry" value="_ClientServicesRegistry"/>
</bean>
```

The "*_ClientServicesRegistry*" is ID of bean that represents client part of distributed registry. Cluster4Spring comes with ready to use implementation of distributed services registry, but, if necessary, it is possible to use custom implementation of such registry.

And this is a sample of server side configuration for exporters and publisher for remote service:

```
<bean class="org.softamis.cluster4spring.rmi.RmiServiceExporter">
  <property name="registry" ref="_RMIRegistry"/>
  <property name="service">
     <bean class="org.softamis.cluster4spring.example.ServiceImpl"/>
  </property>
  <property name="serviceInterface" value="org.softamis.cluster4spring.example.Service"/>
  <property name="serviceName" value="TestService"/>
</bean>

<bean id="RMIServicesPublisher"
      class="org.softamis.cluster4spring.rmi.RmiServicePublisher">
  <property name="servicesRegistry" ref="_ServerServicesRegistry"/>
</bean>
```

8 Architecture and Implementation

This section of the document expands on Cluster4Spring implementation. In general, we pay attention to detail to give a deeper understanding of various remoting configuration specifics and customizations (if ones will be necessary).

Please refer to JavaDoc for Cluster4Spring as well to source code to find implementation details not provided there. Also, please refer to examples of configuration – they are fully commented and there you can find snippets of particular remoting configuration.

Current version of Cluster4Spring supports RMI as underlying remoting protocol. However, Cluster4Spring includes core engine that allows supporting different remoting protocols. Since support of particular protocol represents extending based on core classes, classes that are used to support remoting both on client and server sides have many common features and that is why we will try to describe generic (protocol agnostic) scheme of remoting implementation (highlighting details that are specific for RMI, if necessary).

8.1 Client side

Most of unique features of Cluster4Spring remoting derived from implementation of client-side remoting functionality.

8.1.1 Components of proxy

Internally, the structure of the Cluster4Spring client side proxy is quite simple and is illustrated by diagram below:

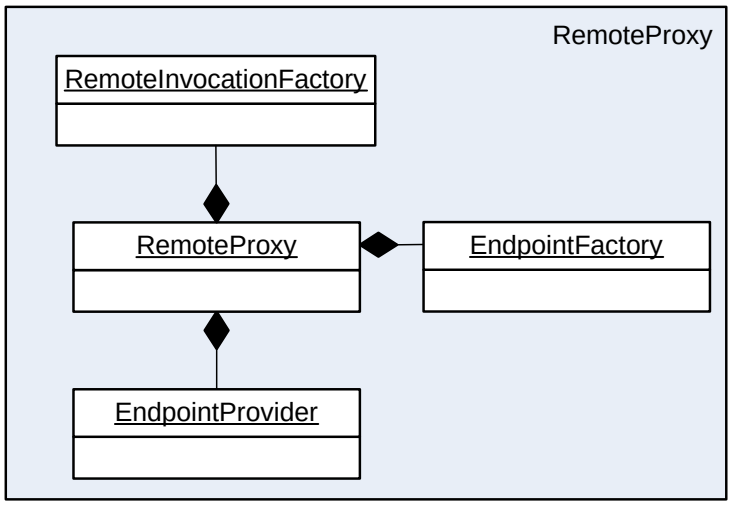

Figure 6 High level overview of client side proxy

Exactly as Spring proxy, Cluster4Spring remote proxy utilizes *RemoteInvocationFactory* for creating instances of *RemoteInvocation*. While it is not highlighted in Spring documentation, this is important part of remoting since custom instances (or customized instances) of *RemoteInvocation* class may be created if proper *RemoteInvocationFactory* is specified (if one is not explicitly specified during configuration of remote proxy, default one will be created automatically). With this mechanism it is possible to pass some additional information to server transparently for caller code with every (or particular, depending on implementation) remote

call. For example, it is possible to pass information about security credentials, some session data associated with particular user etc.

Cluster4Spring client proxy utilizes notion that is absent in standard Spring remoting – *Endpoint*. *Endpoint* represents wrapper over actual remoting protocol. For example, for RMI based remoting there is appropriate implementation of *Endpoint* which holds reference to *Remote* object and is responsible for invoking remote service either via *InvocationHandler* (if Spring based remoting is used on server-side) or via ordinary RMI stub.

Accordingly, *EndpointFactory* is responsible for locating remote service by known location (like using *Naming.lookup()* method for RMI based remoting) and for creation of endpoints. Finally, there is another very important component of client Cluster4Spring proxy – *EndpointProvider*. *EndpointProvider* represents policy that defines which remoting scheme is used. While overall functionality of client proxy does not relate to remoting scheme, *EndpointProvider* knows where (and now) to find remote service and provides *Endpoint* that corresponds to particular remote service location for actual method invocation.

8.1.2 Remote call execution

The following diagram (very simplified) illustrates steps that are performed within client proxy during processing remote call:

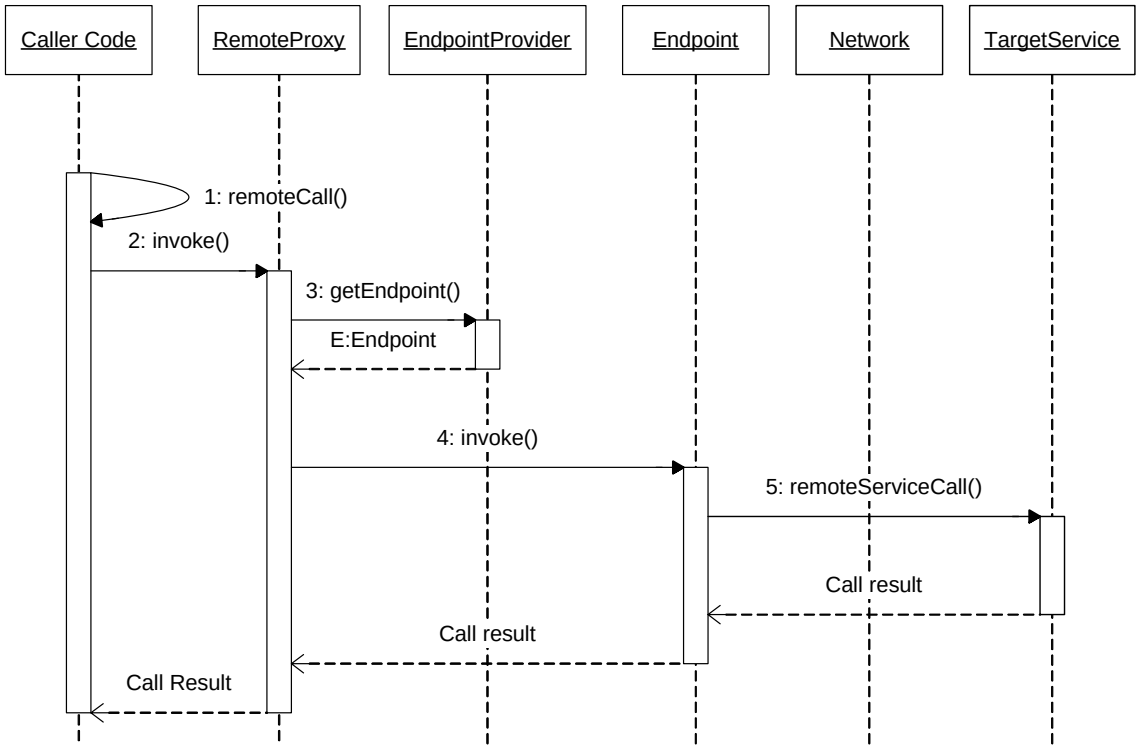

Figure 7 Generic scheme of remote call processing

As soon as caller code of application initiates remote call, *RemoteProxy* requests *Endpoint* from *EndpointProvider* (step 3). As soon as *Endpoint* is obtained, proxy delegates method invocation to it (step 4). *Endpoint* performs necessary actions to perform remote call according to currently used remote protocol (step 5). As soon as remote service finished execution of call, results are sent back to client and to then to application code.

8.1.3 Handling errors

The nature of remoting assumes that in real life various unexpected situations can occur. For example, network connection may be broken; server may become inaccessible or shut down unexpectedly and so on.

While “normal” application exceptions, which are thrown by server-side code, are leaved untouched by Cluster4Spring logic and passed back to caller code of client application, the remoting related ones are internally captured by Cluster4Spring logic and processing (defined by appropriate settings) is performed by Cluster4Spring code.

The additional processing of network related issues is performed to increase stability of distributed application and provide additional recovering mechanisms (for example, try to re-locate remote service and call it again or select another instance of remote service and redirect remote call to it).

The following diagram illustrates simplified scheme of handling networking related exceptions that can occur during remote call execution performed by Cluster4Spring core:

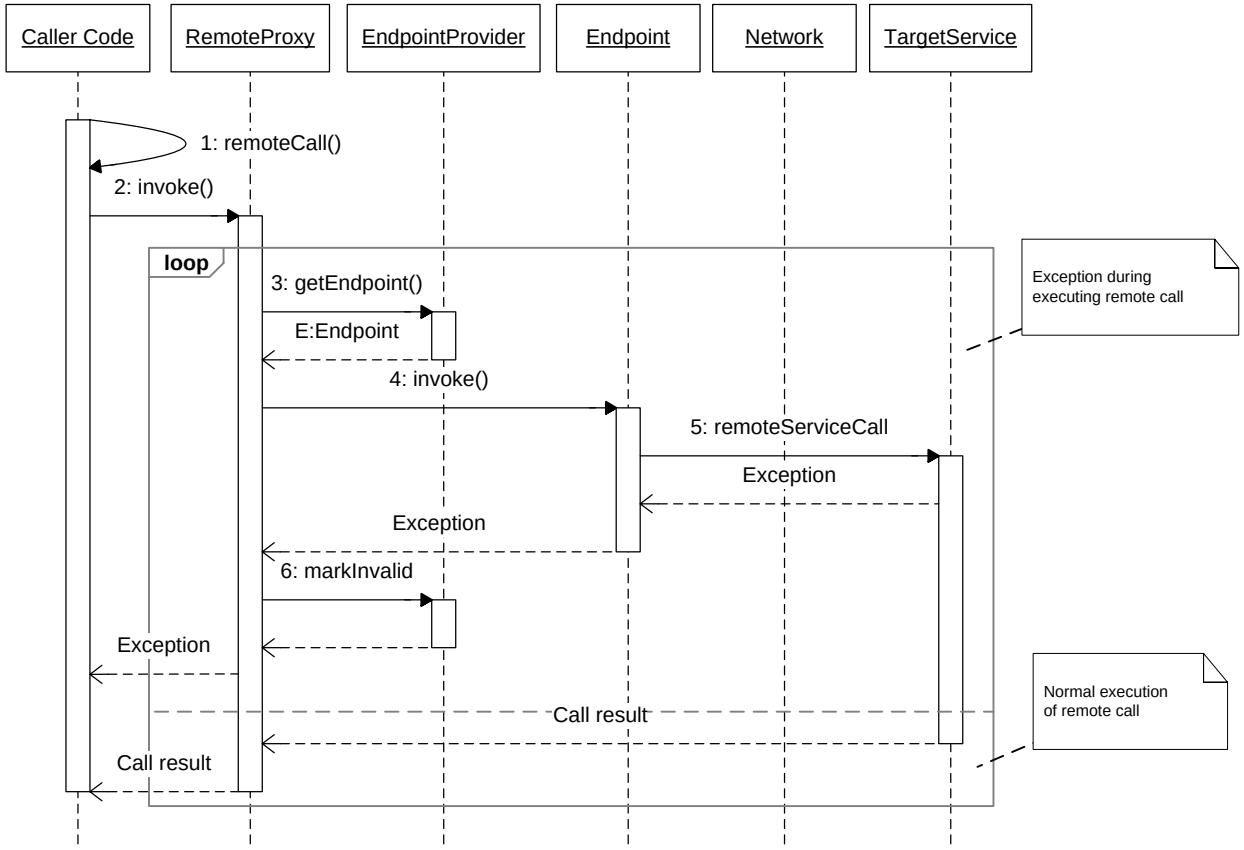

Figure 8 Simplified scheme of errors processing

We can see on the diagram that, if some networking related exception occurred during performing remote call, that exception is caught within *RemoteProxy* and *EndpointProvider* is informed that *Endpoint* used to perform remote call is invalid (step 6). It is assumed that endpoints that were marked as invalid ones will not be returned by *EndpointProvider* during all subsequent requests for endpoints.

Further behavior (not shown on this diagram for simplicity's sake) depends on configuration of remote proxy.

At the moment, there are two options (specified by appropriate properties in the corresponding proxy class) which specify the actions that should be performed later.

These properties are:

- **refreshEndpointsOnConnectFailure** - if set to true, *RemoteProxy* will force *EndpointProvider* to refresh all endpoints (by re-locating them) and will try to perform remote method invocation again;
- **switchEndpointOnFailure** - if set to true, *RemoteProxy* will ask *EndpointProvider* for another *Endpoint* available and will try to perform remote call using obtained endpoint. This process will be repeated (if other exceptions occur) until some non-invalid *Endpoint* exists;

If both these options are set to true, the **refreshEndpointsOnConnectFailure** option has higher priority in current implementation of Cluster4Spring. If both of these options are set to false, the original remoting exception will be passed to caller code.

8.1.4 Caching endpoints

It is possible to implement various schemes of resolving remote service. Two options control this:

- refreshEndpointsOnStartup (on *RemoteProxy*) - if set to true, during initialization of proxy the corresponding *EndpointProvider* will be requested to locate all available endpoints. Such scheme requires that at least one instance of remote service should be available during client application start. If this option is set to false, *EndpointProvider* will perform actual lookup of remote services during first invocation of the remote method of remote service;
- cacheEndpoints (on *EndpointProvider*) – if set to true, *EndpointProvider* will cache located endpoints for remote service and will use them during subsequent invocation. If option is set to false, *EndpointProvider* will perform lookup of endpoints on every call of any method of remote service;

By default, endpoints are not cached and are refreshed on startup.

8.1.5 Selecting instance of remote service for invocation

If there are several instances of remote service available, during performing remote call it is necessary to define which one should be used. Such an issue occurs if one of one-to-many remoting schemes is used.

It is possible to implement various schemes of invocations there using appropriate policy. Corresponding *EndpointProvider* that supports one-to-many remoting scheme does not perform actual selection of endpoint for invocation. Instead, it delegates selection of endpoint for invocation to the appropriate policy (one implemented via *EndpointSelectionPolicy* interface).

The following diagram illustrates the process of performing remote call for one-to-many remoting scheme:

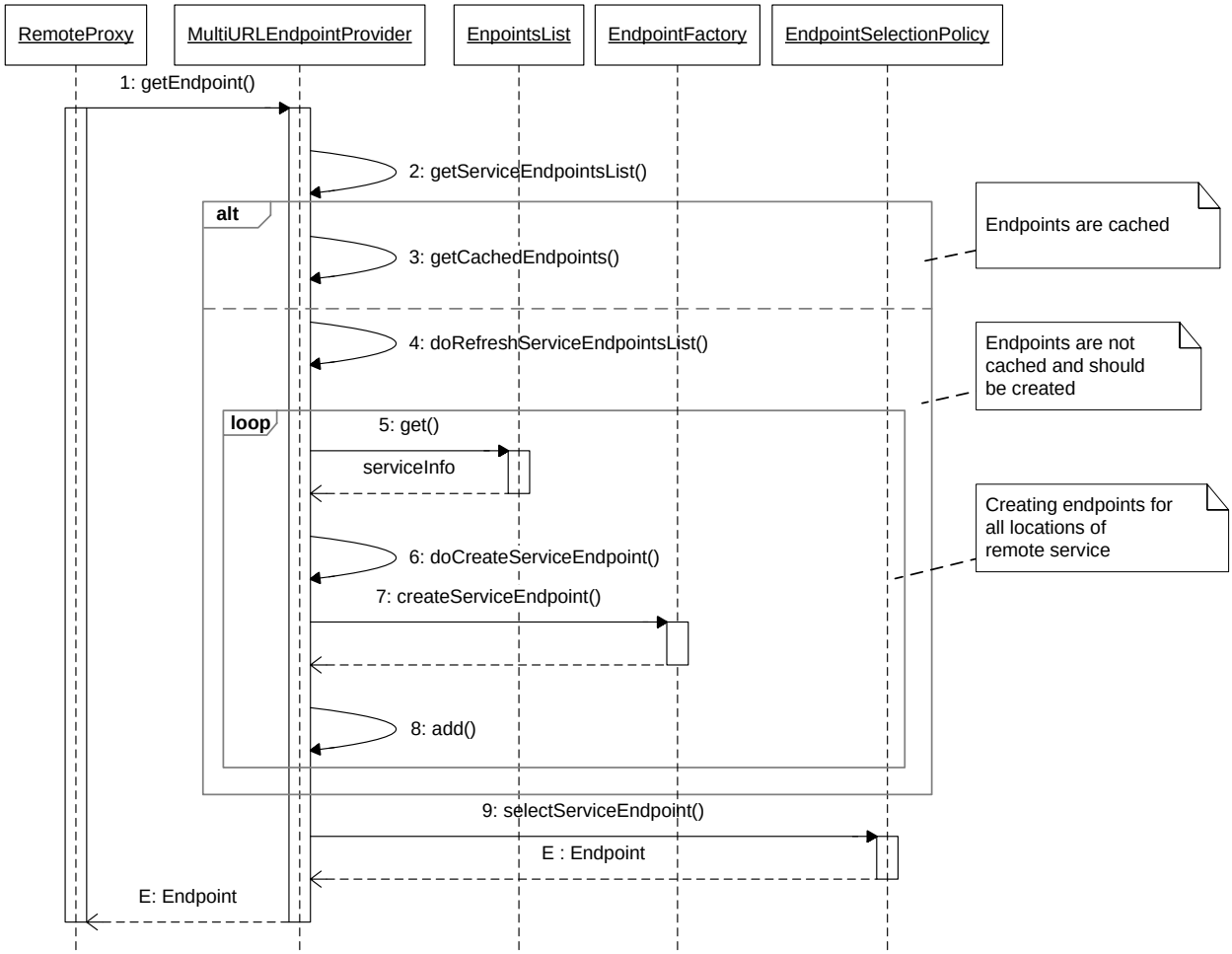

Figure 9 Performing remote call for one-to-many remoting scheme

The actual implementation class of policy must be used to select endpoints, which can be provided during configuration of remote proxy.

Cluster4Spring has three built-in implementations of *EndpointSelectionPolicy* (and obviously, it is possible to use custom ones):

- *DefaultEndpointSelectionPolicy* – returns first valid endpoint from available endpoints list. If such policy is used, all remote calls will be performed to the same instance of remote service (shortly, to one server) until it is available;
- *ShuffleEndpointSelectionPolicy* – shuffles content of available endpoints list and returns the first endpoint from the list. If such policy is used, each remote call will be potentially executed with different endpoints thus making workload of servers more uniform;
- *LastAccessTimeEndpointSelectionPolicy* – selects endpoint for invocation that was not accessed for longest period of time. Such a policy also pays a way for sharing the workload among servers on a uniform basis.

8.1.6 Miscellaneous

This section of document includes various small features that can be configured on client side of Cluster4Spring remoting.

- It is possible to customize all internals of client proxy – *RemoteInvocationFactory*, *EndpointProvider*, *EndpointFactory*, *EndpointSelectionPolicy* using appropriate properties available on proxy;
- It is possible to state that trace interceptor should be used as well as specify implementation of such interceptor. Trace interceptor can be used to trace (or profile or what ever you want) all remote calls which are performed by the particular proxy;
- It is possible to specify list of interceptor names which will be applied to remote calls invocation which are performed by proxy;
- There are two possible ways to configure client proxy for remote service using Spring configuration. In general, these ways are functionally identical, but require more or less verbose declaration markup in XML configuration. Please refer to examples for Cluster4Spring remoting configuration to find more about possible configuration options.

8.2 Server side

Cluster4Spring remoting includes custom version of remote service exporters. However, while Cluster4Spring exporters include some additional features in comparison with ones offered by Spring, it is possible to use standard Spring remoting for all supported remoting schemes except for dynamic discovering one-to-many remoting scheme.

8.2.1 Remote service exporter

To export some bean as remote service, Cluster4Spring uses the same notion of *Exporter* as Spring does. The exporter included in Cluster4Spring has similar functionality with that of Spring and can be configured in a similar way.

In addition to functionality of standard Spring exporter, Cluster4Spring-based one allows specifying whether trace interceptor should be used to track incoming remote calls and, if necessary, provide implementation for such interceptor. Also, Cluster4Spring based exporter makes it possible to specify the list of interceptor names that will be applied to every invocation of remote call.

With Cluster4Spring based exporter it is also possible to specify whether particular remote service be available for dynamic auto-discovering (please refer to the following section for details).

8.2.1.1 Quick configuration example

This is a quick example of minimal configuration for declaration of remote service exporter assuming that RMI is used as underlying remoting protocol.

```
<bean class="org.softamis.cluster4spring.rmi.RmiServiceExporter">
   <property name="registry" ref="_RMIRegistry"/>
   <property name="service">
      <bean class="org.softamis.cluster4spring.example.ServiceImpl"/>
   </property>
   <property name="serviceInterface" value="org.softamis.cluster4spring.example.Service"/>
   <property name="serviceName" value="TestService"/>
```

```
</bean>
```

As it is illustrated by listing above, in general exporting of remote service in Cluster4Spring is very close to exporting remote service using standard Spring remoting.

8.2.2 Remote service publisher

If one-to-many with dynamic services discovering remoting scheme is used, it is necessary to register remote services in distributed services registry. Such a registration is performed by *ServicePublisher*.

The actual publishing of remote services information in distributed registry is performed by *ServicePublisher* (and that is why configured instance of *ServicePublisher* should be present in Spring context).

The following diagram illustrates process of publishing information about remote services exposed in particular Spring context to distributed services registry:

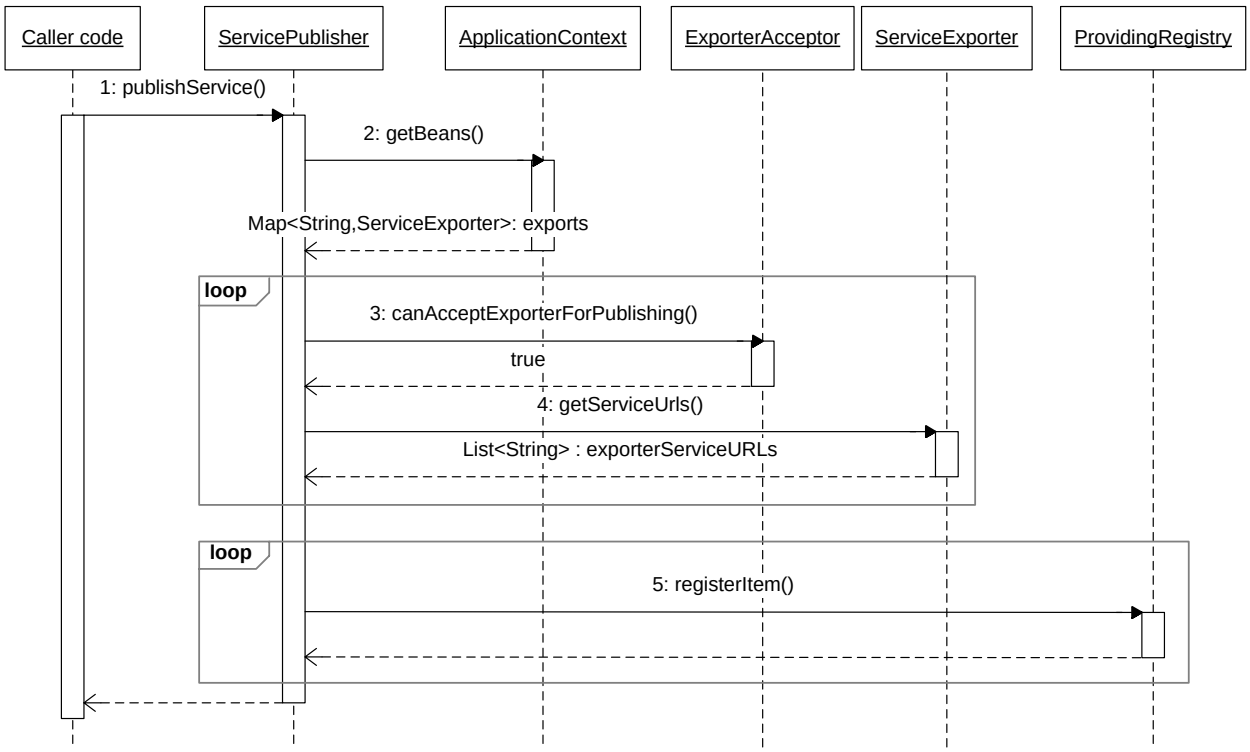

Figure 10 Publishing of remote services to distribute registry

To simplify configuration of remoting related subsystem (and to simplify internal architecture of remote service *Exporter*), each exporter does not store information about its service in distributed services registry. Instead, it just provides API that allows determining whether remote service exported by it should be stored in remote services registry as well as obtaining data that should be stored in services registry.

During initialization of Spring context, *ServicePublisher* collects exporters with specified class from Spring application context.

As soon as list of exporters is collected, *ServicePublisher* invokes instance of *ExporterAcceptor* (which is provided as a part of *ServicePublisher* configuration) to examine each exporter and

decide whether information on remote service exported by that particular exporter should be added into distributed services registry which is associated with particular instance of *ServicePublisher*.

Finally, *ServicePublisher* iterates over the list of exporters that are ready for publishing, retrieves information about remote service and stores it in distributed services registry.

On closing of Spring application context, *ServicePublisher* iterates over the list of published exporters and removes appropriate information about published remote services from distributed services registry.

8.2.2.1 Quick configuration example

This is a quick example of minimal configuration for declaration of remote service publisher.

```
<bean id="RMIServicesPublisher.minimal"
      class="org.softamis.cluster4spring.rmi.RmiServicePublisher">
   <property name="servicesRegistry" ref="_ServerServicesRegistry"/>
</bean>
```

For futher details, please refer to the following section.

9 Examples of configuration

This section contains several examples of configuring various schemes of Cluster4Spring remoting via Spring configuration files. Please note that it is possible to use two forms of remote proxy factory configuration - one that uses generic proxy factory class or one that uses concrete classes of remote proxy factories (shorter form).

9.1 Generic Configuration of Remote Proxy

While it is possible to implement different schemes of remoting using Cluster4Spring, several common configuration options are common to all remoting schemes. In this section, we go into details about these options and later we will highlight only the specifics of every particular remoting scheme.

The following listing illustrates how to configure generic remote proxy factory bean (providing that RMI is used as underlying remoting protocol). This example uses generic form of remote proxy factory bean that is applicable to every remoting scheme and is parameterized by appropriated endpoints provider (to support specifics of particular remoting scheme).

There is shorter form of proxy factory declaration that requires shorter markup and combines both definition of proxy with definition of *EndpointProvider*. Examples of such scheme usage will be shown later.

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

```
[1]  <bean name="TestService.verbose"
           class="org.softamis.cluster4spring.rmi.RmiProxyFactoryBean">
[2]    <property name="serviceInterface"
                 value="org.softamis.cluster4spring.example.Service"/>
[3]    <property name="refreshEndpointsOnStartup" value="true"/>
[4]    <property name="refreshEndpointsOnConnectFailure" value="true"/>
[5]    <property name="switchEndpointOnFailure" value="false"/>
[6]    <property name="registerTraceInterceptor" value="true"/>
[7]    <property name="remoteInvocationTraceInterceptor">
         <bean class="...">
             ...
         </bean>
       </property>
[8]    <property name="endpointProvider">
         ...
         ...
       </property>
[9]    <property name="interceptorNames" value="_Interceptor1, _Interceptor2"/>
[10]   <property name="endpointFactory">
         <bean class="org.softamis.cluster4spring.rmi.support.RmiEndpointFactory"/>
       </property>
[11]   <property name="remoteInvocationFactory">
         <bean
           class="org.springframework.support.DefaultRemoteInvocationFactory"/>
       </property>
    </bean>
```

Configuration details:

1. We declare exporter using appropriate class of remote proxy factory bean;
2. Specification of interface of remote service;

3. Policy that controls whether endpoints should be located on startup;
4. Policy that controls whether endpoints should be re-discovered on remoting failure and another attempt to invoke remote service should be performed;
5. Policy that controls whether attempt to select another endpoint on failure should be performed and attempt to invoke remote service should be performed;
6. Option that specifies whether trace interceptor should be created and applied for remote calls;
7. Sample declaration of remote trace interceptor (optional). If declaration of bean assumes that remote call trace interceptor should be registered, and there is no explicit declaration of interceptor is defined, default interceptor will be created and used;
8. Property that represent configuration of *EndpointProvider* used to locate remote services. Using this property, we can specify which implementation of *EndpointProvider* should be used. By specifying appropriate *EndpointProvider* there, we specify remoting scheme, which will be used by particular proxy;
9. Optionally, it is possible to specify that appropriate interceptors should be applied to remote call invocation;
10. Property that specifies which *EndpointsFactory* should be used by proxy. If factory is not specified explicitly, the default instance will be created internally;
11. Property that specifies which factory should be used to create *RemoteInvocation's*. If factory it not specified explicitly, default instance of factory will be created and used.

The listing above illustrates all properties that can be configured. However, most of them are optional and should be specified only if this is necessary for particular application.

9.2 Generic Configuration of Service Exporter

In general, using ServiceExporter implementation included into Cluster4Spring is required only if "one-to-many with dynamic services discovering" remoting scheme is used. For all other cases, it is possible to use standard services exporter provided by Spring. However, service exporter included in Cluster4Spring offers some additional functionality in comparison with Spring based one, which can be useful for particular applications.

The following listing illustrates generic configuration of service exporter (assuming that RMI is used as underlying remoting protocol).

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

```
[1]<bean name="TestService.custom.verbose"
         class="org.softamis.cluster4spring.rmi.RmiServiceExporter">
[2]   <property name="allowsAutoDiscovering" value="false"/>
[3]   <property name="clientSocketFactory" ref="_RMIClientSocketsFactory"/>
[4]   <property name="interceptorNames" value="_Interceptor1, _Interceptor2"/>
[5]   <property name="registerTraceInterceptor" value="true"/>
[6]   <property name="remoteInvocationExecutor" ref="_RemoteInvocationExecutor"/>
[7]   <property name="remoteInvocationTraceInterceptor">
         <bean class="...">
           ...
         </bean>
      </property>
[8]   <property name="registry" ref="_RMIRegistry"/>
```

```
[9]  <property name="service" ref="TestService.bean"/>
[10] <property name="serviceInterface"
               value="org.softamis.cluster4spring.example.Service"/>
[11] <property name="serviceName" value="TestService"/>
  </bean>
```

Configuration comments:

1. Declaration of remote service exporter using appropriate class of exporter;
2. Property that allows to specify whether this exporter should be available for publishing information on its remote service by *ServicePublisher* (optional);
3. Socket which should be used to create sockets used by RMI (optional);
4. Optional list of interceptors that should be applied for remote method invocation;
5. Optional property which indicates that trace interceptor should be used;
6. Optionally, it is possible to apply custom *RemoteInvocationExcecutor*;
7. Optionally, it is possible to provide custom implementation of trace interceptor. If this property is not specified, default instance of trace interceptor will be created;
8. Reference to RMI registry bean;
9. Reference to bean which represents implementation of remote service;
10. Interface of remote service;
11. Name of remote service which will be used to publish it in RMI registry;

The listing above illustrates all properties that can be configured. However, most of them are optional and should be specified only if this is necessary for particular application.

9.3 Configuring one-to-one remoting

We provide example of one-to-one remoting scheme configuration. For simplicity's sake, only client-side configuration of remoting is shown. On the server side, for this remoting scheme either standard Spring exporter or Cluster4Spring one can be used.

In general, configuring proxy factory for one-to-one remoting scheme is very simple and is illustrated by examples below.

9.3.1 Using specialized factory bean

The listing below illustrates minimal configuration with specialized proxy factory bean that is used to implement one-to-one scheme of remoting.

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

```
[1] <bean name="TestService.compact.minimal"
          class="org.softamis.cluster4spring.rmi.RmiSingleUrlProxyFactoryBean">
[2]   <property name="serviceInterface"
               value="org.softamis.cluster4spring.example.Service"/>
[3]   <property name="serviceUrl" value="rmi://localhost:1099/TestService"/>
    </bean>
```

Configuration comments:

1. Declaration of proxy factory bean using appropriate specialized class which supports one-to-one remoting scheme;
2. Property which represents interface of remote service;
3. Property represents explicitly specified URL of remote service. Cluster4Spring also supports short form of RMI URL without protocol prefix - for this example it can be specified as "localhost:1099/TestService";

Of course, it is possible to configure all generic properties as it shown in section 9.1.

9.3.2 Using generic proxy factory bean

If the use of generic exporter is more preferable for some reasons, it possible to implement different configuration (which is functionally equivalent to one listed above) that utilizes generic exporter customized by specific EndpointProvider. The following listing illustrates examples of such configuration.

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

```
[1]<bean name="TestService.verbose.minimal"
        class="org.softamis.cluster4spring.rmi.RmiProxyFactoryBean">
[2] <property name="serviceInterface"
            value="org.softamis.cluster4spring.example.Service"/>
[3] <property name="endpointProvider">
[4]   <bean
class="org.softamis.cluster4spring.support.provider.SingleUrlEndpointProvider">
[5]     <property name="serviceUrl" value="rmi://localhost:1099/TestService"/>
      </bean>
    </property>
  </bean>
```

Configuration comments:

1. Declaration of proxy factory bean using appropriate generic class which is agnostic to remoting scheme;
2. Property which represents interface of remote service;
3. Property used to specify *EndpointProvider* implementation;
4. Here we specify particular *EndpointProvider* which is used to support one-to-one remoting scheme;
5. Property that represents explicitly specified URL of remote service.

9.4 Configuring static one-to-many remoting

In this section we provide an example of configuration for “static one-to-many” remoting scheme that assumes that remote service is available in several known remoting locations. For simplicity’s sake, here we illustrate only configuration that should be performed on client side. On the server side, for this remoting scheme either standard Spring exporter or Cluster4Spring one can be used.

Configuring proxy factory for such remoting scheme is also quite simple and is illustrated by listings below.

9.4.1 Using specialized proxy factory bean

The listing below illustrates minimal configuration with specialized proxy factory bean that is used to implement static one-to-may scheme of remoting.

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

```
[1]<bean name="TestService.compact.minimal"
        class="org.softamis.cluster4spring.rmi.RmiUrlListProxyFactoryBean">
[2] <property name="serviceInterface"
              value="org.softamis.cluster4spring.example.Service"/>
[3] <property name="serviceURLs">
      <list>
        <value>rmi://localhost:1097/TestService</value>
        <value>rmi://localhost:1098/TestService</value>
        <value>rmi://localhost:1099/TestService</value>
      </list>
    </property>
[4] <property name="endpointSelectionPolicy">
      <bean
class="org.softamis.cluster4spring.support.invocation.DefaultEndpointSelectionPolicy"/>
    </property>
  </bean>
```

Configuration comments:

1. Declaration of proxy factory bean using appropriate specialized class which supports static one-to-many remoting scheme;
2. Property which represents interface of remote service;
3. Property that allows to assign list of available known locations of remote service;
4. Property that allows specifying policy that should be used to select particular endpoint from list of available ones on remote method invocation. Specifying that policy is optional, if one is not provide explicitly, the instance of default policy will be created and internally used;

Again, it is possible to configure all generic properties as it shown in section 9.1.

9.4.2 Using generic proxy factory bean

If, for some reasons, using generic exporter is more preferable, it possible to use a different configuration (which is functionally equivalent to the listing above) that utilizes generic exporter customized by specific *EndpointProvider*. The following listing illustrates such a configuration.

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

```
[1]<bean name="TestService.verbose.minimal"
         class="org.softamis.cluster4spring.rmi.RmiProxyFactoryBean">
[2]  <property name="serviceInterface"
               value="org.softamis.cluster4spring.example.Service"/>
[3]  <property name="endpointProvider">
[4]     <bean
class="org.softamis.cluster4spring.support.provider.UrlListEndpointProvider">
[5]        <property name="serviceURLs">
            <list>
              <value>rmi://localhost:1097/TestService</value>
```

```
            <value>rmi://localhost:1098/TestService</value>
            <value>rmi://localhost:1099/TestService</value>
          </list>
         </property>
       </bean>
    </property>
  </bean>
```

Configuration comments:

1. Declaration of proxy factory bean using appropriate generic class which is agnostic to remoting scheme;
2. Property which represents interface of remote service;
3. Property used to specify *EndpointProvider* implementation;
4. Here we specify particular *EndpointProvider* which is used to support one-to-one remoting scheme (note that the listing does not explicitly declare endpoints selection policy);
5. Property that allows to assign a list of known available locations of remote service;

9.5 Configuring one-to-many remoting with dynamic services discovering

This scheme requires more complicated configuration. However, that complexity doesn't relate to declaration of client proxy and service exporter, but rather is related to necessary additional configuration of distributed registry.

The following examples assume that DistRegistry implementation of distributed registry is used as well as appropriate implementations of endpoint providers and publisher that utilize DistRegistry. However, it is possible to create custom endpoint providers and publishers that will utilize different implementation of distributed services registry.

Please refer to documentation for DistRegistry (included into Cluster4Spring distributive) to get more details on how to configure distributed registry.

9.5.1 Client part of distributed services registry

Before configuring the remote proxy factory itself, it is necessary to configure the distributed services registry used for services discovering. The following listing illustrates configuration for distributed services registry.

Please refer to configuration comments below to get more information on every particular element of declaration.

```
[1]  <bean id="_ClientServicesRegistry"
           class="org.softamis.net.registry.spring.DefaultConsumingRegistry">
[2]    <property name="communicationHelper">
[3]      <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
               init-method="init" destroy-method="close">
[4]       <property name="multicaster">
            <bean class="org.softamis.net.multicast.DefaultMulticaster"
                 init-method="start" destroy-method="close">
              <property name="groupName" value="230.0.0.10"/>
              <property name="timeToLive" value="5"/>
           <property name="port" value="1000"/>
         </bean>
```

```
        </property>
      </bean>
    </property>
[5] <property name="requestItemsOnInit" value="true"/>
[6] <property name="defaultMessageSignature" value="EX"/>
  </bean>
```

Configuration comments:

1. Declaration of client part for distributed services registry. Here we define a registry, which provides information on remote services locations. This is a client part of the distributed services registry. Also, there is the corresponding part, which should be declared on server that publishes it services. This registry will be used by appropriate endpoint providers to obtain the list of available service locations. In general, any class, which implements appropriate protocol for registry, may be used in remoting (for example, one that utilizes some remote cache, like JBoss cache). However, we use default ready to use implementation of client registry there.
2. This property specifies communication helper, which hides details of underlying network communications;
3. In this example, we use UDP multicast based protocol; however, a different implementation of internal registry communication can be used, such as JGroups based one.
4. *Multicaster* which is used to issue and receive UDP multicast messages, which contain information about published services locations;
5. Property that Indicates whether registry should issue request for published items during startup;
6. Signature used to filter UDP messages that contain information about service's locations. Only data obtained with packets which has such signature will be stored in client registry;

9.5.2 Using specialized proxy factory bean

The listing below illustrates minimal configuration with specialized proxy factory bean that is used to implement dynamic one-to-may scheme of remoting.

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

```
[1]<bean name="TestService.compact.minimal"
          class="org.softamis.cluster4spring.rmi.RmiDiscoveringProxyFactoryBean">
[2]  <property name="serviceInterface"
              value="org.softamis.cluster4spring.example.Service"/>
[3]  <property name="serviceName" value="TestService"/>
[4]  <property name="serviceGroup" value="DEFAULT"/>
[5]  <property name="clientServicesRegistry" value="_ClientServicesRegistry"/>
  </bean>
```

Configuration comments:

1. Declaration of proxy factory bean using appropriate specialized class which supports dynamic one-to-many remoting scheme;
2. Property which represents interface of remote service;
3. Property which specifies name of the service;

4. Property that specifies name of group which particular remote service belongs to. Group defines the prefix of key which is used to publish information about service in distributed registry (key is composed as "serviceGroup/serviceName"). If one is not specified, default name of group is used;
5. Reference to client part of distributed services registry used to get information about available locations of service;

Please note that it is not necessary to define one bean that represent client services registry for every declaration of remote service proxy. On the contrary, several remote service proxies may use the same instance of distributed services registry.

Again, it is possible to configure all generic properties as it shown in section 9.1.

9.5.3 Using generic proxy factory bean

If, for some reasons, using generic exporter is more preferable, it is possible to use a different configuration (which is functionally equivalent to listing above) that utilizes generic exporter customized by specific *EndpointProvider*. The following listing illustrates such a configuration.

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

```
[1]<bean name="TestService.verbose.minimal"
      class="org.softamis.cluster4spring.rmi.RmiProxyFactoryBean">
   <property name="serviceInterface" value="org.softamis.cluster4spring.example.Service"/>
   <property name="endpointProvider">
[2] <bean
class="org.softamis.cluster4spring.support.provider.DiscoveringEndpointProvider">
[3]   <property name="serviceName" value="TestService"/>
[4]   <property name="serviceGroup" value="DEFAULT"/>
[5]   <property name="clientServicesRegistry" value="_ClientServicesRegistry"/>
    </bean>
   </property>
 </bean>
```

Configuration comments:

1. Declaration of proxy factory bean using appropriate generic class which is agnostic to remoting scheme;
2. Declaration of *EndpointProvider* that supports appropriate remoting scheme;
3. Property which specifies name of the service;
4. Property that specifies a name of the group which particular remote service belongs to.
5. Reference to client part of distributed services registry used to get information about available locations of service;

9.5.4 Server part of distributed services registry

To support one-to-many remoting scheme with dynamic services discovering, it is necessary to configure server part of distributed services registry.

The following listing illustrates configuration for server part of distributed services registry.

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

```
[1] <bean name="ServerServicesRegistry"
        class="org.softamis.net.registry.spring.DefaultProvidingRegistry"
        init-method="init"
        destroy-method="close">
[2]    <property name="communicationHelper">
[3]      <bean class="org.softamis.net.exchange.udp.UDPCommunicationHelper"
             init-method="init" destroy-method="close">
[4]        <property name="multicaster">
             <bean class="org.softamis.net.multicast.DefaultMulticaster"
                 init-method="start" destroy-method="close">
               <property name="groupName" value="230.0.0.10"/>
               <property name="timeToLive" value="5"/>
               <property name="port" value="1000"/>
             </bean>
           </property>
         </bean>
      </property>
[5]   <property name="defaultMessageSignature" value="EX"/>
  </bean>
```

Configuration comments:

1. Declaration of server part of distributed services registry used to publish information about remote service locations.
2. This property specifies communication helper, which hides the details of underlying network communications;
3. In this example, we use UDP multicast based protocol; however, a different implementation of internal registry communication can be used, such as JGroups based one.
4. *Multicaster* which is used to issue and receive UDP multicast messages, which contains information about published services locations;
5. Signature used to filter UDP messages that contain information about service's locations. Only data obtained with packets which have such a signature will be stored in client's registry;

9.5.5 Configuring remote service exporter

If one-to-many remoting scheme with dynamic services discovering is used, it is necessary to use remote service exporter included into Cluster4Spring.

The following listing illustrates server-side configuration for remote service exporter.

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

```
[1]<bean class="org.softamis.cluster4spring.rmi.RmiServiceExporter">
[2]   <property name="allowsAutoDiscovering" value="true"/>
[3]   <property name="registry" ref="RMIRegistry"/>
[4]   <property name="service">
        <bean class="org.softamis.cluster4spring.example.ServiceImpl"/>
      </property>
[5]   <property name="serviceInterface"
                value="org.softamis.cluster4spring.example.Service"/>
[6]  <property name="serviceName" value="TestService"/>
  </bean>
```

Configuration comments:

1. This is fairly simple configuration of service exporter which is required to let publishing information about service in distributed registry and so dynamic discovering of service by client;
2. This property is used by appropriate ExporterAcceptor and should be set to true if one of acceptors included into Cluster4Spring is used. Default is true.
3. Reference to RMI registry (declaration of registry is not included there);
4. Declaration of bean which represents implementation of remote service;
5. Property which specifies interface of remote service;
6. Property that specifies the name of remote service – the corresponding name should be used for client-side proxy of service.

Please note that general properties that can be configured for service exporter were considered earlier and are not included into the listing above.

9.5.6 Configuring remote services publisher

It is responsibility of *ServicePublisher* to collect information about exported remote services and store it in the distributed services registry. The following listing illustrates server-side configuration for *ServicePublisher*.

Please refer to configuration comments below to get more information on every element of declaration.

```
[1]<bean id="RMIServicesPublisher"
        class="org.softamis.cluster4spring.rmi.RmiServicePublisher">
[2]  <property name="servicesRegistry" ref="ServerServicesRegistry"/>
[3]  <property name="cacheAutoDiscoveredServicesInfo" value="true"/>
[4]  <property name="serverID" value="123"/>
[5]  <property name="serviceGroup" value="DEFAULT"/>
[6]  <property name="serverType" value="DEF"/>
[7]  <property name="exporterAcceptor">
       <bean class="org.softamis.cluster4spring.support.context.DefaultExporterAcceptor"/>
     </property>
   </bean>
```

Configuration comments:

1. First, we declare *ServicePublisher* using appropriate class;
2. Property that contains reference to server part of distributed services registry. This registry is used to publish information about services and this information is discoverable by a client's part of registry;
3. Property that indicates whether publisher should cache information about services it published internally and use it during services un-publishing (instead of re-discovering them). This function is optional and default value is false;
4. Server ID represents unique ID of server within a cluster (or within the same providing registry). If there are several servers of the same type within the same cluster, it is necessary to distinguish them;

5. Name of group which services published by particular publisher belongs to. Group defines the prefix of key which is used to publish information about service in distributed registry (key is composed as "serviceGroup/serviceName"). If one is not specified, default name of group is used; **IMPORTANT**: If service group is specified there, it's **REQUIRED** that the same service group should be specified in configuration of appropriate client part of remote service to insure that both server and client use the same key for service publishing and discovering with the use of distribute registry;
6. This is a type of the server. In general, this property is optional (default is “DEF”), but can be useful for some monitoring tools, for example;
7. Acceptor that is used to define whether particular remote service exporter should be published via this publisher to underlying distributed services registry. Specifying acceptor is optional.

Of course, it is not necessary to declare one instance of *ServicePublisher* per instance of *ServiceExporter*. Instead of this, one *ServicePublisher* should be declared for one instance of distributed service registry.

9.6 Generic configuration tips

The examples above are correct, but since their main purpose was demonstrating options that could be configured, they are not optimal.

This section of document contains some small tips that allow having optimal setup of remoting subsystem with the use of short-form configuration. In general, they are not specific to Cluster4Spring remoting and are quite general.

The list of such tips is very short and contains just two major recommendations (funny, but they are applicable to configuring standard Spring remoting subsystem):

1. Do not rely on built-in defaults for internal remoting components.

   The implementation of Cluster4Spring remoting provides many places of creating default components of, for example, client remote proxy factory “by default” if ones are not specified explicitly. While it is fine and will work, this is not optimal solution – if you have several instances of, for example, client proxy factory, each such instance will contain references to unique instances of these components (the following ones could mentioned there - *RemoteInvocationFactory*, *EndpointFactory*, *EndpointSelectionPolicy* and so on). It is optimal solution to declare instances of such components explicitly and share them between appropriate instances of client-side proxy factory beans. The same is applicable to the server-side part of remoting too.

2. Use bean definition inheritance.

   Imagine that the server exports, say, twenty remote services (and client imports them, accordingly). Instead of defining configuration for every instance of remote service exporter, use abstract bean definition which contains common configuration for all properties of exporter (or remote proxy bean) and later use child bean definition which extends such generic one and represents configuration of particular instance or exporter (or proxy factory).

The following example illustrates the preferred way to define remoting related configuration assuming that client uses more than one remote service applied to configuration of one-to-one remoting (for simplicity’s sake). Obviously, it will be even more effectively applied to other remoting schemes.

```
[1]<bean name="_RmiEndpointFactory"
        class="org.softamis.cluster4spring.rmi.support.RmiEndpointFactory"/>

   <bean name="_RemoteInvocationFactory"
        class="org.springframework.support.DefaultRemoteInvocationFactory"/>

[2]<bean name="REMOTE_SERVICE_CLIENT"
        class="org.softamis.cluster4spring.rmi.RmiSingleUrlProxyFactoryBean"
        abstract="true">
    <property name="refreshEndpointsOnConnectFailure" value="true"/>
    <property name="refreshEndpointsOnStartup" value="true"/>
    <property name="registerTraceInterceptor" value="true"/>
    <property name="switchEndpointOnFailure" value="false"/>
    <property name="interceptorNames"
              value="_TestClientLoggingInterceptor, _TestClientEmptyInterceptor"/>
    <property name="endpointFactory" ref="_RmiEndpointFactory"/>
    <property name="remoteInvocationFactory" ref="_RemoteInvocationFactory"/>
  </bean>

[3]
  <bean name="TestService1" parent="REMOTE_SERVICE_CLIENT">
    <property name="serviceInterface"
              value="org.softamis.cluster4spring.example.ServiceImpl"/>
    <property name="serviceUrl" value="rmi://serverA:1099/TestService"/>
  </bean>

  <bean name="TestService2" parent="REMOTE_SERVICE_CLIENT">
    <property name="serviceInterface"
              value="org.softamis.cluster4spring.example.ServiceImpl"/>
    <property name="serviceUrl" value="rmi://serverB:1099/TestService"/>
  </bean>
```

Configuration comments:

1. First, we define beans instances that will be used by all instances of remote proxies. In general, if these instances are not declared explicitly, they will be created by default for every proxy. Therefore explicit declaration of them is needed if custom implementations are used or to increase the efficiency of the internal structures (no additional instances created) of remote proxies. Potentially, these declarations could be defined directly in-place within the appropriate property of remote proxy factory bean;
2. Than we define an abstract definition of remote proxy factory as abstract bean and specify all necessary properties for it. Later, if concrete remote proxies will be declared as used by this abstract definition, they will share all these properties. Such approach allows to have single place where common properties for remote proxies are managed and simplify declaration of particular remote proxy factory beans;
3. Finally, we declare concrete instances of RMI remote proxy factory beans. Since we declare them with the use of declaration inheritance, definition of particular proxy become quite compact;

10 Cluster4Spring Extension points

From the initial design phase of Cluster4Spring, the ability to extend and customize Cluster4Spring functionality was one of the main goals.

For example, most of methods are intentionally declared as protected ones (not private) to let the further extensions of Cluster4Spring features. In addition, Cluster4Spring internally uses quite a deep hierarchy of classes to provide good points of extensions.

Here is a brief list of possible extensions Cluster4Spring remoting functionality, if necessary:

1. Support of custom remoting protocols based on core functionality of Cluster4Spring remoting;
2. Custom policies for endpoints selection;
3. Custom implementations of endpoints (for example, for adding profiling or security related processing);
4. Integration with different implementations of distributed services registry (for example, based on JBoss cache);

11 Where to find more information

To find more information about Cluster4Spring and to get the newest versions of Cluster4Spring, please visit either SoftAMIS site

http://www.soft-amis.org/cluster4spring/index.html

Or visit project page on SourceForge:

http://sourceforge.net/projects/pscs/

Also, if you have questions or comments regarding Cluster4Spring, please feel to contact us via email:

cluster4spring@soft-amis.org

Please also refer to JavaDoc, examples, and source code for Cluster4Spring – we hope that you will find helpful details there.

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