Table of Contents

MundoCore Tutorial

Platform Tutorials

Reference

Getting Started

Prerequisites

First, make sure that the following packages are installed and that the `CLASSPATH` is set up properly.

- Java Development Kit 1.5 or higher (http://www.oracle.com/technetwork/java/index.html)
- Apache Ant (http://ant.apache.org/)

Installation

Unpack the distribution package:

```
unzip mundocore-java-1.0.0.zip
```

Run the configuration script:

```
./configure.sh
```

or `configure.bat` on Windows.

This will generate the configuration file `config/build.properties` which contains the system-specific paths for the MundoCore installation.

Testing

Change to the directory `samples/chat/pubsub` and run `ant`:

```
cd samples/chat/pubsub
ant
```

After compilation has finished, start two instances of the chat program on the same host. For example, an instance can be started as follows:

```
./run.sh
```

or `./run.bat` on Windows.

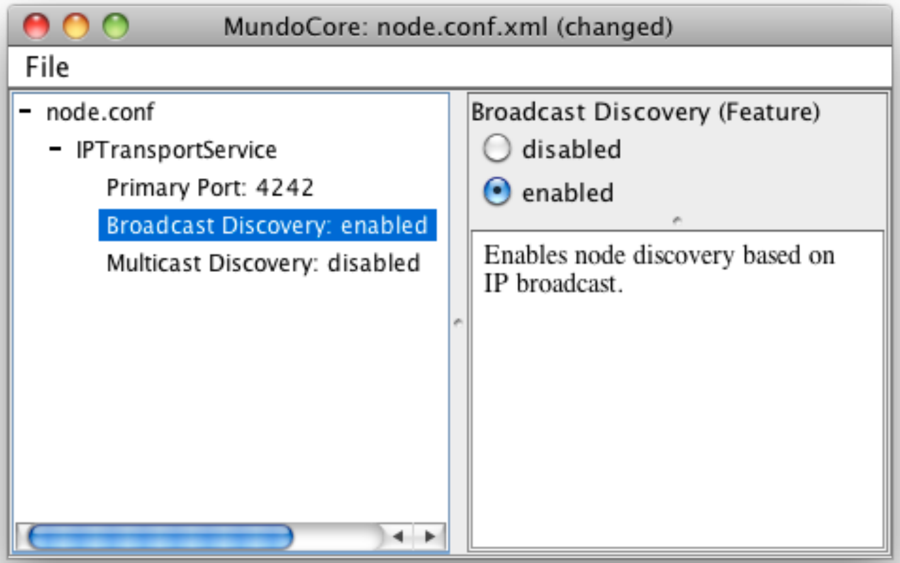
If you type a line of text (terminated by pressing Enter), the text should also appear as output of the other instance. To quit the program, type `.` and Enter at the beginning of a line.

Node Configuration

Initially, MundoCore nodes will only communicate within the local host. To enable communication over the network, a configuration file `node.conf.xml` must be created and a node discovery method must be enabled. To create a configuration file, run

```
ant config
```

Enable broadcast discovery and save the configuration file as `node.conf.xml` into the directory of the example `samples/chat/pubsub`.



With this configuration file, the sample program can be run on two different hosts in the local network. (Please make sure that MundoCore connections and discovery packets are not blocked by personal firewalls!)

Getting Started (Eclipse)

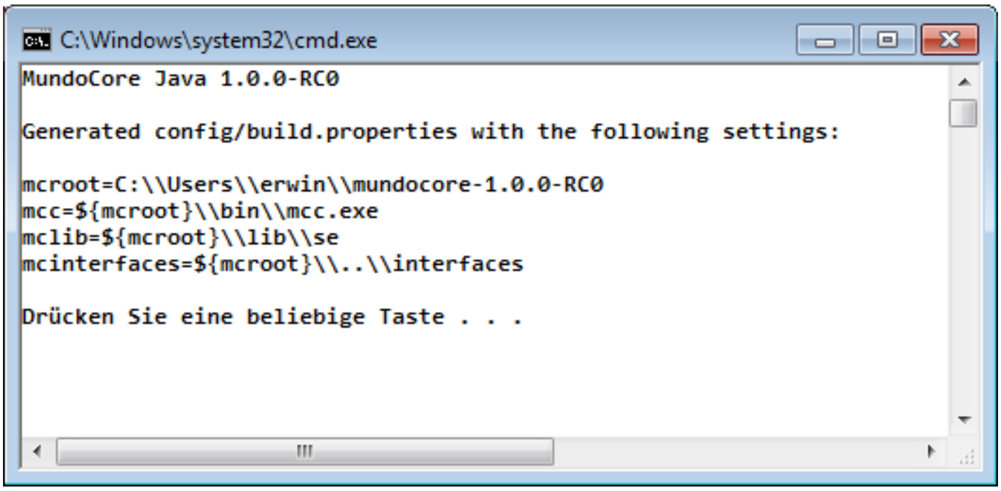
Prerequisites

- Eclipse IDE (any recent version will do; http://www.eclipse.org/)

Installation

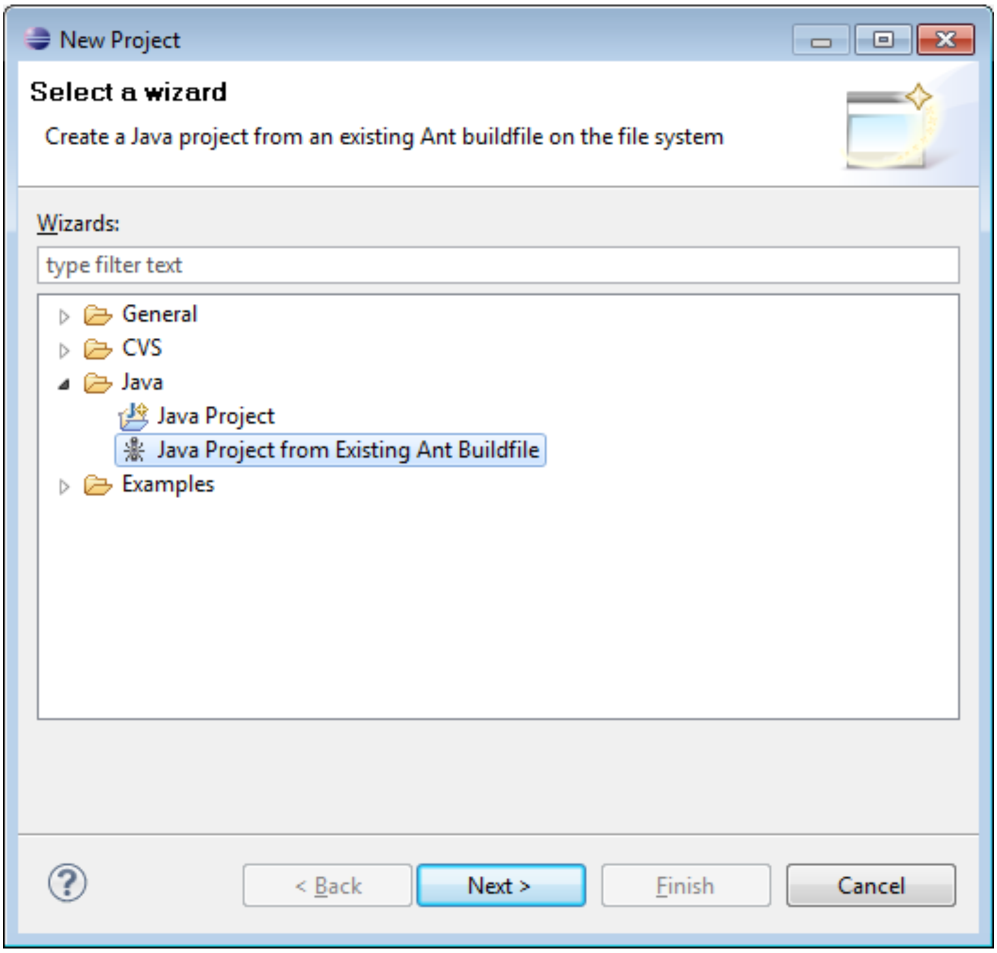
Unpack the distribution package `mundocore-java-1.0.0.zip`

Run the configuration script `configure.bat` (or `configure.sh`). This will generate the configuration file config/build.properties which contains the system-specific paths for the MundoCore installation:

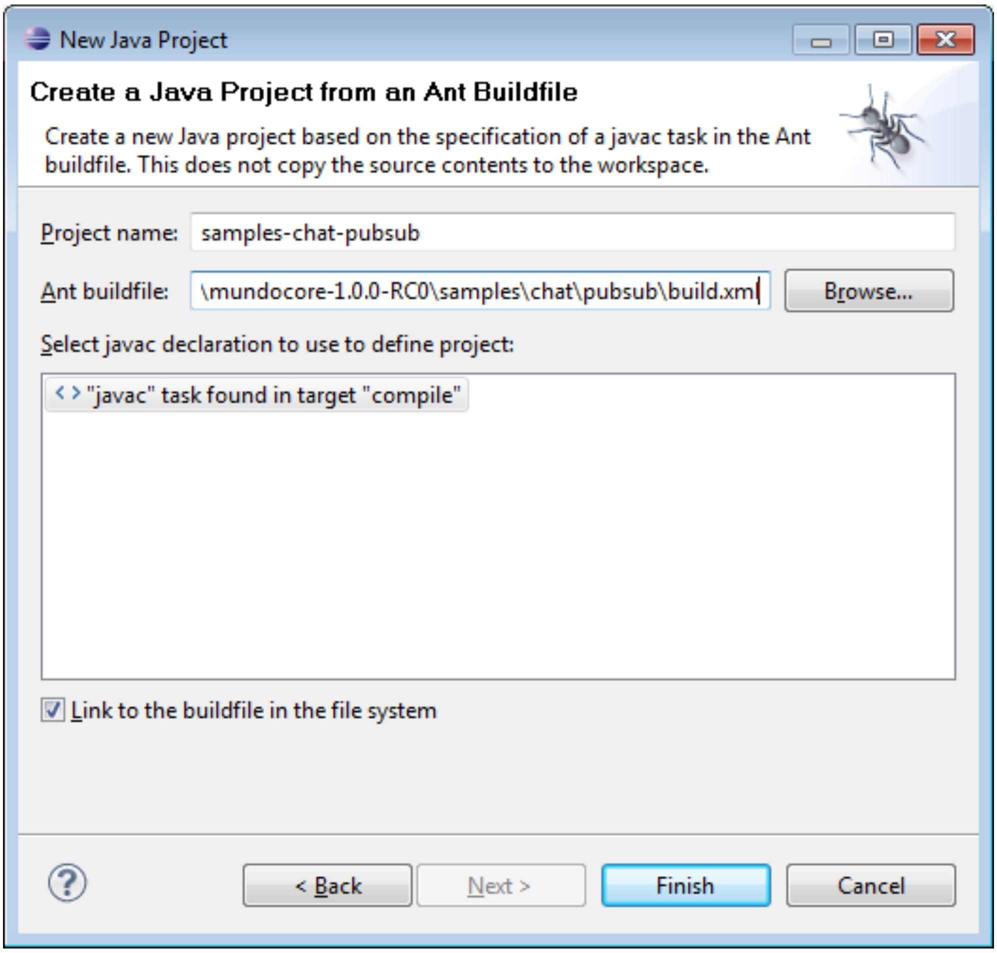


Testing

In Eclipse, create a new project from existing ant buildfile:

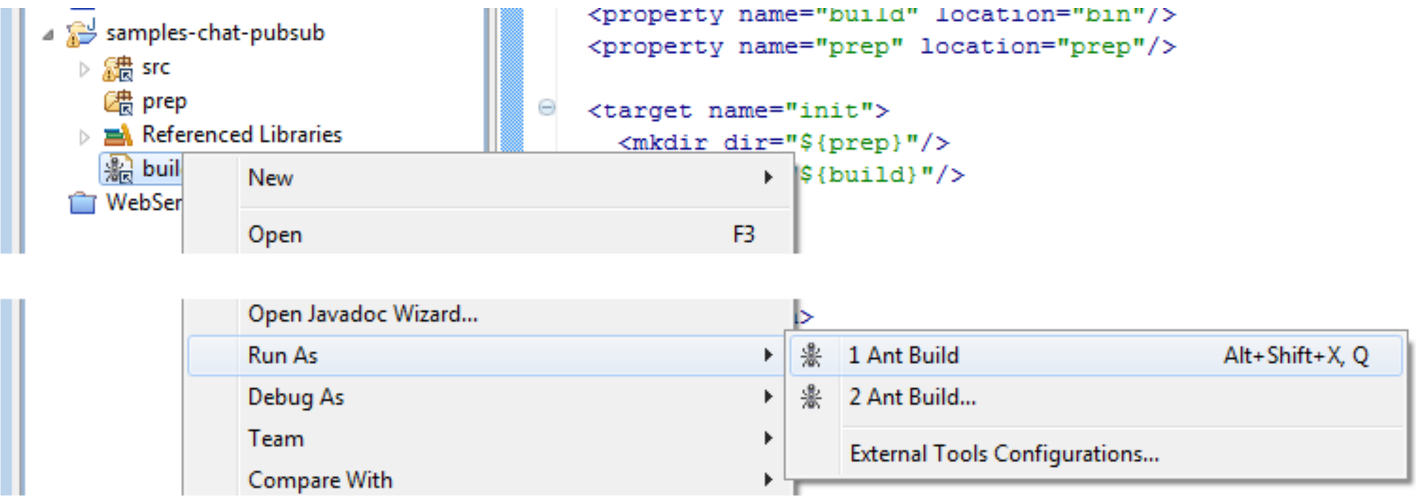


Select `build.xml` in the `samples/chat/pubsub` directory. Make sure that **Link to the buildfile in the file system** is selected:



You can run the sample program now by selecting **Run** from the Eclipse menu.

To run multiple instances of chat, it is helpful to create run scripts. To do this, invoke the build target `runscript`, which is the default target of `build.xml`:

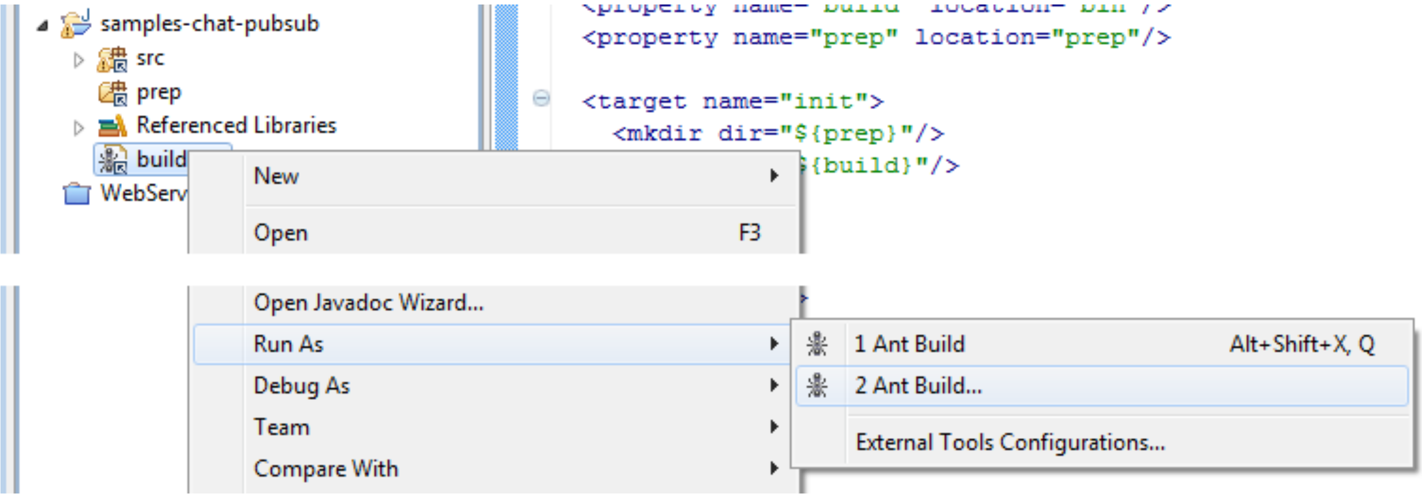


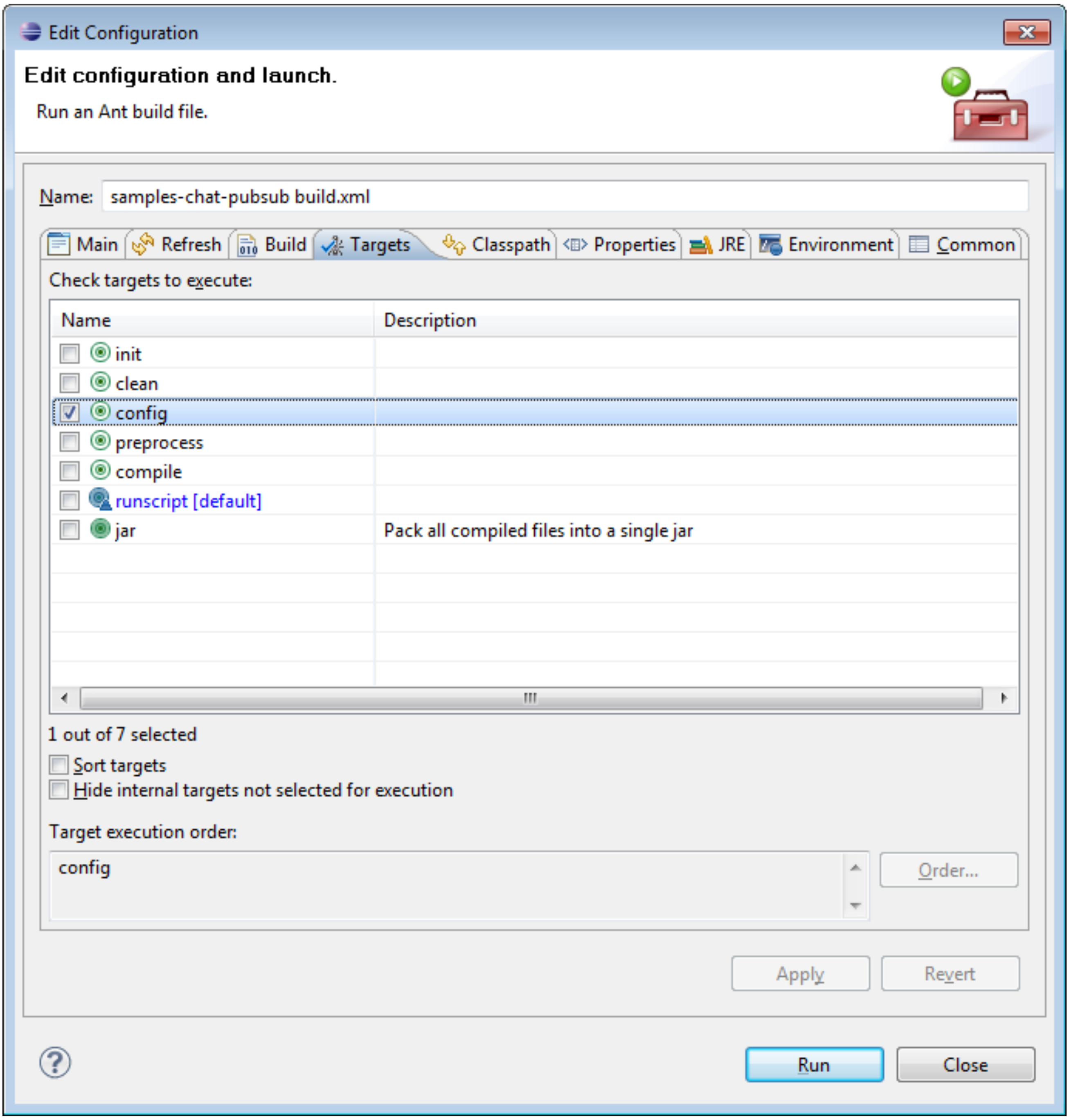
After you have generated the run scripts, start two instances of the chat program on the same host using `run.bat` in `samples/chat/pubsub` using Windows Explorer.

If you type a line of text (terminated by pressing Enter), the text should also appear as output of the other instance. To quit the program, type `.` and Enter at the beginning of a line.

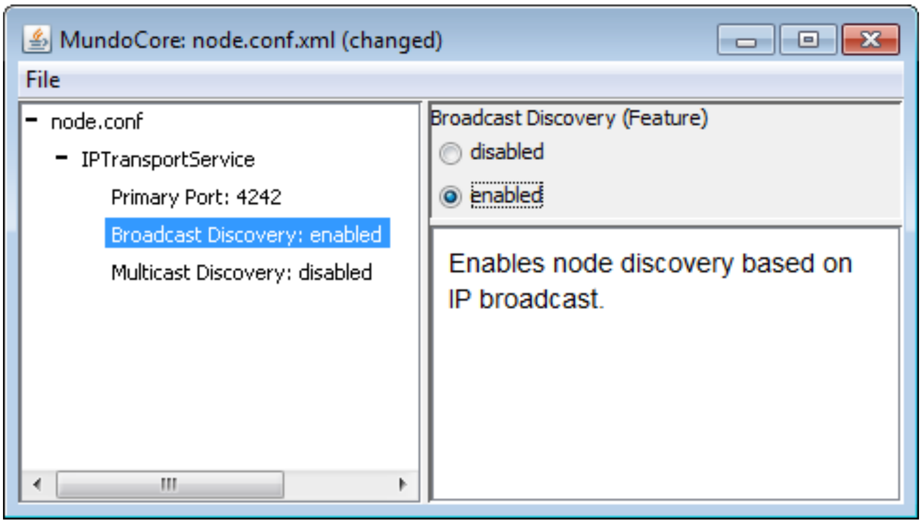
Node Configuration

Initially, MundoCore nodes will only communicate within the local host. To enable communication over the network, a configuration file `node.conf.xml` must be created and a node discovery method must be enabled. To create a configuration file, invoke the `config` target of `build.xml`:



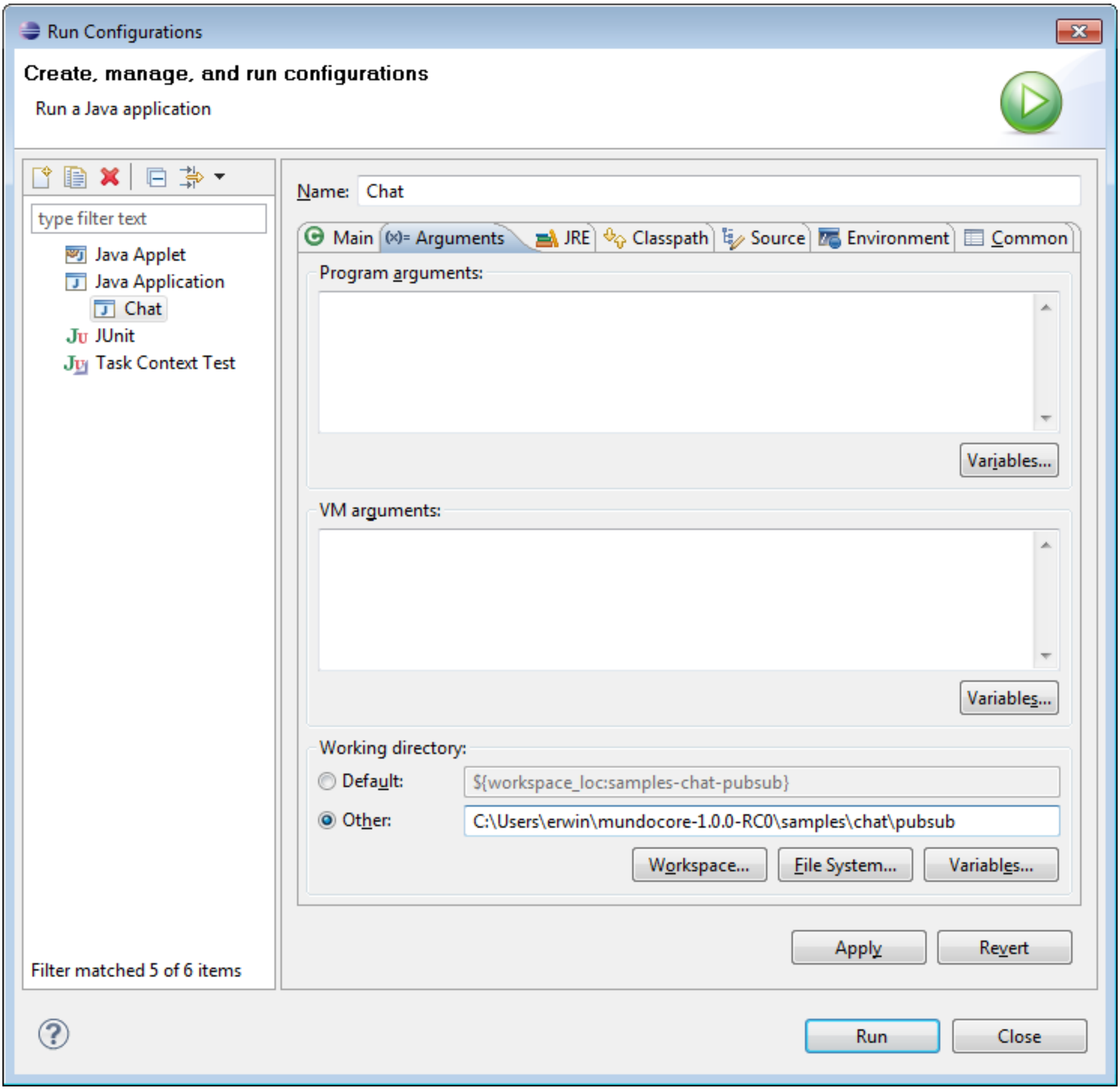


This will bring up the configuration UI. Enable broadcast discovery and save the configuration file as `node.conf.xml` into the directory of the example `samples/chat/pubsub`:



With this configuration file, the sample program can be run on two different hosts in the local network. (Please make sure that MundoCore connections and discovery packets are not blocked by personal firewalls!)

When you are running the program using the **Run** function of Eclipse, you must make sure that the program can find its `node.conf.xml` configuration file. In **Run Configurations** set the working directory to `.../samples/chat/pubsub`:


Run Configurations
Create, manage, and run configurations
Run a Java application
type filter text
Java Applet
Java Application
Chat
JUnit
Task Context Test
Filter matched 5 of 6 items
Name: Chat
Main
Arguments
JRE
Classpath
Source
Environment
Common
Program arguments:
Variables...
VM arguments:
Variables...
Working directory:
Default:
${workspace_loc:samples-chat-pubsub}
Other:
C:\Users\erwin\mundocore-1.0.0-RC0\samples\chat\pubsub
Workspace...
File System...
Variables...
Apply
Revert
Run
Close

Creating a New Project

Create a new directory:

```
mkdir myprj
cd myprj
```

Create a directory for the source code and the main class:

```
mkdir src
touch src/Main.java
```

Create a buildfile from one of the buildfile templates. Replace `$mcroot` with the directory where MundoCore is installed.

```
cp $mcroot/samples/buildfiles/standard-edition-application/build.xml .
```

Copy the `build.properties` file with the path settings to the local directory:

```
cp $mcroot/config/build.properties .
```

Edit the buildfile `build.xml`:

```
<project name="NONAME" default="runscript" basedir=".">
  <property file="build.properties"/>
  <property name="main-class" value="MAINCLASS"/>
  ...
```

Now, change the *name of the project* and the *name of the main class*, e.g., to:

```
<project name="MyProject" default="runscript" basedir=".">
  <property file="build.properties"/>
  <property name="main-class" value="Main"/>
  ...
```

Now the build environment is set up. Running `ant` will now build the project and create runscripts.

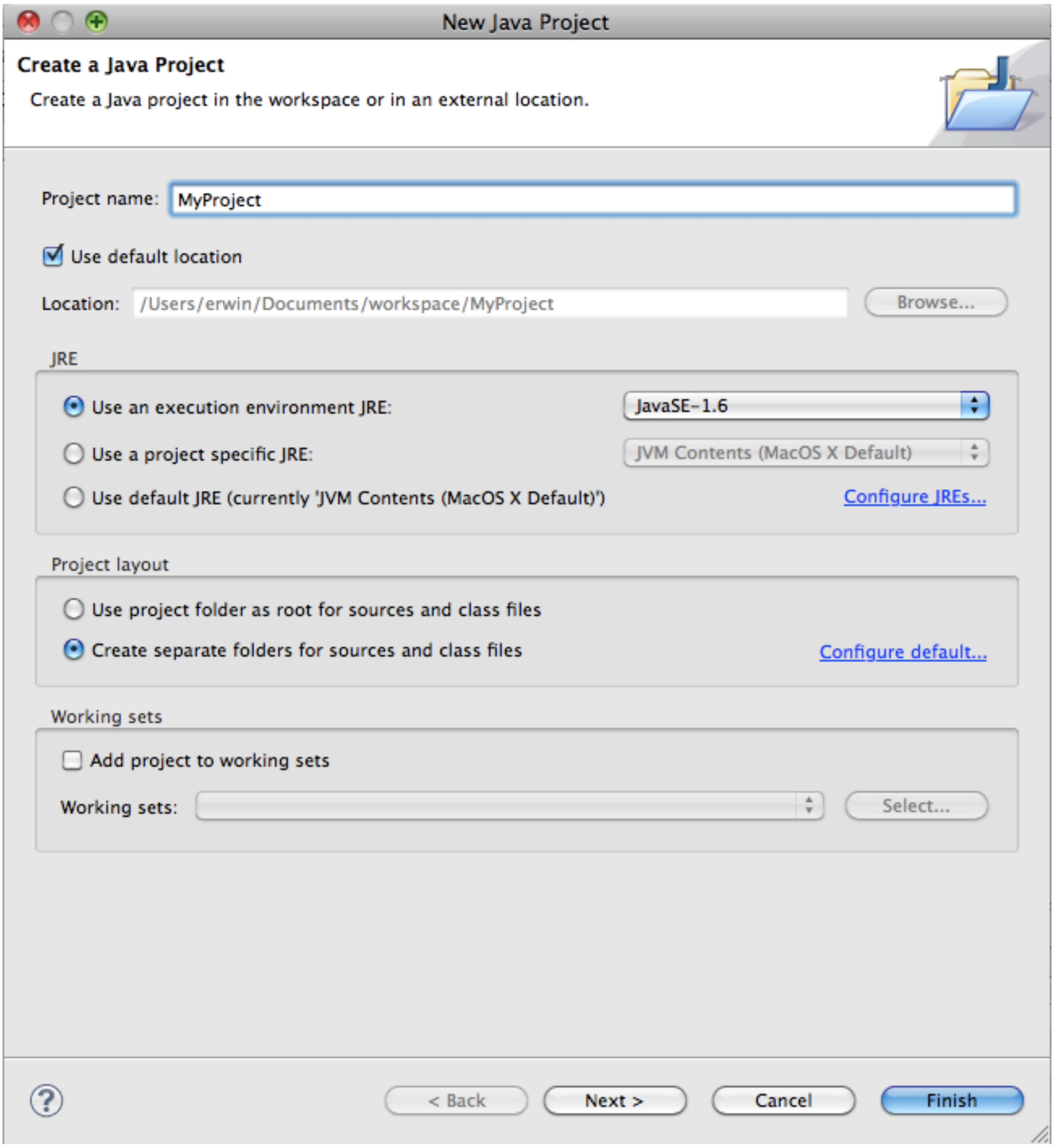
Creating a New Project in Eclipse

This tutorial shows how to create a new project in Eclipse with the correct build and configuration files.

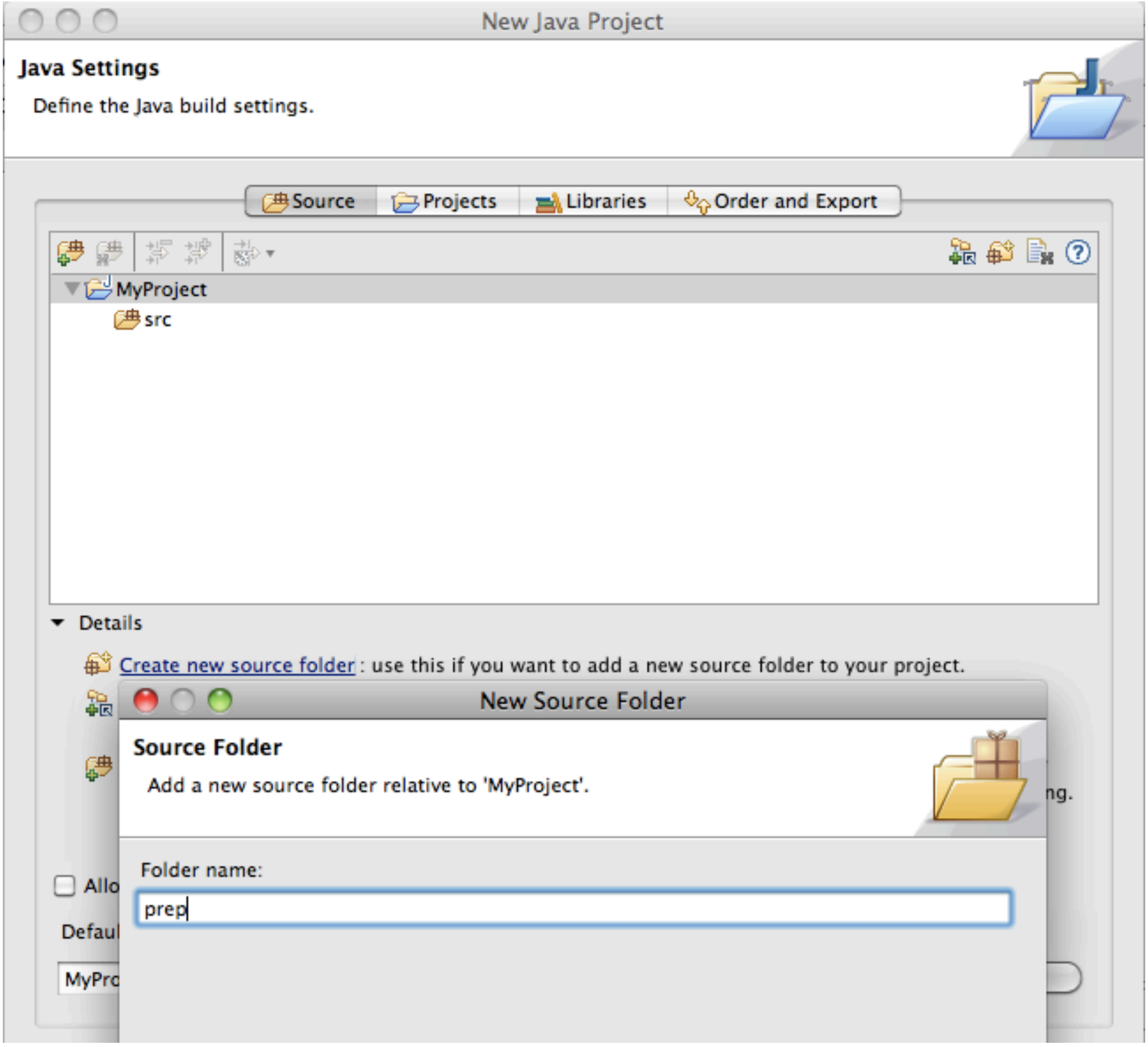
Creating Eclipse Projects

To create a project in Eclipse, perform the following steps:

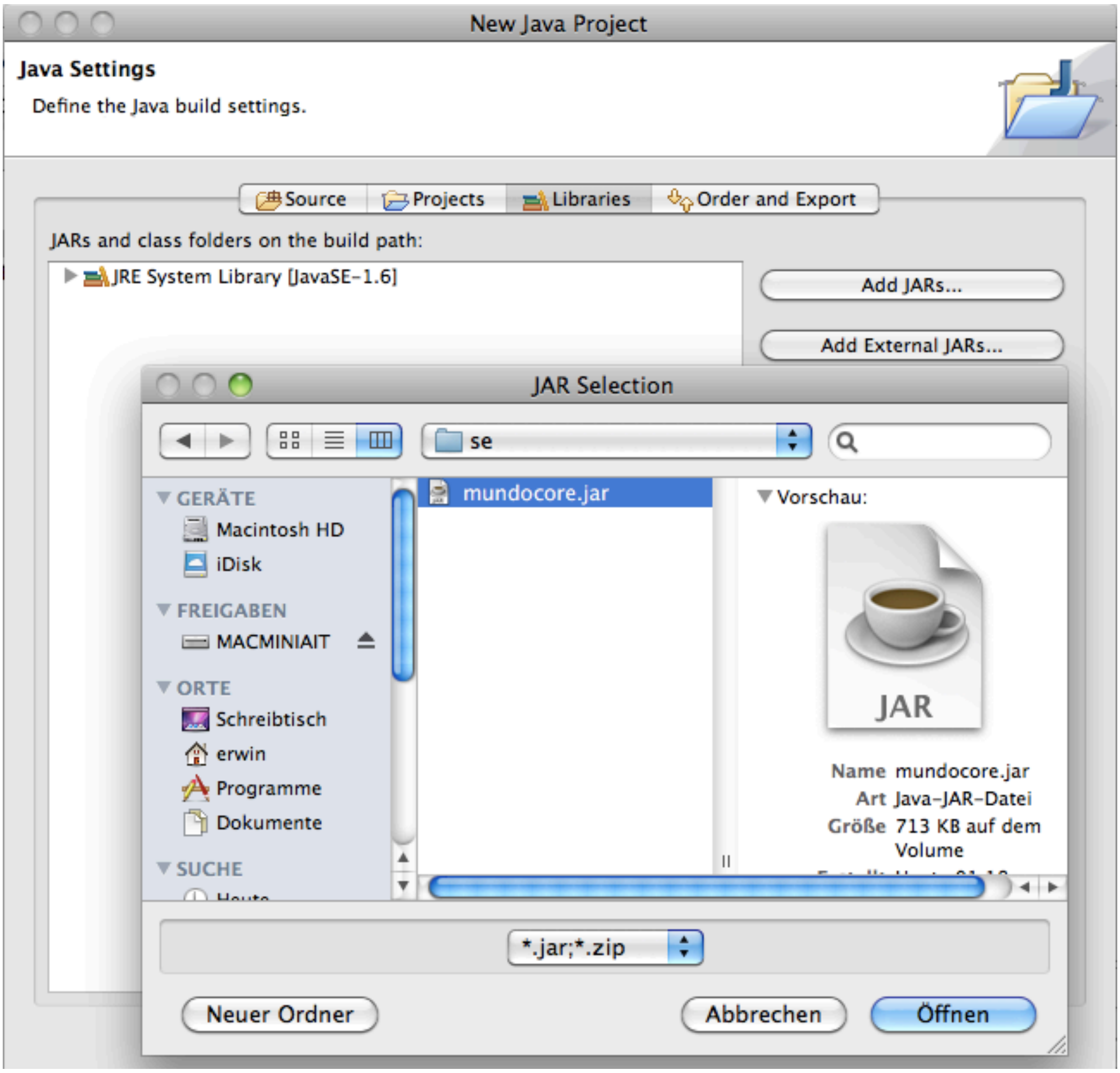
Create a new *Java Project*. Make sure that you select: *Create separate source and output folders*:



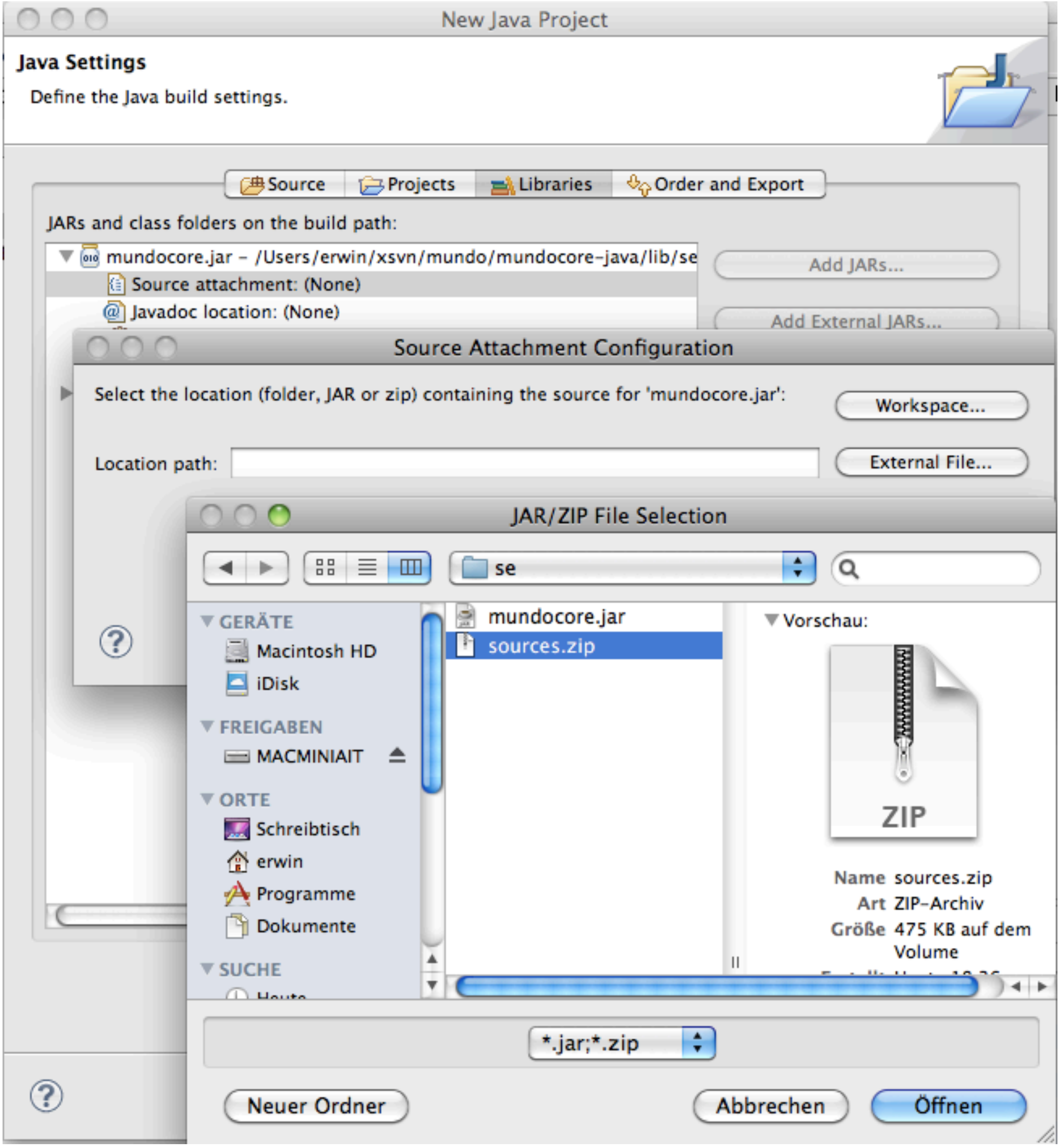
In the Source tab, select *Create new source folder* and name it `prep`:



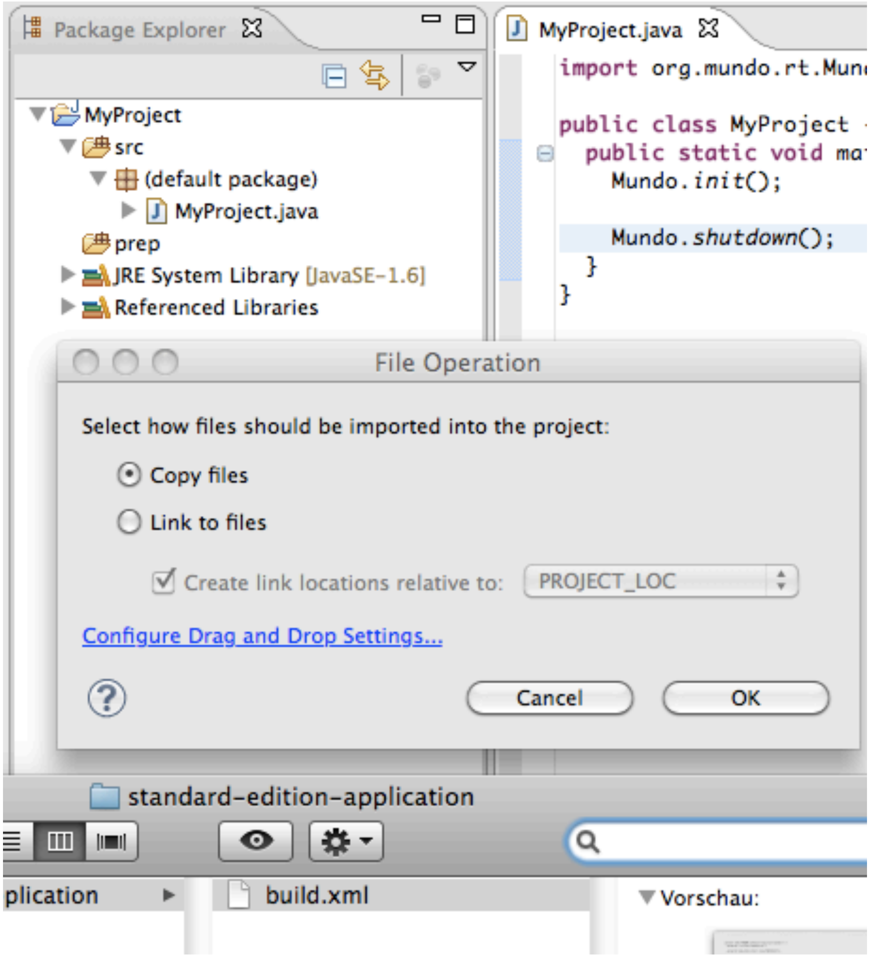
In the Libraries tab, select *Add External JARs*. Now add `lib/mundocore.jar`:



Then expand the new entry, select *Source Attachment*, click *Edit*, and select `sources.zip` from the MundoCore distribution:



Copy a buildfile template (e.g., `samples/buildfiles/standard-edition-application/build.xml`) into the main project directory. You can use drag&drop, then select *copy files*:



Also copy `build.properties` from the `config` folder to the main project directory.

Now open `build.xml` and edit a few settings:

```
<project name="NONAME" default="runscript" basedir=".">
  <property file="build.properties"/>
  <property name="main-class" value="MAINCLASS"/>
  ...
```

Change the *name of the project* and the *name of the main class*, e.g., to:

```
<project name="MyProject" default="runscript" basedir=".">
  <property file="build.properties"/>
  <property name="main-class" value="Main"/>
  ...
```

Now the build environment is set up. You can start the build process by selecting `build.xml` and *Run As > Ant Build*.

Notes

- If you are building primarily with the compiler embedded in Eclipse, you should also change the default build target to `preprocess`. Running `build.xml` then only invokes the preprocessor and no other build steps. However, you may want run `build.xml` with target `runscripts` at least once, to get the run scripts for your application.
- It is not necessary to run the preprocessor each and every time the program is compiled. However, it is important to re-run the preprocessor when metadata information, serializable classes, or remote interfaces change.

Publish/Subscribe

A publish/subscribe system consists of *publishers*, *subscribers*, and an *event service*. Publishers produce event notifications and pass them to the event service. Subscribers express their interest in certain events by defining stateless message filters, called *subscriptions*, and issue them to the event service. The event service is a message broker responsible for distributing messages arriving from multiple publishers to its multiple subscribers.

Most services use the channel-based publish/subscribe abstraction of MundoCore for communication. This example shows how to use the publish/subscribe system at the lowest API level, i.e., directly working with passive message objects.

In the following example, all instances of the chat client use the channel `chattest` for communication. The zone name `lan` defines the local area network as scope for the channel.

Sending messages

Before we can send messages, we have to **advertise** to which channel we plan to publish. This is done by creating a `Publisher` object with the following call:

```
public boolean init() {
  ...
  publisher = getSession().publish("lan", "chattest");
  ...
}
```

Now, our chat application can read user input from `System.in`, create a `Message` object, and send it:

```
BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
String ln;
while ( (ln=r.readLine())!=null && !ln.equals(".") ) {
  TypedMap map = new TypedMap();
  map.putString("ln", ln);
  publisher.send(new Message(map));
}
```

Receiving messages

To receive messages, we **subscribe** to the corresponding channel. This is done by creating a `Subscriber` object with the following call:

```
public boolean init() {
  ...
  subscriber = getSession().subscribe("lan", "chattest", this);
  ...
}
```

Once a message is received that matches the subscription, the callback-method `received` is called:

```
public void received(Message msg, MessageContext ctx) {
  System.out.println(msg.getMap().getString("ln"));
}
```

Sessions

As shown above, **publisher** and **subscriber** objects are obtained from the **session** object. The session concept provides the following functionalities:

- In each session, messages are delivered sequentially to the `received` methods, and never concurrently. Hence, sessions implement a synchronization concept.
- There is no message loopback in the same session: If a subscriber S subscribes to a channel C and a publisher P publishes to a channel C, and P and S are from the same session, then S will not receive messages sent by P.
- Every service comes with a default session. It is possible to create additional sessions for a service, but this is barely needed. For example, advanced routing/brokering services need additional sessions.

Putting everything together

Some additional code is needed to set up and shut down the node properly, and to create and register the Chat service. The `main`-method of the program is shown below:

```
public static void main(String args[]) {
  Mundo.init();

  ChatService cs = new ChatService();
  Mundo.registerService(cs);
  cs.run();

  Mundo.shutdown();
}
```

`Mundo.init` must be run at the beginning of the program. It starts up basic services that provide e.g. discovery, message transport and message routing.

Before the program terminates. `Mundo.shutdown` should be called whenever possible. When the basic services are shut down properly, they tell our neighbour nodes that we are shutting down and are then no longer available. If a node does not shut down properly, other nodes can not be sure if the node crashed or the network link is just down temporarily and will try to reconnect a few times. This way, shutting down nodes properly will reduce network traffic.

Running Chat

Now, you can run multiple instances of `Chat` on the same machine. The processes will automatically discover each other and distribute chat messages typed in to all other processes.

You may have noticed that the shutdown sometimes takes several seconds. Because MundoCore guarantees the delivery of messages also in presence of some network propagation delays, the routing service does not shut down before all messages to export have expired.

The full Chat program

`samples/chat/pubsub/src/Chat.java`

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import org.mundo.rt.IReceiver;
import org.mundo.rt.Message;
import org.mundo.rt.MessageContext;
```

```
import org.mundo.rt.Mundo;
import org.mundo.rt.Publisher;
import org.mundo.rt.Service;
import org.mundo.rt.TypedMap;

class ChatService extends Service implements IReceiver {
  private Publisher publisher;

  ChatService() {
  }
  public void init() {
    publisher = getSession().publish("lan", "chattest");
    getSession().subscribe("lan", "chattest", this);
  }
  public void run() {
    try {
      BufferedReader r=new BufferedReader(new InputStreamReader(System.in));
      String ln;
      while ( (ln=r.readLine())!=null && !ln.equals(".") ) {
        TypedMap map=new TypedMap();
        map.putString("ln", ln);
        publisher.send(new Message(map));
      }
    }
    catch (Exception x) {
      x.printStackTrace();
    }
  }
  public void received(Message msg, MessageContext ctx) {
    System.out.println(msg.getMap().getString("ln"));
  }
}

public class Chat {
  public static void main(String args[]) throws Exception {
    Mundo.init();
    ChatService cs = new ChatService();
    Mundo.registerService(cs);
    cs.run();
    Mundo.shutdown();
  }
}
```

Sending and receiving messages on the same channel

According to the definition of publish/subscribe, a program should receive all messages it subscribed to that were published. Usually, this would also include messages published by the program itself. You may have noticed that SimpleChat? program does have this behaviour and does not not echo back the messages you sent on the `chattest` channel, even though you subscribed to the channel as well.

This is caused by a feature of MundoCore that automatically depresses messages you sent. Usually this is also the behaviour desired by the programmer. For those rare cases when you actually want to get a notification for messages you sent as well, you must enable this explicitly by calling the Publisher's `enableLocalLoopback` method.

A note on channel naming

Channels are identified by names, that have to be defined by the application programmer. The concept of zones is used to create separate namespaces to limit the visibility and accessibility of channels to certain domains. For point-to-point-links, channels are often named simply by using GUIDs. When writing applications, you typically want to define some access points to your application by using well-defined channel names, other temporary links can be assigned randomly with GUIDs. If you have no idea for a well-defined channel name, a reasonable choice is the package and/or class name you publish your channel in.

Object Serialization

With *serialization* it is possible to send Java objects to remote peers or write objects to files. For example, objects can be transformed into XML documents and vice versa, usually without the need to write any additional code. The methods that perform the actual data conversions are automatically generated by `mcc`.

In contrast to several other frameworks, this conversion is a two-step process:

1. First, the *active object graph* to be serialized is converted to a *passive data structure*. The passive structure can only contain base types, arrays, and maps. Consequently, this representation is programming language-independent. The conversion from active to passive objects is performed by the method *passivate*, while the conversion from passive to active objects is performed by the method *activate*. The implementations of these methods are automatically generated by the precompiler *mcc*. The process of converting from the active to the passive representation is also referred to as *externalization*.
2. The passive structure is then serialized to XML, JSON, or binary formats.

This two-step process for serialization provides better modularization and allows to employ additional transformation and filtering steps on the passive objects, like used by content-based publish/subscribe.

Using Annotations and `mcc` for Serialization

The following example uses a custom class to encapsulate the message and shows how to generate externalizers. The source code of `ChatMessage.java` looks as follows:

```
import org.mundo.annotation.*;

@mcSerialize
public class ChatMessage {
  public String text;

  public ChatMessage() {
  }
  public ChatMessage(String t) {
    text=t;
  }
}
```

Note:

- Fields declared as `transient` are not serialized.
- The visibility of fields to serialize must be more than `private`. Fields do not have to be declared `public` or `protected`, though, it is sufficient to use *package private* visiblity by not specifying any visibility modifier at all.
- Any class with `@mcSerialize` must have a `public` nullary (empty) constructor (Here: `public ChatMessage?()`). Otherwise the framework will not be able to create instances of the class during deserialization.

MundoCore uses metaclasses for externalization, which are generated by the preprocessor `mcc`. `mcc` is usually invoked from the buildfile, such as:

```
<target name="preprocess" depends="init">
  <apply executable="${mcc}" parallel="true">
    <arg value="-O${prep}" />
    <arg value="-x" />
    <fileset dir="${src}">
```

```
      <include name="*.java" />
    </fileset>
  </apply>
  <copy file="${prep}/metaclasses.xml" todir="${build}" failonerror="false"/>
</target>
```

The complete buildfile can be found in the directory of this example: `samples/chat/serialization/build.xml`.

The `SerChat` Program

The `SerChat` program is very similar to the `SimpleChat` program. Sending messages changes to:

```
ChatMessage cm = new ChatMessage(text);
publisher.send(Message.fromObject(cm));
```

and the method for receiving messages changes to:

```
public void received(Message msg, MessageContext ctx) {
  ChatMessage cm = (ChatMessage)msg.getObject();
  System.out.println(cm.text);
}
```

The Full `SerChat` Program

`samples/chat/serialization/src/SerChat.java`

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import org.mundo.rt.IReceiver;
import org.mundo.rt.Message;
import org.mundo.rt.MessageContext;
import org.mundo.rt.Mundo;
import org.mundo.rt.Publisher;
import org.mundo.rt.Service;

class ChatService extends Service implements IReceiver {
  private Publisher publisher;

  ChatService() {
  }
  public void init() {
    publisher = getSession().publish("lan", "serchattest");
    getSession().subscribe("lan", "serchattest", this);
  }
  public void run() {
    try {
      BufferedReader r=new BufferedReader(new InputStreamReader(System.in));
      String ln;
      while ( (ln=r.readLine())!=null && !ln.equals(".") )
        publisher.send(Message.fromObject(new ChatMessage(ln)));
    }
    catch(Exception x) {
```

```
      x.printStackTrace();
    }
  }
  public void received(Message msg, MessageContext ctx) {
    try {
      System.out.println(((ChatMessage)msg.getObject()).text);
    }
    catch(Exception x) {
      x.printStackTrace();
    }
  }
}

class SerChat {
  public static void main(String args[]) throws Exception {
    Mundo.init();
    ChatService cs = new ChatService();
    Mundo.registerService(cs);
    cs.run();
    Mundo.shutdown();
  }
}
```

Remote Method Calls

A remote method call allows to call methods on remote services, with the abstraction of a local method call. In the following, an implementation of the chat example based on remote method calls is discussed.

RMC Interfaces

Remote method calls rely on additional client and server stub classes that are generated by the precompiler. The metadata tag `@mcRemote` indicates that the precompiler should generate client and server stub classes for the following class or interface. It is usually preferable to generate stubs for interfaces instead of classes, because this allows to have the server-side implementation interchangeable.

For the Chat-service, we define the following interface:

```
@mcRemote
public interface IChat {
  public void chatMessage(String msg);
}
```

The Server Side

As server-side implementation we define a `Service` that implements the interface `IChat`. To receive chat messages, an instance of this service has to be connected to a *channel*. This way, other peers can send us messages.

To export an object on the server side, a subscriber must be connected to the server object:

```
Subscriber sub = getSession().subscribe("lan", "chat_rmc");
Signal.connect(sub, this);
```

The service now has to implement the method `chatMessage`, as defined in `IChat`. It can print the received message to the console:

```
public void chatMessage(String msg) {
  System.out.println(msg);
}
```

The client side

A remote object can now be accessed by creating a client stub object and connecting the stub to a publisher:

```
DoChatService stub = new DoChatService();
Publisher pub = getSession().publish("lan", "chat_rmc");
Signal.connect(stub, pub);
```

Sending Messages

Remote method calls can return a value or throw an Exception, just like regular methods. To make this behaviour work, MundoCore usually suspends your program at the method call until it gets a response from the remote method. This is called a *blocking call* and is also a behavior similar to regular method calls.

It is however possible, to use other call semantics with MundoCore. After all, if you don't expect an answer from a method invoked on another machine, why should you wait for that? In this example, we therefore use *one-way-calls* to publish chat messages:

```
stub.chatMessage(ln, stub.ONEWAY);
```

The `chatMessage` method that takes the additional option parameter is generated by `mcc`. Another reason why it is necessary to use one-way-calls in this example is because blocking RMC-calls would not work with more than two clients (i.e., one client and more than one server), because each server would send a response message to the client. The client, ohn the other hand, only expects one such response.

The full `RMCChat` program

`samples/chat/rmc/src/RMCChat.java`

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import org.mundo.rt.Mundo;
import org.mundo.rt.Service;
import org.mundo.rt.Signal;

class ChatService extends Service implements IChat {
  private static final String CHANNEL_NAME = "chat_rmc";
  private static final String ZONE_NAME = "lan";
  DoIChat doChat;

  public ChatService() {
  }
  public void init() {
    try {
      // connect channel to this object to receive chat messages
      Signal.connect(getSession().subscribe(ZONE_NAME, CHANNEL_NAME), this);

      // connect DoIChat stub to channel to send chat messages
      doChat = new DoIChat();
      Signal.connect(doChat, getSession().publish(ZONE_NAME, CHANNEL_NAME));
    }
    catch(Exception x) {
      x.printStackTrace();
    }
  }
  public void run() {
    try {
      BufferedReader r=new BufferedReader(new InputStreamReader(System.in));
      String ln;
      while ( (ln=r.readLine())!=null && !ln.equals(".") )
        doChat.chatMessage(ln, doChat.ONEWAY);
    }
    catch (Exception x) {
      x.printStackTrace();
    }
  }
  public void chatMessage(String msg) /*IChat*/ {
    System.out.println(msg);
  }
}
```

```java
public class RMCChat {
  public static void main(String args[]) {
    Mundo.init();
    ChatService cs=new ChatService();
    Mundo.registerService(cs);
    cs.run();
    Mundo.shutdown();
  }
}
```

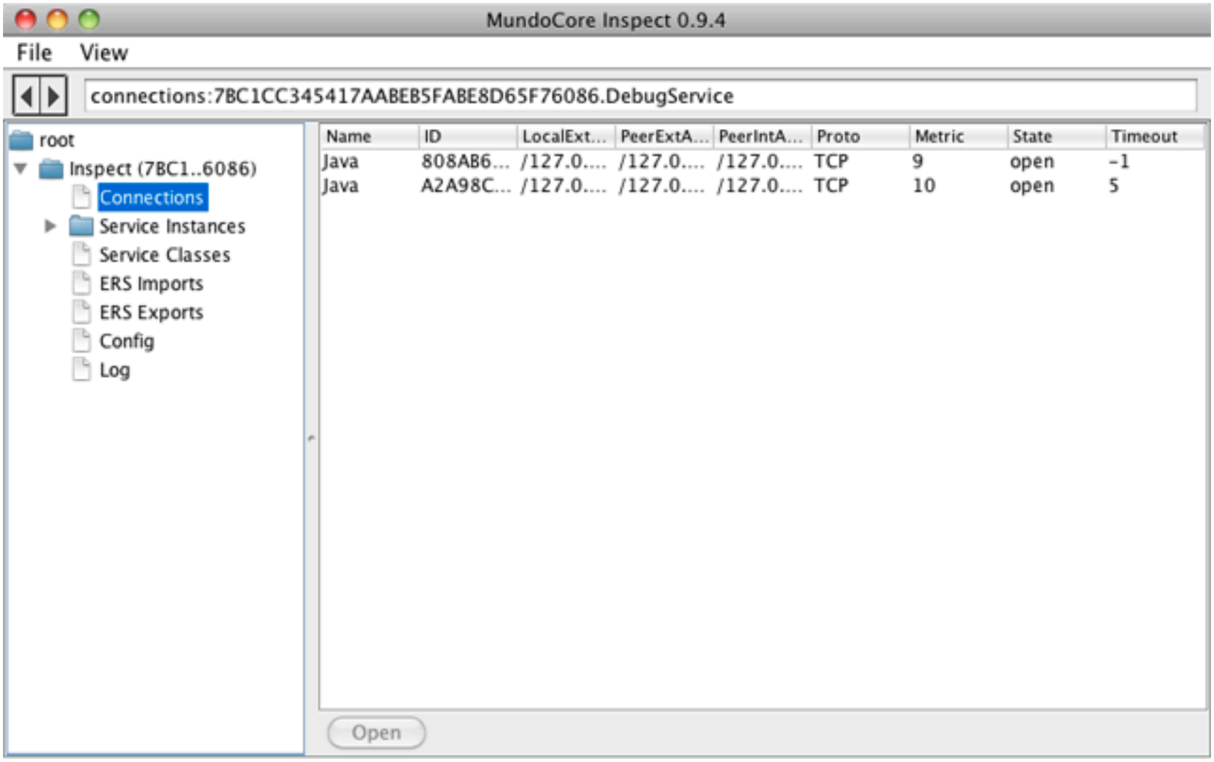
Inspect

Inspect is a tool with a graphical user interface to monitor and manage local or remote MundoCore nodes. It can be started as follows:

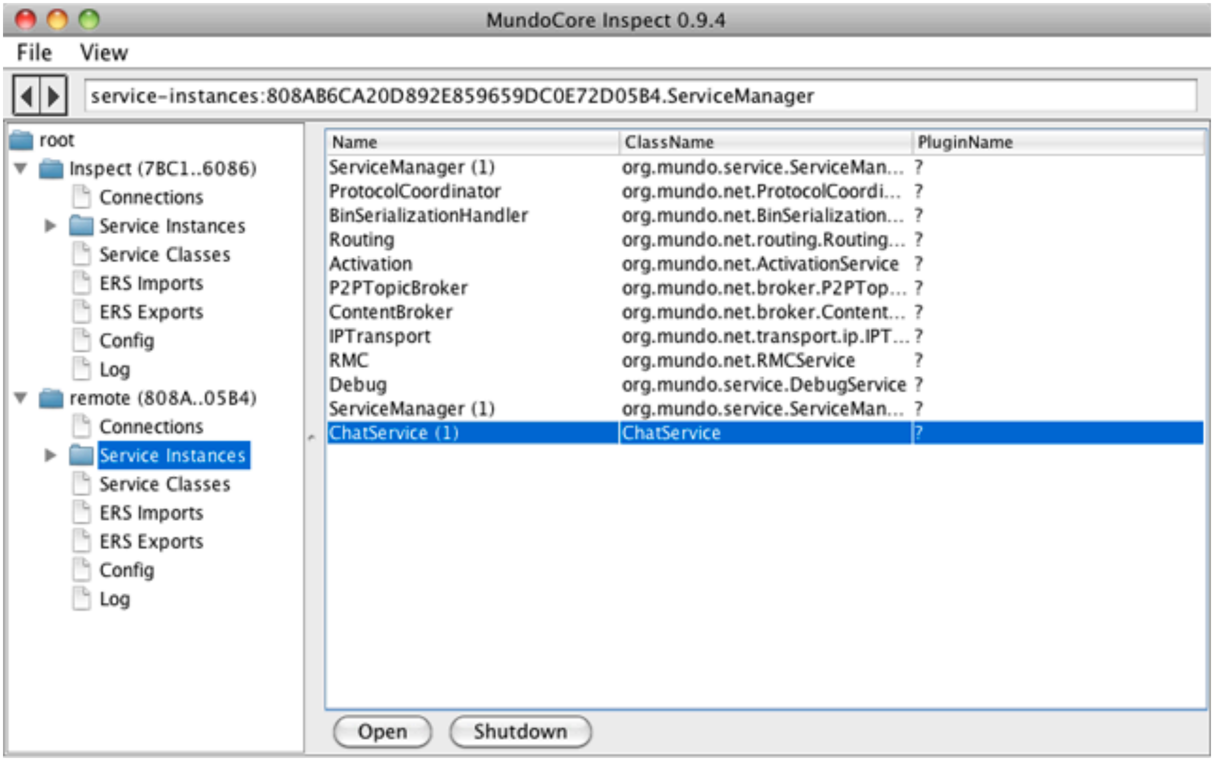
```
java -jar tools/inspect.jar
```

Using Inspect

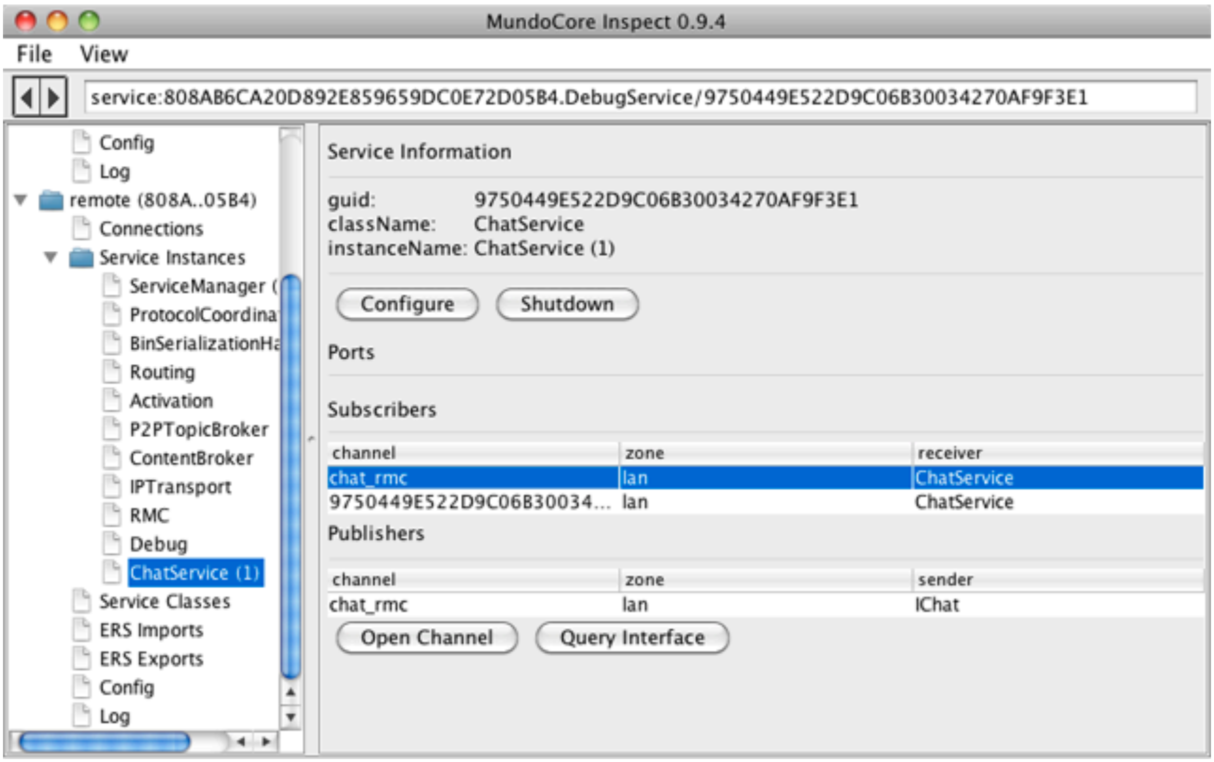
The *connections* view lists all adjacent nodes, to which Inspect has a direct communication link to. When two instances of `RMCChat` are running beside Inspect, the *connections* view will appear as follows:



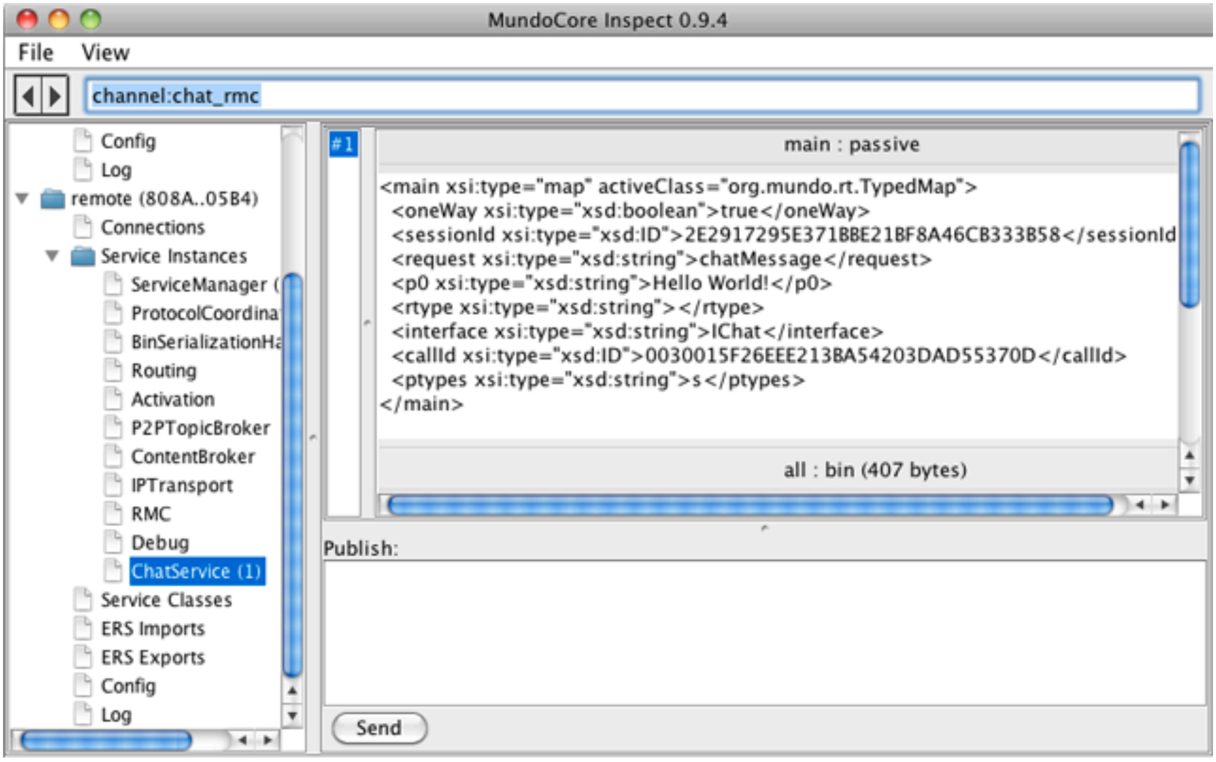
In this view, we can select an entry and click `Open`. An additional subtree appears in the left pane, which allows us to access various information about one of the `RMCChat` nodes. Selecting *Service Instances* gives a list of the services running on the remote node. The list contains several core services and the main service of `RMCChat`:



The service `ChatService` can now be selected to access detailed information about this service:



This view shows detail information about the service and its *publishers* and *subscribers*. To monitor the messages exchanged over a channel, we can select a channel in the list and then click *Open Channel*. Now, when a message is typed into an instance of `RMCChat`, this produces the following output:



Messages in MundoCore consist of multiple chunks. The first chunk is always structured (here: XML). The other chunks may be structured as well or in binary format.

Content-based Publish/Subscribe

With content-based subscriptions a client does not subscribe to a topic (a channel) but specifies the content of messages the client is interested in. The client specifies **filters** that are matched with messages that are send by services. Only messages that are **covered** by the filters are delivered to the clients. Consider a `Thermometer` service that send messages containing the actual temperature. A client could specify a filter to receive only temperatures that are greater or less a certain value.

Building Filters with MCC

Filters can automatically be build from Mundo-serializable objects by means of the MCC. Classes have to be tagged with `@mcSerialize` and `@mcFilter`. MCC creates a new class whereas the name of the new class is the name of the source class file with the string `Filter` appended. When the source file was `Temperature.java` the name of filter class is `TemperatureFilter.java`. The filter class is derived from the source class and contains for each `field` of the source class an additional field `_op_field`. This `op` field contains the operator for comparison and the inherited field contains the value that messages should be compared with. The listing shows an example for a `Temperature` event.

```
package thermometer;

import org.mundo.annotation.*;

@mcSerialize
@mcFilter
public class TemperatureEvent {
   public long temp;

   public TemperatureEvent() {
      temp = 0;
   }
}
```

The MCC creates a filter as the following listing shows.

```
// This file was generated by mcc. DO NOT EDIT THIS FILE!
// classname: thermometer.TemperatureEvent
package thermometer;
public class TemperatureEventFilter
        extends TemperatureEvent
        implements org.mundo.net.IFilter
{
  public int _op_temp = OP_IGNORE;

  public org.mundo.net.TypedMapFilter _getFilter()
  {
    org.mundo.net.TypedMapFilter f=new org.mundo.net.TypedMapFilter();
    f.putLong("temp", _op_temp, temp);
    return f;
  }
  public String toString()
  {
    return _getFilter().toString();
  }
}
```

A filter that filters on temperature less than 0 is created by

```
TemperatureEventFilter fn = new TemperatureEventFilter();
fn._op_temp = IFilterConstants.OP_LESS;
fn.temp = 0;
```

Classes with several fields

Classes with more than one field get an individual operator per field. All operators have to match (AND) whereas operators with the value `OP_IGNORE` are ignored.

Nested Filters

The examples so far only describe Java built-in data types. A class also can contain user defined (complex) classes. With nested filters it is possible to filter on the contend of these classes.

Consider a scenario where `Person` classes contain an `Address` field as shown in the following listings.

```
import org.mundo.annotation.*;

@mcFilter
@mcSerialize
public class Address
{
  public String  street;
  public int      zip;
  public String  city;

  public String toString(){
   return "Address: " + street + ", " + zip + " " + city;
  }
}

import org.mundo.annotation.*;

@mcFilter
@mcSerialize
public class Person
{
  public String   firstname;
  public String   lastname;
  public Address  address;

  public String toString(){
   return "Person: " + firstname + " " + lastname + ", " + address;
  }
}
```

With nested filtering it is possible to also filter on person objects with particular address objects. There are only two important things:

- the nested filter has to on the right place in the object tree
- the `_op_` has to be set to `IFilterConstants.OP_FILTER`.

Point 1 means in our example that the `AddressFilter` has to be put into a `PersonFilter` to be able to filter on the address of persons. If the `AddressFilter` is applied directly to the Content Subscription, we only receive messages containing addresses but not messages containing persons with links to addresses.

Point 2 has to be done because the systems does not recognize filters automatically.

The code for our example looks like this:

```
PersonFilter pf=new PersonFilter();    // Look for persons
pf.firstname="Erwin";                   // with firstname ``Erwin''
pf._op_firstname=OP_EQUAL;

AddressFilter af=new AddressFilter(); // who lives
af.zip=64289;                          // in city with zip 64289
af._op_zip=OP_EQUAL;

pf.address=af;                         // nest filters
pf._op_address=OP_FILTER;              // and tell system that address contains a filter
```

Filter using XQuery

This is an advanced topic. Specifying filters using XQuery is an alternative to Java filters. Because XQuery works on the externalized objects it is recommended to read the chapter on serialization first. The usage is straight forward if you know content-based filtering. Create object from `XQuery` class. Call method `parse(string)` where the string contains a XQuery expression. Subscribe using the XQuery.

```
public boolean init() {
  ...
  XQuery xq = new XQuery();

  try {
     xq.parse("for $o in $msg/object where $o/firstname='XYZ' " +
              "and $o/address/city='Darmstadt'");
     subscriber = ContentSubscription.subscribe(getSession(),xq.getMapFilter());
     subscriber.setReceiver(IReceiver);
     subscriber.enable();
  } catch (Exception e){
     e.printStackTrace();
  }
  ...
}
```

Sending messages

Before we can send messages, we have to **advertise** that we publish content-based. This is done by creating a `Publisher` object with the following call:

```
public boolean init()
{
  ...
  publisher = ContentSubscription.publish(getSession());
  ...
}
```

Now, our the application can send messages as described in previous chapters of the tutorial.

```
publisher.send(Message.fromObject(evnt));
```

Receiving messages

To receive messages, we have to define a filter and then **subscribe** with this filter. **Attention:** There is one peculiarity that has to be taken into account. The filter has to be put into a `TypedMapFilter` with the key `object`. This TypedMapFilter? than can be registered with the `ContentSubscription` as shown in listing.

This is done by creating a `Subscriber` object with the following call (whereas fn is a (nested) filter object):

```
public boolean init()
{
  ...
  TypedMapFilter() tmf = new TypedMapFilter();
  tmf.putObject("object", OP_FILTER, fn);
  Subscriber subscriber = ContentSubscription.subscribe(getSession(), tmf);
  subscriber.setReceiver(IReceiver);
  subscriber.enable();
  ...
}
```

Once a message is received that is **covered** by the subscription, the callback-method `received` is called by the system.

Custom filtering

Sometimes it is necessary to implement filtering with special semantics that can not be expressed with standard filters. Therefore it is possible to implement custom filters. The method explained in this section still works on passive (aka. externalized) objects. A custom filter extended from `AttributeFilter`. Several methods have to be overwritten:

- `public abstract boolean covers(java.lang.Object o)`: tests if this attribute filter covers the specified attribute.
- `public abstract java.lang.Object getValue()`: returns the comparison value.
- `public int hashCode()` and
- `public boolean equals(java.lang.Object o)`

Further the filter may define custom operators. The following listings show essential parts of an custom filter implementation.

```
public class MyBirthdayFilter extends AttributeFilter  {
   /** A decent set of operator should be defined in an interface class... */
   public static final int OP_AFTER    = 0x50;
   public static final int OP_BEFORE   = 0x51;
     /** Implementation compares to this date...*/
   Date temp;

   /**
    * Because I was lazy I do not pass a Birthday object
    * but a Date. You see, with you own implementation
    * you can do whatever you want ;-)
    */
   public MyBirthdayFilter (int op, Date compareToDate){
```

```
        int m=(op & MASK_OP);
        if (m!=OP_IGNORE && m!=OP_AFTER && m!=OP_BEFORE)
        throw new IllegalArgumentException("invalid operator");
        if (compareToDate == null)
        throw new IllegalArgumentException("person must not be null");
         this.op = op;
         temp = compareToDate;
    }

    /**
     * Implements our sematic for the filter.
     */
    public boolean covers(Object obj)
    {
        /**
         * obj is passive object. Because we know what object we expect
         * we activate it and work on the object.
         * Is is also possibe to work on the nested typedmap.
         */
        MetaBirthday mb = new MetaBirthday();
        Birthday bi = new Birthday();
        try {
        mb.activate(bi, (TypedMap)obj, null);
        } catch (Exception e){
        return false;
        }

         boolean b;
         switch (op & MASK_OP) {
         case OP_BEFORE:
            b= bi.getDate().compareTo(temp) < 0;
            break;
         case OP_AFTER:
            b=bi.getDate().compareTo(temp) > 0;
            break;
         case OP_IGNORE:
            b=true;
            break;
         default:
            throw new IllegalStateException("invalid operator");
         }
         if ((op & OP_NOT)>0)
         b=!b;
         return  b;
    }
}
```

Active object filtering

The filter mechanism so far works on externalized objects. Advantages of this approach are that it can be efficiently implemented and there is much support through MundoCore and mcc to create filters. In most cases there should be no need for active object filtering (also called active filters) as described in this section. Active object filtering helps overcoming two issues: lack of custom operators and impossibility to filter on objects that can not be Mundo-serialized.

Active object filtering differs in several ways compared to normal filtering:

- active object filters implement the `org.mundo.net.IActiveObjectFilter` interface,
- programmer has to handle nested objects,
- active object filters are evaluated on the client.

The interface `IActiveObjectFilter` has one method `boolean covers(java.lang.Object obj)`. The implementation of the method contains the logic whether the `obj` is covered by the filter ("is interesting to the client") or not. The following listing demonstrates the implementation of a filter that can compare the birthdate of a Person to a given date.

```
package ga.tests.content;

import org.mundo.net.IActiveObjectFilter;

public class MyPersonFilter implements IActiveObjectFilter {
 // A decent set of operator should be defined in an interface class...
 public static final int OP_AFTER       = 0x50;
 public static final int OP_BEFORE      = 0x51;
 int op;
 Person2 person;

 public MyPersonFilter (){
 this(OP_IGNORE, new Person2());
 }

 public MyPersonFilter (int op, Person2 person){
    int m=(op & MASK_OP);
    if (m!=OP_IGNORE && m!=OP_AFTER && m!=OP_BEFORE)
       throw new IllegalArgumentException("invalid operator");
    if (person == null)
   throw new IllegalArgumentException("person must not be null");
       this.op = op;
       this.person = person;
 }

 public boolean covers(Object obj){
       if (!(obj instanceof Person2)) {
          return false;
       }
       Person2 p2 = (Person2)obj;
       boolean b;
       switch (op & MASK_OP) {
          case OP_BEFORE:
             b=p2.birthdate.compareTo(person.birthdate) < 0;
             break;
          case OP_AFTER:
                  b=p2.birthdate.compareTo(person.birthdate) > 0;
             break;
          case OP_IGNORE:
             b=true;
             break;
          default:
             throw new IllegalStateException("invalid operator");
       }
       if ((op & OP_NOT)>0)
```

```
            b=!b;
        return  b;
    }
}
```

Service Discovery

Offering a Service

By default, services are not immediately visible for service discovery. The visibility of a service is controlled by its *zone* property. Hence, to make a service visible in the network, its zone property must be defined and the zone must be set to a value other than `"rt"`. The zone name `"rt"` stands for runtime and expresses that a service or channel is only visible within the same runtime, i.e. node.

The service of the RMC Chat Example can be made visible for service discovery by adding the following line of code with the `setServiceZone` statement:

```
public void init() {
  try {
    // make this service visible for Service Discovery
    setServiceZone("lan");
```

Service Query by Interface

The following program shows how to find all services implementing `IChat`:

```
import org.mundo.util.DefaultApplication;
import org.mundo.service.ServiceManager;
import org.mundo.service.ServiceInfoFilter;
import org.mundo.service.ResultSet;

public class Discovery1 extends DefaultApplication {
  public Discovery1() {
  }
  @Override
  public void run() {
    try {
      ServiceInfoFilter filter = new ServiceInfoFilter();
      filter.filterInterface("IChat");
      ResultSet rs = ServiceManager.getInstance().query(filter);
      Thread.sleep(1000);
      System.out.println(rs);
    } catch(Exception x) {
      x.printStackTrace();
    }
  }
  public static void main(String args[]) {
    start(new Discovery1());
  }
}
```

Service Discovery in MundoCore is based on content-based publish/subscribe, which was described already in an earlier tutorial. The general ideas behind mapping the service discovery problem to content-based pub/sub are the following:

- Advertising a service corresponds to publishing a notification, containing the service description.
- A service query defines a filter and only matching service descriptions are delivered to the requesting application. Consequently, the concepts to formulate service queries are very similar to defining content-based filters.

The class `ServiceInfoFilter` encapsulates the query. In the program above, the method `filterInterface` is used to define a filter for interface `IChat`. Hence, only services that implement the interface `IChat` will be discovered. Next, the method `query` is called on the `ServiceManager` singleton. It will send the query message to the network, and all matching services respond back. The call returns a `ResultSet` containing a list of all matching services, which is printed to the console.

After the query, the program waits for 1 second before printing the discovery result. Hence, it allows peers in the network 1 second to respond at maximum. Note that it is generally only possible to discover all matching services, if the upper bound for the network latency is predictable.

Continuous Queries

The concept of continuous queries aims to overcome the waiting problem discussed above. It allows an application to receive notifications when new services appear, disappear, or change their properties.

```
import org.mundo.util.DefaultApplication;
import org.mundo.service.ServiceManager;
import org.mundo.service.ServiceInfoFilter;
import org.mundo.service.ResultSet;

public class ContQuery1 extends DefaultApplication {
  public ContQuery1() {
  }
  @Override
  public void run() {
    try {
      ServiceInfoFilter filter = new ServiceInfoFilter();
      filter.filterInterface("IChat");
      ServiceManager.getInstance().contQuery(filter, getSession(), UPDATE_HANDLER);
      pause();
    } catch(Exception x) {
      x.printStackTrace();
    }
  }
  private final ResultSet.ISignal UPDATE_HANDLER = new ResultSet.SignalAdapter() {
    @Override
    public void inserted(ResultSet rs, int i, int n) {
      System.out.println("inserted: " + rs.getList().subList(i, i+n));
    }
    @Override
    public void removing(ResultSet rs, int i, int n) {
      System.out.println("removing: " + rs.getList().subList(i, i+n));
    }
  };
  public static void main(String args[]) {
    start(new ContQuery1());
  }
}
```

The `contQuery` method takes the following parameters:

1. The first argument is the filter. Only when services matching this filter join, leave, or change their properties, then the callback functions will be called.
2. The second argument specifies the client's session.
3. The third argument specifies the event receiver.

`ResultSet.ISignal` notifies the receiver about the following events:

inserted	Raised after one or more elements were inserted into the result set
removing	Raised before one or more elements are removed from the result set
removed	Raised after one or more elements were removed from the result set
propChanging	Raised before the properties of an element are changed
propChanged	Raised after the properties of an element were changed

Custom Service Descriptions

The advertisement of a service is expressed by the `ServiceInfo` data structure. It contains the following fields:

guid	the unique ID of the service
doService	a remote object reference to the service
instanceName	the friendly name of the service
className	the name of the service class
superClasses	the ancestors of the service class
interfaces	the interfaces the service implements
zone	the visibility of the service
nodeId	the ID of the node hosting the service
pluginName	if the service is part of a component, the name of the plug-in component
userData	a description object provided by the user

The `userData` entry allows service developers to add arbitrary descriptions of functional or non-functional properties to services. Description objects are provided by overriding the method `Service.getServiceInfoUserData`. `ServiceInfoFilter` already provides the functionality to define filters on such user-defined attributes as well.

Building Mundo Components

This chapter describes how to create Mundo Components. Components are self-contained JAR archives that contain one or more services, all support classes (metaclasses, externalization, RMC stubs, pub/sub filtering, etc.), required libraries and descriptions of external dependencies.

Components can be loaded by any MundoCore node that runs a `org.mundo.service.ServiceManager`. This service supports dynamic loading and unloading of services contained in components and it offers interfaces for remote administration of services.

Creating a simple Plugin

First, we create the implementation file (Plugin1.java):

```
import org.mundo.rt.Service;

public class Plugin1 extends Service {
  public Plugin1() {
  }
  @Override
  public boolean init() {
    super.init();
    System.out.println("Plugin1 loaded");
    return true;
  }
  @Override
  public void shutdown() {
    System.out.println("Plugin1 unloaded");
    super.shutdown();
  }
}
```

Next, a configuration file for the Plugin must be created (META-INF/plugin.xml):

```
<plug-in xmlns="http://mundo.org/2004/plugin/">
  <service xmlns="http://mundo.org/2004/plugin/service"
           xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
    <classname>Plugin1</classname>
    <new-instance>
      <name>Plugin1 default instance</name>
    </new-instance>
  </service>
</plug-in>
```

The `classname` tag specifies the fully-qualified name of the class that this Plugin provides. The `new-instance` tag creates an instance of the previously specified class with `Plugin1 default instance` as friendly name for the new service instance. A `service` tag may contain an arbitrary number of `new-instance` tags: If no `new-instance` tag is present, services may be solely created by the application program at a later time. On the other hand, it is also possible to immediately create multiple service instances during loading of the Plugin. The Plugin configuration file may contain multiple `service` sections, allowing Plugins to provide more than one service class.

Running the Plugin

First, let's have a look at the configuration file of the Service Host:

```
<NodeConfig xsi:type="map">
...
  <ServiceManager xsi:type="map" activeClass="ServiceManager$OptServiceManager">
    <plugin-directory>plugins</plugin-directory>
  </ServiceManager>
...
</NodeConfig>
```

It is important that a `pluginDirectory` is specified in this configuration file. Otherwise the Service Host will not enable support for Plugins. Run the service host now:

```
java ServiceHost
```

You have two choices how to deploy plugins: Either you pack them in a jar file or you create a subdirectory-structure for the plugin. While the latter is useful for debugging purposes, JAR files are preferred for production code.

If you are using jar deployment for Plugins, first pack it:

```
jar cf plugin1.jar Plugin1.class META-INF
```

Now copy the archive `plugin1.jar` to the `plugins` directory.

If you chose to use the subdirectory-deployment, first create a subdirectory inside your `plugins` directory. Then, copy all resources needed by the Plugin into that subdirectory. Finally, modify the file creation stamp of the subdirectory using the `touch` command.

```
mkdir /servicehost/plugins/plugin1
cp -r Plugin1.class META-INF /servicehost/plugins/plugin1
touch /servicehost/plugins/plugin1
```

No matter what deployment method you chose, you should see the output from `Plugin1.init` on the console:

```
Plugin1 loaded
```

If you delete `plugin1.jar` or the `plugin1` directory, you should see the output from `Plugin1.shutdown` on the console:

```
Plugin1 unloaded
```

Once the ServiceManager? recognizes that a Plugin file has been deleted, it calls `Mundo.unregisterService` for all service instances that depend on classes contained in the deleted Plugin. Note that the service implementations must ensure that all references to their objects are removed. Otherwise, the behaviour of the unload operation is undefined.

Dynamic service configuration

The following Plugin implements the configuration interface `IConfigure`. The configuration data is stored in a `TypedMap`. The program has also a `main` method such that it can either run as a Plugin or in a standalone mode. (Note that a well-programmed service should test the configuration data for valid data before using the data. This was skipped in the example to keep it short and simple.)

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import org.mundo.rt.Mundo;
import org.mundo.rt.Service;
import org.mundo.rt.TypedMap;
import org.mundo.service.ServiceManager;
import org.mundo.service.IConfigure;
import org.mundo.util.DefaultApplication;

public class Plugin2 extends DefaultApplication implements IConfigure {
  public Plugin2() {
    conf = new TypedMap();
  }
  @Override
  public boolean init() {
    super.init();
    System.out.println("Plugin2 loaded");
    return true;
  }
  @Override
  public void shutdown() {
    System.out.println("Plugin2 unloaded");
    super.shutdown();
  }
  @Override
  public void setServiceConfig(Object obj) {
    conf=(TypedMap)obj;
    System.out.println("param="+conf.getString("param", ""));
  }
  @Override
  public Object getServiceConfig() {
    return conf;
  }
  private TypedMap conf;

  public static void main(String args[]) {
    runServices(new Plugin2());
  }
}
```

You can now run *Inspect* and connect to the program. Right-click the line *Plugin2...* in the *Services* pane and select *Configure*. Clicking on the *Get Configuration* button should yield the following output:

Now change the text to:

```
testvalue
```

and click the *Set Configuration* button. The *Plugin2* program will now print to the console:

```
param=testvalue
```

Using custom objects to store configuration data

The following example shows how configuration data can be stored in custom application objects.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import org.mundo.annotation.*;
import org.mundo.rt.Mundo;
import org.mundo.rt.Service;
import org.mundo.rt.TypedMap;
import org.mundo.service.ServiceManager;
import org.mundo.service.IConfigure;
import org.mundo.util.DefaultApplication;

public class Plugin3 extends DefaultApplication implements IConfigure {
  @mcSerialize
  public static class Config {
    public String param1;
    public int param2;

    public String toString() {
      return param1+", "+param2;
    }
  }
  public Plugin3() {
    conf = new Config();
  }
  public boolean init() {
    super.init();
    System.out.println("Plugin3 loaded");
    return true;
  }
  public void shutdown() {
    System.out.println("Plugin3 unloaded");
    super.shutdown();
  }
  public void setServiceConfig(Object obj) {
    try {
      conf = (Config)obj;
      System.out.println(conf);
    } catch(ClassCastException x) {
      x.printStackTrace();
    }
  }
  public Object getServiceConfig() {
    return conf;
  }
  private Config conf;

  public static void main(String args[]) {
    runServices(new Plugin3());
  }
}
```

Implementing and using the IConfigure interface

The `Service` class provides a default implementation for the `IConfigure` interface: `void setServiceConfig(java.lang.Object cfg)` just throws an `Exception` and should be overwritten by the programmer. Have a look to the Javadoc API definition where rules for a convenient behaviour of the implementation are given. Before parameters are used, you should check if needed parameters are present, if values are not `null` and that they are within the expected range.

`void setServiceConfigMap(TypedMap? map)` is `final` and can not be overwritten. `setServiceConfigMap` activates the specified map and then calls the method `setServiceConfig`.

It is anyway always a good idea to implement the `IConfigure` interface. It makes the service customizable and reusable. As examples in this chapter showed the `setServiceConfig()` method is called before the service is registered to set it up properly. But there is a way when the method is called not directly by the programmer. This is when the service is run as plugin. In this case the `ServiceManager` takes the configuration data from the `plugin.xml` document and passes it to the service by calling the `setServiceConfig()` method. The configuration data has to be in the XML entity that has to be a externalized `TypedMap` (or any other Mundo serializable class) as shown above. The examples also shows how to use other types as strings like integer or Mundo serializable objects. The entity `serialparams` is deserialized to an object of class `SerialParameters`.

```
<plug-in xmlns="http://mundo.org/2004/plugin/">
   <service xmlns="http://mundo.org/2004/plugin/service"
            xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
      <classname>ga.mundo.service.rfid.impl.CSAL2002MundoServiceImpl</classname>
      <new-instance xsi:type="map">
         <name>RFID Reader Plugin</name>
         <config xsi:type="map">
            <zone>lan</zone>
            <channel>kaffeekueche.reader</channel>
            <serialparams xsi:type="map"
                  activeClass="ga.service.util.SerialParameters">
               <portName>COM4</portName>
               <baudRate xsi:type="xsd:int">9600</baudRate>
               <databits xsi:type="xsd:int">8</databits>
               <stopbits xsi:type="xsd:int">1</stopbits>
               <parity xsi:type="xsd:int">0</parity>
               <flowControlIn xsi:type="xsd:int">0</flowControlIn>
               <flowControlOut xsi:type="xsd:int">0</flowControlOut>
            </serialparams>
         </config>
      </new-instance>
   </service>
</plug-in>
```

The `ServiceManager` calls at startup `setServiceConfig(Object config)`, where `config` is a `TypedMap`. `zone`, `channel` and `serialparams` are keys in the map. The keys are used to get the values that are of class `String` except `serialparams` returns a object of class `SerialParameters`.

Mobile Agents

Mobile Code and Agent Basics

The source code for this example is located in the directory `samples/agent/mobilecode`. It is structured into three packages:

- **api**: Defines the interfaces used between agent and main application.
- **agent**: Contains the service that will travel around.
- **app**: Contains the main program that will send the agent around.

api

samples/agent/mobilecode/src/api/IMyAgent.java

```
package api;
import org.mundo.annotation.mcRemote;

@mcRemote
public interface IMyAgent {
  void sayHello();
}
```

This defines the interface of the agent. It is marked as a remote interface, because we want to be able to call the method `sayHello` from the master, regardless of the agent's current location.

agent

samples/agent/mobilecode/src/agent/MyAgent.java

```
package agent;
import org.mundo.annotation.mcSerialize;
import org.mundo.service.Node;
import org.mundo.agent.Agent;
import api.IMyAgent;

@mcSerialize
public class MyAgent extends Agent implements IMyAgent {
  public void sayHello() {
    System.out.println("*** Hello from " + Node.thisNode().getName());
  }
}
```

This class defines the implementation for the agent. It will print the name of the current MundoCore node to the console.

The agent must be packaged into a plug-in component. For this, a plugin description file is required:

samples/agent/mobilecode/META-INF/plugin.xml

```
<plug-in xmlns="http://mundo.org/2004/plugin/">
```

```
  <service xmlns="http://mundo.org/2004/plugin/service"
           xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
    <classname>agent.MyAgent</classname>
  </service>
</plug-in>
```

To preprocess, compile, and package the agent into `agent0.jar`, the following ant build rules are used:

samples/agent/mobilecode/build.xml

```
...
<target name="preprocess" depends="init">
  <apply executable="${mcc}" parallel="true">
    <arg value="-O${prep}" />
    <arg value="-x" />
    <fileset dir="${src}">
      <include name="**/*.java" />
    </fileset>
  </apply>
  <copy file="prep/metaclasses.xml" todir="${build}"/>
</target>

<target name="compile" depends="preprocess"
        description="Compile the Java sources">
  <javac destdir="${build}"
         debug="on" debuglevel="lines,vars,source" deprecation="on"
         encoding="utf-8" includeantruntime="false">
    <classpath refid="project.classpath" />
    <src path="${src}" />
    <src path="${prep}" />
    <include name="**/*.java" />
  </javac>
</target>

<target name="jar" depends="compile">
  <mkdir dir="var/master/com" />
  <jar jarfile="var/master/com/agent1.jar">
    <fileset dir=".">
      <include name="META-INF/*"/>
    </fileset>
    <fileset dir="${build}">
      <include name="api/**"/>
      <include name="agent/**"/>
      <include name="metaclasses.xml"/>
    </fileset>
  </jar>
</target>
...
```

app

Now let's have a look at the main application, which will create the agent and send it around. `MyApp` will be run on the MundoCore node `master`. To configure this node name and to enable loading of plug-in components, the master node uses the following `node.conf.xml` configuration file. It defines that plug-ins will be loaded from the subdirectory `com`. This is the directory, where we will initially place the agent, which we have packaged into `agent0.jar` before.

samples/agent/mobilecode/var/master/node.conf.xml

```
<NodeConfig xsi:type="map" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <config-type>overlay</config-type>
  <name>master</name>

  <ServiceManager xsi:type="map" activeClass="ServiceManager$Options">
    <plugin-directory>com</plugin-directory>
  </ServiceManager>
  ...
</NodeConfig>
```

The following implementation file creates an instance of the agent, sends it to `server1` and then back to the `master`. It performs the following steps:

1. It creates a service for the client. This is necessary, because for all communications in MundoCore, there must always be a service context. This applies for the callee as well as the caller.
2. An instance of the agent is created using `Agent.newInstance`. It would not be possible to directly create the object (i.e., using `new MyAgent?()`, because the agent is loaded from a plug-in. `Agent.newInstance` returns a remote reference of type `DoIMobility`, which can be used to move the agent.
3. Next, we cast the reference to `DoIMyAgent`, which allows us to call the methods of our agent interface `IMyAgent`.

Because calls on the agent interface and calls to the `IMobility` interface are done through distributed objects, these calls can be performed regardless of the agent's current location.

samples/agent/mobilecode/src/app/MyApp.java

```
package app;
import org.mundo.rt.Mundo;
import org.mundo.rt.Service;
import org.mundo.agent.Agent;
import org.mundo.agent.DoIMobility;
import api.DoIMyAgent;

public class MyApp {
  public static void main(String args[]) {
    Mundo.init();
    try {
      Service client = new Service();
      Mundo.registerService(client);

      DoIMobility mobility = Agent.newInstance(client.getSession(), "agent.MyAgent");
      DoIMyAgent agent = new DoIMyAgent(mobility);
      agent.sayHello();

      mobility.moveTo("server1");
      agent.sayHello();

      mobility.moveTo("master");
      agent.sayHello();
    }
    catch(Exception x) {
```

```
        x.printStackTrace();
      }
      Mundo.shutdown();
    }
  }
```

Running the example

The example uses two MundoCore nodes, named `master` and `server1`. Their working directories reside in `var/master` and `var/server1`, respectively. The buildfile will generate the runscripts `run-server1` and `run-master` to start these nodes. To run the example, start `run-server1` and then `run-master`.

- **server1** is an "empty" MundoCore node. It simply runs `org.mundo.util.DefaultApplication` from the MundoCore library. The `node.conf.xml` configuration file in the `var/server1` subdirectory defines the node name and the plug-in directory. This is necessary to receive mobile code from peers.
- **master** runs the application **MyApp** described above.

Next: Creating Autonomous Agents

Creating Autonomous Agents

Previous: Mobile Code and Agent Basics

In this step we will extend the previous example to an autonomous agent, i.e., an agent, that is able to move itself.

api

Again, the `api` package contains the interfaces used between the agent and the application. Here, the method `run` initializes the agent with a `name` parameter and starts it.

samples/agent/agent1/src/api/IMyAgent.java

```
package api;
import org.mundo.annotation.mcRemote;

@mcRemote
public interface IMyAgent {
  void run(String name);
}
```

agent

Now, most of the functionality of this example is implemented in the agent itself. Agents can carry arbitrary data with them. Here, this is demonstrated with the field `name`. To migrate this data when the agent moves, we declare the class to be serializable with `@mcSerialize`. Note that serializable fields may not have `private` visibility. In the code below, the default visibility is used, i.e., the field `name` will be package private.

The agent performs the following steps:

1. The application will create the agent and then call the `run` method. It initializes the field `name` and then moves to `server1`. Because at a `moveTo` statement, the agent has to suspend its operation on the current node, migrate to another node, and then resume there, the execution of a method cannot continue after a `moveTo` operation. (Aborting the execution in the middle of a method would be doable, but resuming the execution in the middle of a method would be virtually impossible in Java.) Hence, the code must be split up into multiple methods. The second parameter of `moveTo` specifies the name of the method that will be invoked at the new location immediately after the service resumes there. Here, the agent is instructed to move to `server1` and then continue with `atServer1`.
2. Arrived at `server1`, the agent executes method `atServer1`. It writes a message to the console and then moves on to `server2`.
3. Similarly, arrived at `server2`, the agent executes method `atServer2` and then moves back to the `master`.
4. Finally, arrived at `master`, the method `atMaster` is executed.

samples/agent/agent1/src/agent/MyAgent.java

```
package agent;
import org.mundo.agent.Agent;
import org.mundo.annotation.mcSerialize;
import org.mundo.service.Node;
import api.IMyAgent;

@mcSerialize
public class MyAgent extends Agent implements IMyAgent {
  String name;
```

```
  public void run(String name) {
    this.name = name;
    System.out.println("*** "+name+" starting at "+Node.thisNode().getName());
    moveTo("server1", "atServer1");
  }
  public void atServer1() {
    System.out.println("*** "+name+" now at "+Node.thisNode().getName());
    moveTo("server2", "atServer2");
  }
  public void atServer2() {
    System.out.println("*** "+name+" now at "+Node.thisNode().getName());
    moveTo("master", "atMaster");
  }
  public void atMaster() {
    System.out.println("*** "+name+" back at "+Node.thisNode().getName());
  }
}
```

app

samples/agent/agent1/src/app/MyApp.java

```
package app;
import org.mundo.rt.Mundo;
import org.mundo.rt.Service;
import org.mundo.agent.Agent;
import org.mundo.agent.DoIMobility;
import api.DoIMyAgent;

public class MyApp {
  public static void main(String args[]) {
    Mundo.init();
    try {
      Service client = new Service();
      Mundo.registerService(client);

      DoIMobility mobility = Agent.newInstance(client.getSession(), "agent.MyAgent");
      DoIMyAgent agent = new DoIMyAgent(mobility);
      agent.run("Agent1");
    }
    catch(Exception x) {
      x.printStackTrace();
    }
    Mundo.shutdown();
  }
}
```

Running the example

The example uses three MundoCore nodes, named `master`, `server1`, and `server2`. Their working directories reside in `var/master`, `var/server1`, and `var/server2`, respectively. The buildfile will generate the runscripts `run-server1`, `run-server2`, and `run-master` to start these nodes. To run the example, start `run-server1`, `run-server2`, and then `run-master`.

- **server1** and **server2** are "empty" MundoCore nodes. They simply run `org.mundo.util.DefaultApplication` from the MundoCore library. The `node.conf.xml` configuration file in these directories define the node name and the plug-in directory. This is necessary to receive mobile code from peers.
- **master** runs the application **MyApp** described above.

Next: Communicating With Local Services

Communicating With Local Services

Previous: Creating Autonomous Agents

Any nontrivial agent will have to communicate with other services or agents to do its task. This part of the tutorial will show two things:

- How to obtain references to local services running on the same node and how to communicate with them.
- How to obtain information about other nodes currently present in the MundoCore overlay network.

api

Like in the examples before, `IMyAgent` specifies the interface of the agent, which will be used by the master.

samples/agent/agent2/src/api/IMyAgent

```
package api;
import org.mundo.annotation.mcRemote;

@mcRemote
public interface IMyAgent {
  void run();
}
```

In this example, the agent will visit a server and communicate with it. The interface `IMyServer` defines the interface of the server that will be used by the agent.

samples/agent/agent2/src/api/IMyServer

```
package api;
import org.mundo.annotation.mcRemote;

@mcRemote
public interface IMyServer {
  void step1();
  void step2();
  void step3();
  void step4();
  void step5();
}
```

agent

When the agent is started through the `run` method, it first determines the server to which it will move. In the previous examples, the name of the destination node was hardcoded. Here, the agent uses `Node.getNeighbors()` to obtain a list of the neighbor nodes. The agent then chooses the first item of the list and moves to this node.

Arrived at the server, the agent uses `Mundo.Mundo.getServiceByType` to obtain a reference to a service that implements the `IMyServer` interface. After that, it calls five methods on the server and finally migrates back to the master.

This example illustrates a frequently cited feature of mobile agents: Instead of performing multiple (slow) remote calls to a server, we move the agent to the server and perform the calls there locally.

samples/agent/agent2/src/agent/MyAgent

```
package agent;
import org.mundo.agent.Agent;
import org.mundo.annotation.mcSerialize;
import org.mundo.service.Node;
import org.mundo.rt.Mundo;
import api.IMyAgent;
import api.IMyServer;

@mcSerialize
public class MyAgent extends Agent implements IMyAgent {
  public void run() {
    System.out.println("*** starting at "+Node.thisNode().getName());

    Node[] nodes = Node.getNeighbors();
    if (nodes.length < 1)
      throw new IllegalStateException("no peers present");
    moveTo(nodes[0].getName(), "atServer1");
  }
  public void atServer1() {
    System.out.println("*** now at "+Node.thisNode().getName());

    IMyServer srv = (IMyServer)Mundo.getServiceByType(IMyServer.class);
    if (srv == null)
      throw new IllegalStateException("server service not found!");
    srv.step1();
    srv.step2();
    srv.step3();
    srv.step4();
    srv.step5();

    System.out.println("*** leaving "+Node.thisNode().getName());
    moveTo("master", "atMaster");
  }
  public void atMaster() {
    System.out.println("*** back at "+Node.thisNode().getName());
  }
}
```

server

The server implementation just prints to console, when a method is called.

samples/agent/agent2/src/server/MyServer

```
package server;
import org.mundo.util.DefaultApplication;
import api.IMyServer;

public class MyServer extends DefaultApplication implements IMyServer {
  public MyServer() {
  }
  public void step1() {
```

```
    System.out.println("step1");
  }
  public void step2() {
    System.out.println("step2");
  }
  public void step3() {
    System.out.println("step3");
  }
  public void step4() {
    System.out.println("step4");
  }
  public void step5() {
    System.out.println("step5");
  }
  public static void main(String args[]) {
    start(new MyServer());
  }
}
```

app

The following implementation of `MyApp` creates an instance of the agent and runs it.

samples/agent/agent2/src/app/MyApp

```
package app;
import org.mundo.rt.Mundo;
import org.mundo.rt.Service;
import org.mundo.agent.Agent;
import org.mundo.agent.DoIMobility;
import api.DoIMyAgent;

public class MyApp {
  public static void main(String args[]) {
    Mundo.init();
    try {
      Service client = new Service();
      Mundo.registerService(client);
      DoIMobility mobility = Agent.newInstance(client.getSession(), "agent.MyAgent");
      DoIMyAgent agent = new DoIMyAgent(mobility);
      agent.run();
    }
    catch(Exception x) {
      x.printStackTrace();
    }
    Mundo.shutdown();
  }
}
```

Running the example

The example uses two MundoCore nodes, named `master` and `server`. Their working directories reside in `var/master` and `var/server`, respectively. The buildfile will generate the runscripts `run-server` and `run-master` to start these nodes. To run the example, start `run-server` and then `run-master`.

- **server** runs the server service **MyServer**.
- **master** runs the application **MyApp**.

MundoCore on Android

Prerequisites

- The Android SDK. It can be obtained from: http://developer.android.com/

AndroidChat?

In the following, an implementation of the simple chat example for Android is presented.

The creation of Eclipse projects for Android is similar to the creation of projects for the Standard Edition, as described here: Creating a New Project in Eclipse.

- Select **File** > **New Project**
- Select **Android** > **Android Project**, then click **Next**
- This example uses the following settings:

Project name:	AndroidChat?
Application name:	AndroidChat?
Package name:	org.mundo.chat
Create Activity:	ChatTest?
Min SDK Version:	8

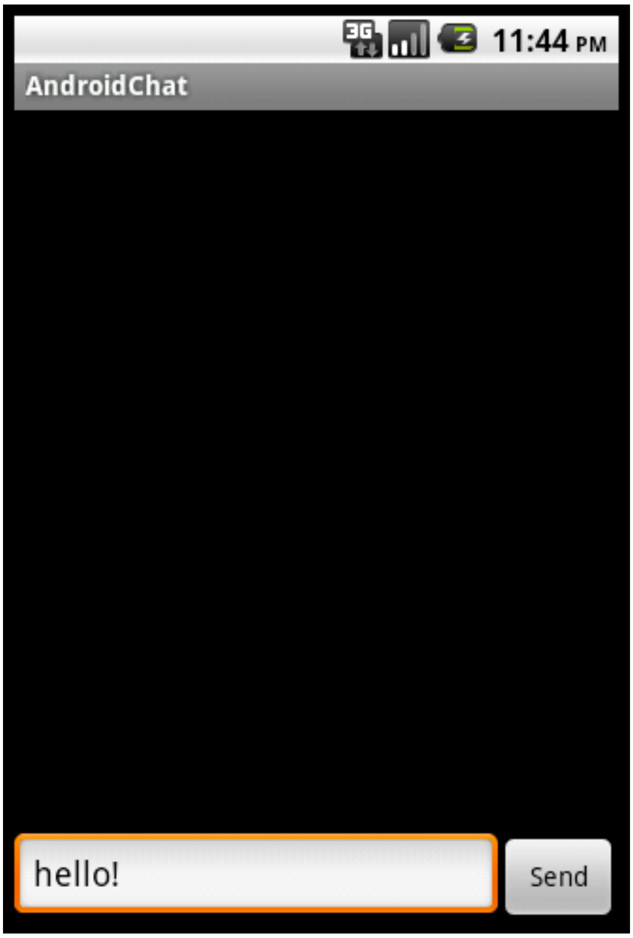
Project Structure

The following shows the Eclipse project structure of the AndroidChat? example program:

- src
  - org.mundo.chat
    - **ChatTest.java**: the main Java code goes here
    - R.java (automatically generated from the resources)
- Android Library
  - android.jar
- Referenced Libraries

    - **mundocore.jar**: link the standard 1.5 library here
- assets
    - **node.conf.xml**: put the MundoCore configuration file here
- res
    - layout
        - **main.xml**: description of the user interface
    - ...
- **AndroidManifest.xml**

User Interface



The user interface is described in the file **main.xml**.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">
    <ScrollView android:layout_width="fill_parent"
                android:layout_height="fill_parent"
                android:layout_weight="1"
                android:id="@+id/scrollView">
        <TextView android:layout_width="fill_parent"
                  android:text="@string/hello"
                  android:layout_height="fill_parent"
                  android:layout_weight="1"
                  android:id="@+id/textPane" />
    </ScrollView>
    <LinearLayout android:id="@+id/LinearLayout01"
                  android:layout_height="wrap_content"
                  android:orientation="horizontal"
                  android:layout_width="fill_parent">
```

```
        <EditText android:layout_height="wrap_content"
                  android:clickable="false"
                  android:layout_width="fill_parent"
                  android:layout_weight="1"
                  android:id="@+id/inputLine"
                  android:singleLine="true" />
        <Button android:layout_height="wrap_content"
                android:text="Send"
                android:layout_width="fill_parent"
                android:layout_weight="4"
                android:id="@+id/sendButton" />
    </LinearLayout>
</LinearLayout>
```

Implementation

ChatTest.java contains the implementation.

```
package org.mundo.chat;

import java.io.InputStreamReader;

import android.app.Activity;
import android.os.Bundle;
import android.view.View;
import android.widget.Button;
import android.widget.EditText;
import android.widget.ScrollView;
import android.widget.TextView;

import org.mundo.rt.LogEntry;
import org.mundo.rt.Mundo;
import org.mundo.rt.Logger;
import org.mundo.rt.Service;
import org.mundo.rt.Publisher;
import org.mundo.rt.IReceiver;
import org.mundo.rt.Message;
import org.mundo.rt.MessageContext;
import org.mundo.rt.TypedMap;

public class ChatTest extends Activity {
  /** Called when the activity is first created. */
  @Override
  public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);

    scrollView = (ScrollView)findViewById(R.id.scrollView);
    textPane = (TextView)findViewById(R.id.textPane);
    inputLine = (EditText)findViewById(R.id.inputLine);
    sendButton = (Button)findViewById(R.id.sendButton);
    sendButton.setOnClickListener(new View.OnClickListener() {
      public void onClick(View v) {
        String text = inputLine.getText().toString();
        TypedMap map = new TypedMap();
        map.putString("ln", text);
```

```java
        publisher.send(new Message(map));
        textPane.append("("+text+")\n");
        scrollView.fullScroll(ScrollView.FOCUS_DOWN);
        inputLine.setText("");
      }
    });

    Logger.getLogger("global").addHandler(LOG_HANDLER);
    try {
      Mundo.setConfigXML(new InputStreamReader(getClass().getClassLoader().
          getResourceAsStream("assets/node.conf.xml"), "UTF-8"));
    } catch(Exception x) {
      Logger.getLogger("global").warning("could not read node.conf.xml!");
    }
    Mundo.init();
    Service svc = new Service();
    Mundo.registerService(svc);
    publisher = svc.getSession().publish("lan", "chattest");
    svc.getSession().subscribe("lan", "chattest", CHAT_RECEIVER);
  }

  @Override
  protected void onDestroy() {
    Mundo.shutdown();
    super.onDestroy();
  }

  private final IReceiver CHAT_RECEIVER = new IReceiver() {
    public void received(Message msg, MessageContext ctx) {
      runOnUiThread(new PrintAction(msg.getMap().getString("ln")));
    }
  };

  class PrintAction implements Runnable {
    PrintAction(String l) {
      ln = l;
    }
    public void run() {
      textPane.append(ln+"\n");
    }
    private String ln;
  }

  private final Logger.IHandler LOG_HANDLER = new Logger.IHandler() {
    public void publish(LogEntry e) {
      if (e.getLevel() <= Logger.WARNING)
        runOnUiThread(new PrintAction(e.toString()));
    }
  };

  private Button sendButton;
  private TextView textPane;
  private EditText inputLine;
  private ScrollView scrollView;
  private Publisher publisher;
```

```
}
```

AndroidManifest?.xml

By default, Android applications may not enumerate the available network interfaces or open server sockets. It is necessary to request the required permissions first in the manifest. When using Eclipse, perform the following steps to add permissions:

- Open `AndroidManifest.xml` (with the Android Manifest Editor)
- Select the **Permissions** tab
- Click **Add**
- Select **Uses Permission**
- Change the name of the new permission to `android.permission.INTERNET`
- Save the manifest

Android-specifics in the Java Code

Loading the Configuration File

An Android program uses the following code to read and set the configuration file `node.conf.xml`. Note that `Mundo.setConfigXML` must be called before `Mundo.init`.

```
Mundo.setConfigXML(new InputStreamReader(getClass().getClassLoader()
                   .getResourceAsStream("assets/node.conf.xml"), "UTF-8"));
Mundo.init();
```

Threads

Calls to UI components must not be made from a non-UI thread. For that reason, a message receiver cannot directly modify views, it has to delay the changes and execute them within the UI thread. Using `runOnUiThread` is one possible approach:

```
private final IReceiver CHAT_RECEIVER = new IReceiver() {
    public void received(Message msg, MessageContext ctx) {
        runOnUiThread(new PrintAction(msg.getMap().getString("ln")));
    }
};

class PrintAction implements Runnable {
    PrintAction(String l) {
        ln = l;
    }
    public void run() {
        textPane.append(ln+"\n");
    }
    private String ln;
}
```

Logging

The following is optional. Log output can also be redirected to, e.g., a text view using a log handler:

```
private final Logger.IHandler LOG_HANDLER = new Logger.IHandler() {
```

```
    public void publish(LogEntry e) {
        runOnUiThread(new PrintAction(e.toString()));
    }
}
```

The handler must then be registered with the MundoCore framework:

```
Logger.getLogger("global").addHandler(LOG_HANDLER);
```

Known Issues

Discovery

The emulator comes with a somewhat problematic NAT implementation that does not give the host a real IP address in the NATed address space. For that reason, automatic discovery will not work. Use a host entry in the configuration file of the Android node to explicitly connect to the host PC, which is reachable under the special address `10.0.2.2`:

```
<primary-port xsi:type="xsd:int">4242</primary-port>
<hosts xsi:type="array">
  <host xsi:type="map" activeClass="IPTransportService$OptHost">
    <name>10.0.2.2</name>
    <retry-interval xsi:type="int">3</retry-interval>
  </host>
</hosts>
```

(also see: http://d.android.com/guide/developing/tools/emulator.html#emulatornetworking)

On the real device, broadcast discovery can be used as usual, e.g.:

```
<primary-port xsi:type="xsd:int">4242</primary-port>
<broadcast xsi:type="map" activeClass="IPTransportService$OptBroadcast">
  <send xsi:type="boolean">true</send>
</broadcast>
```

Pause/Resume

MundoCore will not be able to communicate correctly after a pause/resume, because the application on the Android phone is completely suspended for a while. Remote peers will run into timeouts and think that the Android node has gone. To fix this problem, force close the application in `onStop`:

```
@Override
public void onStop() {
  Mundo.shutdown();
  System.exit(0);
}
```

Alternatively, put the communication inside an Android Service that continues running in the background (http://developerlife.com/tutorials/?p=356).

Node Configuration

node.conf.xml

The configuration file is named `node.conf.xml` and usually resides in the same directory as the program. The structure of this file is as follows:

```
<NodeConfig xsi:type="map" xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance">
  <name ... ?>
  <ServiceManager ...>
  <Logger ... ?>
</NodeConfig>
```

Type default for `xsi:type` is `string`. For that reason, the type specification can be omitted for string options.

- `...>`: appears once
- `... ?>`: is optional
- `... *>`: appears zero or many times

General Settings

Node name

```
<name xsi:type="string">NAME</name>
```

The NAME attribute defines the friendly name for the node. The node name can be displayed in the *Inspect* application and is very helpful during monitoring and debugging.

ServiceManager?

```
<ServiceManager xsi:type="map" activeClass="ServiceManager$Options">
  <instances ...>
  <plugin-directory ... ?>
</ServiceManager>
```

instances

```
<instances xsi:type="array">
  <new-instance ... *>
</instances>
```

new-instance

```
<new-instance xsi:type="map">
  <name xsi:type="xsd:string">...?</name>
  <classname xsi:type="xsd:string">...</classname>
  <config ...?>
</new-instance>
```

`name`: The friendly name of the new service instance.

`classname`: The class name of the service class to create.

`config`: Configuration options for the new service instance. This is service-specific.

plugin-directory

```
<plugin-directory xsi:type="xsd:string">...</plugin-directory>
```

`plugin-directory`: Specifies the directory to search for components?.

IPTransportService?

```
<new-instance xsi:type="map">
  <name>IPTransport</name>
  <classname>org.mundo.net.ip.IPTransportService</classname>
  <config xsi:type="map" activeClass="IPTransportService$Options">
    <primary-port xsi:type="xsd:int">...?</primary-port>
    <protocol xsi:type="xsd:string">...?</protocol>
    <tcp-server xsi:type="xsd:boolean">...?</tcp-server>
    <udp-server xsi:type="xsd:boolean">...?</udp-server>
    <localhost-udp xsi:type="xsd:boolean">...?</localhost-udp>
    <route-timeout xsi:type="xsd:int">...?</route-timeout>
    <connection-timeout xsi:type="xsd:int">...?</connection-timeout>
    <keep-open xsi:type="xsd:boolean">...?</keep-open>
    <per-interface-server xsi:type="xsd:boolean">...?</per-interface-server>
    <discovery ...>
```

```
    <broadcast ...>
    <multicast ...>
    <hosts ...>
  </config>
</new-instance>
```

primaryPort (default: 4242): The primary port is a well known TCP port that is shared by all nodes participating in the same overlay network. When a node is started, it tries to allocate the primary port. This allocation will fail if another MundoCore process is already running on the same machine. In that case, the process allocates any other free port for its server and connects to the primary port on the local machine. If the connection to the primary port breaks, this means that the process that held the primary port has shut down. When this happens, the current process tries to allocate the primary port. Thus, if at least one MundoCore process is running on a machine, some process will listen on the primary port.

discovery: Specifies the options for node discovery.

tcp-server: Specifies if the node supports incoming TCP connections. In almost any case a node also provides TCP server functionality. However, for example, a node running on a cellphone might set this option to false and therefore only act as a TCP client.

udp-server: Defines whether this node allows incoming UDP connections.

route-timeout (default: 75): Defines the timeout in seconds for routes. The route timeout counter is reset each time a broadcast discovery packet or a neighbors message is received. Usually, route-timeout should be set to something like 2.5 * broadcast-interval to take possible packet losses into account. If UDP packet loss in the network is high, then the factor between route-timeout and broadcast-interval should be increased. The value of routeTimeout also determines how long it takes, in the worst case, for the node to realize that a peer has become unavailable.

connection-timeout (default: 10): Defines the idle timeout in seconds for connections. MundoCore closes unused TCP connections after a relatively short time and re-opens them on demand.

keep-open: Defines if idle TCP connections should be closed automatically. If set to true, this node will ignore the connectionTimeout option and will not close any connections automatically. In addition, the node requests peers at connection setup to keep connections open from the other side as well. Otherwise, peers could close connections based on their local idle timeout counters. The keep-open option should not be carelessly set to true, because many connections will be opened during discovery that are useless in the long term. These connections will be closed automatically as soon as they become idle.

per-interface-server: Starts a TCP server for each network interface separately. If this option is set false, then this service will bind server sockets without specifying IP addresses. If this option is set true, then this service will perform separate bind operations for each IPv4 network interface. If a socket bind operation is performed without specifying an address, then the bind affects all IPv4 and IPv6 network interfaces. Now, it might occur that an IPv4 port is already allocated, but the corresponding IPv6 port is available. The operating system will return success on the bind operation, but the resulting socket is useless for IPv4.

discovery

Specifies options for automatic peer discovery.

```
<discovery xsi:type="map" activeClass="IPTransportService$OptDiscovery">
  <connect-primary xsi:type="xsd:boolean">...?</connect-primary>
  <send-neighbors xsi:type="xsd:boolean">...?</send-neighbors>
  <attributes xsi:type="xsd:string">...?</attributes>
  <required-attribute xsi:type="xsd:string">...?</required-attribute>
</discovery>
```

`connect-primary` (default: true): Defines if a node should try to connect to the local primary port. This mechanism allows to discover peer nodes running on the same computer even when no real network interface is present. Broadcast discovery would not work reliably in this case, because, e.g., the Windows loopback network interface randomly loses broadcast packets. The default value of \code{connectToPrimary} is \code{true}. There is usually no reason to disable this function.

`attributes` (default: ""): Defines a list of attributes for this node. This option can be used to restrict the scope of peer discovery. For example, *super peers* can identify themselves by setting certain attributes.

`required-attribute` (default: ""): Requires peers to have the specified attribute. This option can be used to restrict the scope of peer discovery. This node will only connect to nodes that have the specified attribute defined. For example, a node can define that it will only connect to super peers.

broadcast

```
<broadcast xsi:type="map" activeClass="IPTransportService$OptBroadcast">
  <send xsi:type="xsd:boolean">...?</send>
  <answer xsi:type="xsd:boolean">...?</answer>
  <interval xsi:type="xsd:int">...?</interval>
  <nets ...?>
</broadcast>
```

`answer` (default: false): Defines if the node responds to broadcast discovery packets. If a node receives a broadcast discovery packet, it opens a TCP connection to the originating node.

`interval` (default: 30): Defines the interval in seconds between sending broadcast packets.

broadcast/nets

The broadcast option defines the list of network interfaces on which broadcast discovery packets should be sent. If a `net` entry matches a network interface, then the specified `net` configuration settings will be used. For all other network interfaces, the `default` settings will be used. If `default` is omitted, then broadcasts will only be sent on the interfaces explicitly specified by `net` entries.

```
<broadcasts xsi:type="xsd:array">
  <net ...*>
  <default ...?>
</broadcasts>
```

The `net` element has two subelements, `broadcast` and `netmask`. Both must be specified.

```
<net xsi:type="map" activeClass="IPTransportService$OptNet">
  <broadcast>...</broadcast>
  <netmask>...</netmask>
</net>
```

In the *default* entry, the broadcast address is omitted.

```
<default xsi:type="map" activeClass="IPTransportService$OptNet">
  <netmask>...</netmask>
</default>
```

`broadcast`: The broadcast address to use. MundoCore will automatically search for a suitable network interface. If none can be found, the configuration entry is ignored. `broadcast` contains four decimal octets separated by dots. The broadcast address is defined as follows:

```
broadcast = netaddr | (0xffffffff & ~netmask)
```

`netmask`: Specifies an IPv4 netmask, which defines the class of the network. The string contains four decimal octets separated by dots.

These options can be used to override netmasks and broadcast addresses provided by the operating system. It is necessary to do so under the following two circumstances:

- The Java API permits to enumerate the network interfaces present, but it does not provide netmasks. Thus, MundoCore assumes that each interface lives in a class C network. If this is not the case, the correct netmask and broadcast address must be specified in the configuration file.
- The C++ version is able to read out the netmasks from the operating system. However, the information obtained from the OS can be wrong (e.g., when using VPN connections on Windows CE) and must be overridden in the configuration file.

MundoCore automatically matches the available network interfaces with the configuration entries made in this section. An interface with `ipaddr` matches a configuration entry, if `(ipaddr & netmask) =` netaddr= with `netaddr = (broadcast & netmask).`

If a match is found, then the `netmask` and `broadcast` attributes defined in the configuration override the settings of the network interface.

Broadcast example 1: 10.x.x.x

Consider a PDA with a WLAN interface that is configured as follows:

IP address:	10.0.0.100
broadcast:	10.255.255.255
netmask:	255.0.0.0

Since MundoCore Java is unable to get the netmask from the operating system, it assumes a class C network by default. However, some operating systems do not propagate broadcast packets at all, if a wrong broadcast address is specified.

To override broadcast and netmask with the correct values, make the following entry in the configuration file:

```
<net xsi:type="map" activeClass="IPTransportService$OptNet">
  <broadcast>10.255.255.255</broadcast>
  <netmask>255.0.0.0</netmask>
</net>
```

Broadcast example 2: VPN

Consider a PDA with a WLAN interface using a PPTP VPN. After connection establishment, the VPN client provides a virtual network interface with the following settings:

IP address:	130.83.163.240 (set by DHCP)
broadcast:	130.83.255.255
netmask:	255.255.0.0

The MundoCore C++ version uses the information provided by the operating system. However, the target network is in reality a half class C network and there seems to be no way to configure the correct netmask at the VPN client or server.

To override broadcast and netmask with the correct values, make the following entry in the configuration file:

```
<net xsi:type="map" activeClass="IPTransportService$OptNet">
  <broadcast>130.83.163.255</broadcast>
  <netmask>255.255.255.128</netmask>
</net>
```

multicast

```
<multicast xsi:type="map" activeClass="IPTransportService$OptMulticast">
  <group xsi:type="xsd:string">...?</group>
  <port xsi:type="xsd:int">...?</port>
  <ttl xsi:type="xsd:int">...?</ttl>
  <send xsi:type="xsd:boolean">...?</send>
  <answer xsi:type="xsd:boolean">...?</answer>
  <interval xsi:type="xsd:int">...?</interval>
</multicast>
```

hosts

```
<hosts xsi:type="xsd:array">
  <host ...*>
</hosts>

<host xsi:type="map" activeClass="IPTransportService$OptHost">
  <name xsi:type="xsd:string">...</name>
  <port xsi:type="xsd:string">...?</port>
  <retry-interval xsi:type="xsd:int">...?</retry-interval>
  <disconnected-retry-interval xsi:type="xsd:int">...?</disconnected-retry-interval>
</host>
```

Requires MoundoCore? to explicitly connect to a node. MundoCore will try to open a TCP connection to the specified host and port.

`name`: Defines the host name or IP address of the peer.

`port`: Defines the port number on the peer. If omitted, the port number defaults to the primary port number of this node.

`retry-interval`: Defines the retry interval in seconds. If MundoCore is unable to open the specified connection, then it will retry every `retry-interval` seconds. If this option is omitted, MundoCore will only make a single connection attempt at program startup and no further retries.

`disconnected-retry-interval`: Defines the retry interval in seconds. This value overrides `retry-interval` in case that this node is "isolated", i.e., it does not have any connection open to any peer.

Example configurations

Discovery example 1: Localhost

If no discovery configuration is present, then MundoCore uses a secure default configuration. The node will only discover other peers running on the same host and it will not respond to discovery requests received from the network. This internal default configuration is equivalent to the following configuration:

```
<IPTransportService xsi:type="map">
  <primary-port xsi:type="xsd:int">4242</primary-port>
  <discovery xsi:type="map" activeClass="IPTransportService$OptDiscovery">
    <connect-primary xsi:type="xsd:boolean">true</connect-primary>
  </discovery>
  <broadcast xsi:type="map" activeClass="IPTransportService$OptBroadcast">
    <send xsi:type="boolean">false</send>
    <answer xsi:type="boolean">false</answer>
  </broadcast>
</IPTransportService>
```

Discovery example 2: LAN

To enable discovery within the local subnet, the options `broadcasts` and `answer-broadcasts` must be set:

```
<IPTransportService xsi:type="map">
  <primary-port xsi:type="xsd:int">4242</primary-port>
  <broadcast xsi:type="map" activeClass="IPTransportService$OptBroadcast">
    <send xsi:type="boolean">true</send>
    <nets xsi:type="array">
      <default xsi:type="map" activeClass="IPTransportService$OptNet">
        <netmask>255.255.255.0</netmask>
      </default>
    </nets>
  </broadcast>
</IPTransportService>
```

This configuration should suit most applications. A couple of special configurations are described in the following sections.

Discovery example 3: ActiveSync?

It is possible to use the USB connection between PDA and host for communication. To set up this type of connection, install the following configuration file on the PDA:

```
<IPTransportService xsi:type="map">
  <primary-port xsi:type="xsd:int">4242</primary-port>
  <hosts xsi:type="array">
    <host xsi:type="map" activeClass="IPTransportService$OptHost">
      <name>192.168.55.100</name>
      <retry-interval xsi:type="int">3</retry-interval>
    </host>
  </hosts>
  <keep-open xsi:type="boolean">true</keep-open>
</IPTransportService>
```

The `keep-open` option is set, because TCP connections can only be opened from the PDA to the host and not vice versa. The host would not be able to reopen connections.

Discovery example 4: Firewalled Broadcasts

If UDP broadcast packets are blocked by the network, then connections can be configured explicitly. In the following, we assume that a server process is running on host `mundo` and one or multiple clients want to connect to it.

Configuration file on server:

```
<primary-port xsi:type="xsd:int">4242</primary-port>
```

Configuration file on client:

```
<primary-port xsi:type="xsd:int">4242</primary-port>
<hosts xsi:type="array">
  <host xsi:type="map" activeClass="IPTransportService$OptHost">
    <name>mundo.tk.informatik.tu-darmstadt.de</name>
    <retry-interval xsi:type="int">3</retry-interval>
  </host>
</hosts>
```

PluginManager?

Specifies options for `PluginManager`.

```
<PluginManager xsi:type="map">
  <enable-subdirectories xsi:type="xsd:boolean">...?</enable-subdirectories>
  <exclude ...?>
</PluginManager>
```

`enable-subdirectories` (default: false): You can set up the plugin manager to search not only JAR files, but also subdirectories that contain classfiles and other resources accoding to Java's standard file lookup. Because this may lead to security holes, you must explicitly enable this feature using the `enable-subdirectories` attribute.

exclude

```
<exclude xsi:type="array">
  <path xsi:type="xsd:string">...*</path>
</exclude>
```

If necessary, you can explicitly exclude one or more JAR files or subdirectories from the plugin mechanism. This is especially useful if you are using an IDE like Eclipse that duplicates the `META-INF/plugin.xml` file between a source and a binary directory.

Logging Configuration

Specifies options for logging.

```
<Logger xsi:type="map">
  <filename xsi:type="xsd:string">...?</filename>
  <console xsi:type="xsd:boolean">...?</console>
  <default-log-level xsi:type="xsd:string">...?</default-log-level>
  <log-levels ...?>
</Logger>
```

`filename`: Specifies the name of the log file.

`console`: If true, log output will be printed to the console.

`default-log-level`: Defines the default log level. Can be overridden by settings in `log-levels`.

`log-levels`: Allows to override the log level for certain loggers.

log-levels

```
<log-levels xsi:type="array">
  <a ...*>
</log-levels>
```

```
<a xsi:type="map">
  <category xsi:type="xsd:string">...</category>
  <log-level xsi:type="xsd:string">...</log-level>
</a>
```

`category`: The name of the `Logger` object created in the service implementation.

`log-level`: The log level for the specified logger.

The supported log levels are shown in the following table. You can use any notation to specify a log level (level, name, or short name).

level	name	short name
0	OFF	
1	SEVERE	S
2	WARNING	W
3	INFO	I
4	CONFIG	C
5	FINE	F
6	FINER	
7	FINEST	

Logging example configuration

```
<Logger xsi:type="map">
  <filename>log.txt</filename>
  <console xsi:type="xsd:boolean">true</console>
  <default-log-level>WARNING</default-log-level>
  <log-levels xsi:type="array">
    <a xsi:type="map"><category>meta</category><log-level>SEVERE</log-level></a>
    <a xsi:type="map"><category>oa</category><log-level>FINEST</log-level></a>
  </log-levels>
</Logger>
```

Writing custom protocol handlers

The programs described in the following sections can also be found in the `samples/handlers` subdirectory of the distribution or in the CVS.

Step 1: Handler skeleton and using logging

The first example shows the skeleton of a protocol handler and how to use logging.

```
import org.mundo.rt.IMessageHandler;
import org.mundo.rt.Message;
import org.mundo.rt.Service;
import org.mundo.rt.Logger;
import org.mundo.net.AbstractHandler;

/**
 * This is one of the simplest possible handlers.
 *
 * @author erwin
 */
public class SampleHandler1 extends AbstractHandler
{
  public SampleHandler1()
  {
  }
  /**
   * Called on initialization of the service.
   */
  @Override
  public void init() // Service
  {
    super.init();
    log.fine("init");
  }
  /**
   * Called on shutdown of the service.
   */
  @Override
  public void shutdown() // Service
  {
    log.fine("shutdown");
    super.shutdown();
  }
  /**
   * Called when a packet travels down the stack.
   */
  public boolean down(Message msg)
  {
    log.fine("down: " + msg.getBlob("all", "bin").size());
    return emit_down(msg);
  }
  /**
   * Called when a packet travels up the stack.
   * This method will not be called, because we did not set the MIMEType yet.
   */
  public boolean up(Message msg)
```

```
  {
    log.fine("up: " + msg.getBlob("all", "bin").size());
    return emit_up(msg);
  }

  private Logger log = Logger.getLogger("sample");
}
```

This new protocol handler can now be used in custom protocol stacks or in the global protocol stack. The global protocol stack is configured in the `node.conf.xml` configuration file. To add `SampleHandler1` to the protocol stack, insert it into the protocol stack configuration at the appropriate place:

```
<new-instance xsi:type="map">
  <name>ProtocolCoordinator</name>
  <classname>org.mundo.net.ProtocolCoordinator</classname>
  <config xsi:type="map" activeClass="ProtocolCoordinator$Options">
    <default-stack xsi:type="array">
      <handler>org.mundo.net.ActivationService</handler>
      <handler>org.mundo.net.P2PTopicBroker</handler>
      <handler>org.mundo.net.RoutingService</handler>
      <handler>org.mundo.net.BinSerializationHandler</handler>
      <handler>org.mundo.net.BinCollectHandler</handler>
      <handler>SampleHandler1</handler>
      <handler>org.mundo.net.ip.IPTransportService</handler>
    </default-stack>
  </config>
</new-instance>
```

In addition, a service declaration must be made for `SampleHandler1`, so that an instance of `SampleHandler1` is created at startup:

Logging can also be configured in the `node.conf.xml` file:

```
<Logger xsi:type="map">
  <filename>log.txt</filename>
  <console xsi:type="boolean">true</console>
  <default-log-level>INFO</default-log-level>
  <log-levels xsi:type="array">
    <a xsi:type="map"><category>sample</category><log-level>FINEST</log-level></a>
  </log-levels>
</Logger>
```

This configuration contains the following parts:

- You can use the `filename` option to write log output to a file.
- `default-log-level` defines the default log level.
- The `log-levels` section allows to override the default log level setting for certain logger names (=category).

For more information about logging configuration, please refer to Logging Configuration?

Step 2: Setting the MIME type for upward processing

In the first example, packets pass our handler on the sender side, but not on the receiver side. In order that upward processing of messages works correctly, the MIME type of the handler must be registered with the

ProtocolCoordinator.

In addition, the MIME type of a message must be changed always when a message travels down the stack on the sender side. On the receiver side, the type must be changed back accordingly when a message travels up the stack.

```
import org.mundo.rt.IMessageHandler;
import org.mundo.rt.Message;
import org.mundo.rt.Service;
import org.mundo.rt.Logger;
import org.mundo.net.ProtocolCoordinator;
import org.mundo.net.AbstractHandler;

/**
 * This is one of the simplest possible handlers.
 *
 * @author erwin
 */
public class SampleHandler2 extends AbstractHandler
{
  public SampleHandler2()
  {
  }
  /**
   * Called on initialization of the service.
   */
  @Override
  public void init() // Service
  {
    super.init();
    log.fine("init");
    // Register our MIME Type with the protocol coordinator
    ProtocolCoordinator.register(mimeType, this);
  }
  /**
   * Called on shutdown of the service.
   */
  @Override
  public void shutdown() // Service
  {
    log.fine("shutdown");
    super.shutdown();
  }
  /**
   * Called when a packet travels down the stack.
   */
  public boolean down(Message msg)
  {
    log.fine("down: " + msg.getBlob("all", "bin").size());
    // Set our MIME Type before sending the message
    msg.setType(mimeType);
    return emit_down(msg);
  }
  /**
   * Called when a packet travels up the stack.
   */
  public boolean up(Message msg)
```

```
  {
    log.fine("up: " + msg.getBlob("all", "bin").size());
    // Change the MIME Type back. In this example, we assume that the next
    // handler is always mc-bincoll. Note that this assumption cannot be
    // made in general!
    msg.setType("message/mc-bincoll");
    return emit_up(msg);
  }

  private static final String mimeType = "message/sample2";
  private Logger log = Logger.getLogger("sample");
}
```

Step 3: Adding custom headers

The handler presented in Step 2 always assumes that the next higher handler is `mc-bincoll`. However, this assumption cannot be made in the general case.

To write a more generic handler, the previous MIME type of the message must be stored in a custom header. This header must be created when a packet is passed down. On the receiver side, the MIME type must be restored using information from the custom header.

```
import org.mundo.rt.IMessageHandler;
import org.mundo.rt.Message;
import org.mundo.rt.Service;
import org.mundo.rt.Logger;
import org.mundo.rt.TypedMap;
import org.mundo.net.ProtocolCoordinator;
import org.mundo.net.AbstractHandler;

/**
 * Example handler 3
 */
public class SampleHandler3 extends AbstractHandler
{
  public SampleHandler3()
  {
  }
  /**
   * Called on initialization of the service.
   */
  @Override
  public void init() // Service
  {
    super.init();
    log.fine("init");
    // Register our MIME Type with the protocol coordinator
    ProtocolCoordinator.register(mimeType, this);
  }
  /**
   * Called on shutdown of the service.
   */
  @Override
  public void shutdown() // Service
  {
```

```java
    log.fine("shutdown");
    super.shutdown();
  }
  /**
   * Called when a packet travels down the stack.
   */
  public boolean down(Message msg)
  {
    try
    {
      log.fine("down: " + msg.getBlob("all", "bin").size());

      // Add our own header to the message
      TypedMap hdr = new TypedMap();
      // Store the current MIME type of the message in our header
      hdr.putString("type", msg.getType());
      putHeader(msg, "sample", hdr);

      // Set our MIME type
      msg.setType(mimeType);

      return emit_down(msg);
    }
    catch(Exception x)
    {
      log.exception(x);
    }
    // Tell the caller that we dropped the packet
    return false;
  }
  /**
   * Called when a packet travels up the stack.
   */
  public boolean up(Message msg)
  {
    try
    {
      log.fine("up: " + msg.getBlob("all", "bin").size());

      // Get our header
      TypedMap hdr = getHeader(msg, "sample");
      // Restore the previous MIME type
      msg.setType(hdr.getString("type"));

      return emit_up(msg);
    }
    catch(Exception x)
    {
      log.exception(x);
    }
    // Tell the caller that we dropped the packet
    return false;
  }

  private static final String mimeType = "message/sample3";
  private Logger log = Logger.getLogger("sample");
```

```
}
```

Step 4: Sending reply messages

Protocol handlers also often need to send control messages as a reply to received packets. The following example program demonstrates how to send an acknowledge message back to the sender from the `up`-handler.

```
import org.mundo.rt.IMessageHandler;
import org.mundo.rt.Message;
import org.mundo.rt.Service;
import org.mundo.rt.Logger;
import org.mundo.rt.TypedMap;
import org.mundo.net.ProtocolCoordinator;
import org.mundo.net.AbstractHandler;

/**
 * This handler demonstrates how to send back reply messages.
 */
public class SampleHandler4 extends AbstractHandler
{
  public SampleHandler4()
  {
  }
  /**
   * Called on initialization of the service.
   */
  @Override
  public void init() // Service
  {
    super.init();
    log.fine("init");
    // Register our MIME Type with the protocol coordinator
    ProtocolCoordinator.register(mimeType, this);
  }
  /**
   * Called on shutdown of the service.
   */
  @Override
  public void shutdown() // Service
  {
    log.fine("shutdown");
    super.shutdown();
  }
  /**
   * Called when a packet travels down the stack.
   */
  public boolean down(Message msg)
  {
    try
    {
      log.fine("down: " + msg.getBlob("all", "bin").size());

      // Add our own header to the message
      TypedMap hdr = new TypedMap();
```

```
      // Store the current MIME type of the message in our header
      hdr.putString("type", msg.getType());
      // The packet contains a message
      hdr.putString("request", "message");
      putHeader(msg, "sample", hdr);

      // Set our MIME type
      msg.setType(mimeType);

      return emit_down(msg);
    }
    catch(Exception x)
    {
      log.exception(x);
    }
    // Tell the caller that we dropped the packet
    return false;
  }
  /**
   * Called when a packet travels up the stack.
   */
  public boolean up(Message msg)
  {
    try
    {
      log.fine("up: " + msg.getBlob("all", "bin").size());

      // Get our header
      TypedMap hdr = getHeader(msg, "sample");
      // Get the request type
      String req = hdr.getString("request");
      if ("message".equals(req))
      {
         // Send an acknowledgement message
        TypedMap ack = new TypedMap();
        ack.putString("request", "ack");
        Message ackMsg = new Message();
        putHeader(ackMsg, "sample", ack);
        ackMsg.setType(mimeType);
        sendReply(msg, ackMsg);

        // Restore the previous MIME type
        msg.setType(hdr.getString("type"));
        return emit_up(msg);
      }
      else if ("ack".equals(req))
      {
         log.fine("ack received");
         return true;
      }
      log.warning("received unknown request: "+req);
    }
    catch(Exception x)
    {
      log.exception(x);
```

```
    }
    // Tell the caller that we dropped the packet
    return false;
  }

  private static final String mimeType = "message/sample4";
  private Logger log = Logger.getLogger("sample");
}
```

Step 5: Getting information about routes

A node can communicate with an arbitrary number of remote peers and it is often necessary to maintain per-peer state (packet sequence numbers, etc.). The ID of a remote node can be obtained as follows:

- Use `getRoute(msg)` to obtain the route object.
- `getRoute(msg).remoteId` obtains the GUID of the remote node.
- (Note that MundoCore supports multiple routes between two nodes, e.g., one using TCP and one using UDP.)

The following example program shows how to get the destination or source node ID of a message.

```
import org.mundo.rt.IMessageHandler;
import org.mundo.rt.Message;
import org.mundo.rt.Service;
import org.mundo.rt.Logger;
import org.mundo.rt.TypedMap;
import org.mundo.net.ProtocolCoordinator;
import org.mundo.net.AbstractHandler;

/**
 * This handler demonstrates how to duplicate a packet.
 */
public class SampleHandler5 extends AbstractHandler
{
  public SampleHandler5()
  {
  }
  /**
   * Called on initialization of the service.
   */
  @Override
  public void init() // Service
  {
    super.init();
    log.fine("init");
    // Register our MIME Type with the protocol coordinator
    ProtocolCoordinator.register(mimeType, this);
  }
  /**
   * Called on shutdown of the service.
   */
  @Override
  public void shutdown() // Service
  {
    log.fine("shutdown");
    super.shutdown();
```

```
}
/**
 * Called when a packet travels down the stack.
 */
public boolean down(Message msg)
{
  try
  {
    log.fine("down: " + msg.getBlob("all", "bin").size());

    // It is often necessary to keep connection state. Such state
    // should be associated with the ID of the remote node.
    log.finest("sending to node: "+getRoute(msg).remoteId);

    // Add our own header to the message
    TypedMap hdr = new TypedMap();
    // Store the current MIME type of the message in our header
    hdr.putString("type", msg.getType());
    putHeader(msg, "sample", hdr);

    // Set our MIME type
    msg.setType(mimeType);
    return emit_down(msg);
  }
  catch(Exception x)
  {
    log.exception(x);
  }
  // Tell the caller that we dropped the packet
  return false;
}
/**
 * Called when a packet travels up the stack.
 */
public boolean up(Message msg)
{
  try
  {
    log.fine("up: " + msg.getBlob("all", "bin").size());

    // It is often necessary to keep connection state. Such state
    // should be associated with the ID of the remote node.
    log.finest("receiving from node: "+getRoute(msg).remoteId);

    // Get our header
    TypedMap hdr = getHeader(msg, "sample");
    // Restore the previous MIME type
    msg.setType(hdr.getString("type"));

    return emit_up(msg);
  }
  catch(Exception x)
  {
    log.exception(x);
  }
```

```
    // Tell the caller that we dropped the packet
    return false;
  }

  private static final String mimeType = "message/sample5";
  private Logger log = Logger.getLogger("sample");
}
```

Writing custom message brokers

The programs described in the following sections can also be found in the `samples/handlers` subdirectory of the distribution or in the CVS.

A message broker is a special kind of a protocol handler. Hence, we start by examining the first three steps explained in the section Writing custom protocol handlers.

The following sections discuss the implementation of a simple broker.

Handling BCLProvider events

A central concept of MundoCore is its communication micro-broker, which is implemented by the Basic Communications Layer Provider (BCLProvider). The BCLProvider implements a channel-based publish/subscribe system that enables local services to communicate with each other.

Message brokers are now responsible for forwarding all non-local messages from the BCLProvider to remote nodes and vice-versa. As a first step, the broker registers for events from the BCLProvider. Whenever a local service subscribes, unsubscribes, advertises, or unadvertises, our new broker will be notified. The following program registers for BCLProvider events and generates log output on such events.

```
import org.mundo.rt.IMessageHandler;
import org.mundo.rt.Message;
import org.mundo.rt.TypedMap;
import org.mundo.rt.Service;
import org.mundo.rt.Publisher;
import org.mundo.rt.Subscriber;
import org.mundo.rt.Signal;
import org.mundo.rt.Logger;
import org.mundo.rt.IBCLProvider;
import org.mundo.net.ProtocolCoordinator;
import org.mundo.net.AbstractHandler;

/**
 * This is the skeleton for a message broker.
 * @author Erwin Aitenbichler
 */
public class DumbTopicBroker
        extends AbstractHandler
        implements IBCLProvider.ISignal {
  public DumbTopicBroker()
  {
  }
  /**
   * Called on initialization of the service.
   */
  @Override
  public void init() // Service
{
    super.init();
    log.fine("init");
    // Register our MIME Type with the protocol coordinator
    ProtocolCoordinator.register(mimeType, this);
  }
  /**
   * Called on shutdown of the service.
   */
```

```
@Override
public void shutdown() // Service
{
  log.fine("shutdown");
  super.shutdown();
}
/**
 * Called when a packet travels down the stack.
 */
public boolean down(Message msg) // IMessageHandler
{
  try
  {
    log.fine("down: " + msg);

    // Add our own header to the message
    TypedMap hdr = new TypedMap();
    // Store the current MIME type of the message in our header
    hdr.putString("type", msg.getType());
    putHeader(msg, headerChunkName, hdr);

    // Set our MIME type
    msg.setType(mimeType);

    // Pass the message to the next lower handler
    return emit_down(msg);
  }
  catch(Exception x)
  {
    log.exception(x);
  }
  // Tell the caller that we dropped the packet
  return false;
}
/**
 * Called when a packet travels up the stack.
 */
public boolean up(Message msg) // IMessageHandler
{
  try
  {
    log.fine("up: " + msg);

    // Get our header
    TypedMap hdr = getHeader(msg, headerChunkName);
    // Restore the previous MIME type
    msg.setType(hdr.getString("type"));

    return emit_up(msg);
  }
  catch(Exception x)
  {
    log.exception(x);
  }
  // Tell the caller that we dropped the packet
```

```
        return false;
    }
    /**
     * Called when a new subscriber is added by a local service.
     * @param s  the subscriber object.
     */
    public void subscriberAdded(Subscriber s) // IBCLProvider.ISignal
    {
      log.fine("subscriberAdded: "+s);
    }
    /**
     * Called when a subscriber is removed by a local service.
     * @param s  the subscriber object.
     */
    public void subscriberRemoved(Subscriber s) // IBCLProvider.ISignal
    {
      log.fine("subscriberRemoved: "+s);
    }
    /**
     * Called when a new publisher is added by a local service.
     * @param p  the publisher object.
     */
    public void publisherAdded(Publisher p) // IBCLProvider.ISignal
    {
      log.fine("publisherAdded: "+p);
    }
    /**
     * Called when a publisher is removed by a local service.
     * @param p  the publisher object.
     */
    public void publisherRemoved(Publisher p) // IBCLProvider.ISignal
    {
      log.fine("publisherRemoved: "+p);
    }

    private static final String headerChunkName = "dumbtb";
    private static final String mimeType = "message/dumbtb";
    private Logger log = Logger.getLogger("dumbtb");
  }
```

Forwarding messages

The following program shows how to implement a simple channel-based publish/subscribe broker. This broker does not perform any filtering and simply forwards all messages from local services to all remote nodes. Hence, this broker is simple, but totally inefficient.

- The broker makes a wildcard subscription to receive all local messages (in `init`).
- When a local message is received, it adds a parameter chunk to the message. This is necessary to access the name of the target channel later on. The message is then passed to the ProtocolCoordinator? to process the protocol handler chain between application and broker (in `received`).
- The broker broadcasts all received messages to the current zone, i.e., typically all discovered peers (in `down`).
- When a message is received from a remote peer, the address chunk must be reconstructed (in `up`).

```
import org.mundo.net.AbstractHandler;
```

```java
import org.mundo.net.ProtocolCoordinator;
import org.mundo.rt.IBCLProvider;
import org.mundo.rt.IMessageHandler;
import org.mundo.rt.IReceiver;
import org.mundo.rt.Logger;
import org.mundo.rt.Message;
import org.mundo.rt.MessageContext;
import org.mundo.rt.Publisher;
import org.mundo.rt.Service;
import org.mundo.rt.Signal;
import org.mundo.rt.Subscriber;
import org.mundo.rt.TypedMap;

/**
 * This is the skeleton for a message broker.
 * @author Erwin Aitenbichler
 */
public class DumbTopicBroker
        extends AbstractHandler
        implements IBCLProvider.ISignal, IReceiver
{
  public DumbTopicBroker()
  {
  }
  /**
   * Called on initialization of the service.
   */
  @Override
  public void init() // Service
  {
    super.init();
    log.fine("init");
    // Register our MIME Type with the protocol coordinator
    ProtocolCoordinator.register(mimeType, this);
    // Receive all local messages
    session.subscribe("rt", null, this);
  }
  /**
   * Called on shutdown of the service.
   */
  @Override
  public void shutdown() // Service
  {
    log.fine("shutdown");
    super.shutdown();
  }
  /**
   * Receives all local messages. Messages are forwarded to the topmost
   * protocol handler in the protocol stack.
   */
  public void received(Message msg, MessageContext c) // IReceiver
  {
    msg = msg.copyFrame();
    TypedMap pmap = new TypedMap();
    pmap.putString("channel", c.channel.getName());
    msg.put(headerChunkName, "param", pmap);
    ProtocolCoordinator.getInstance().firstDown(msg);
```

```java
}
/**
 * Called when a packet travels down the stack.
 */
public boolean down(Message msg) // IMessageHandler
{
  TypedMap param = msg.getMap(headerChunkName, "param");
  if (param==null)
  {
    log.warning("no "+headerChunkName+" parameter in message");
    return false;
  }
  String channel = param.getString("channel");
  log.fine("down: channel="+channel);

  // Add our own header to the message
  msg = msg.copyFrame();
  TypedMap hdr = new TypedMap();
  hdr.putString("channel", channel);
  msg.put(headerChunkName, "passive", hdr);

  // Set our MIME type
  msg.setType(mimeType);

  // Create a parameter chunk for the routing service and define the whole
  // zone as destination for the message
  TypedMap rs = new TypedMap();
  rs.putString("destType", "zone");
  msg.put("rs", "param", rs);

  // Pass the message to the next lower handler
  return emit_down(msg);
}
/**
 * Called when a packet travels up the stack.
 */
public boolean up(Message msg) // IMessageHandler
{
  // Get our header
  TypedMap hdr = msg.getMap(headerChunkName, "passive");
  if (hdr==null)
  {
    log.warning("no "+headerChunkName+" header in message");
    return false;
  }
  String channel = hdr.getString("channel");
  log.fine("up: channel="+channel);
  // Reconstruct the address chunk
  TypedMap amap = msg.getOrCreateMap("address", "passive");
  amap.put("channel", channel);
  // Also put the session of this broker into the address chunk. This will
  // prevent that the message will be looped back to us
  amap.put("session", session);
  return emit_up(msg);
}
```

```java
  /**
   * Called when a new subscriber is added by a local service.
   * @param s  the subscriber object.
   */
  public void subscriberAdded(Subscriber s) // IBCLProvider.ISignal
  {
    log.fine("subscriberAdded: "+s);
  }
  /**
   * Called when a subscriber is removed by a local service.
   * @param s  the subscriber object.
   */
  public void subscriberRemoved(Subscriber s) // IBCLProvider.ISignal
  {
    log.fine("subscriberRemoved: "+s);
  }
  /**
   * Called when a new publisher is added by a local service.
   * @param p  the publisher object.
   */
  public void publisherAdded(Publisher p) // IBCLProvider.ISignal
  {
    log.fine("publisherAdded: "+p);
  }
  /**
   * Called when a publisher is removed by a local service.
   * @param p  the publisher object.
   */
  public void publisherRemoved(Publisher p) // IBCLProvider.ISignal
  {
    log.fine("publisherRemoved: "+p);
  }

  private static final String headerChunkName = "dumbtb";
  private static final String mimeType = "message/dumbtb";
  private Logger log = Logger.getLogger("dumbtb");
}
```