Elisa Gonzalez Boix (email:egonzale@vub.ac.be, office:10F731)
Jorge Vallejos (email:jvallejo@vub.ac.be, office:10F724)

8 Omnireferences: Building Group Communication Abstractions

Language reference and tutorial available at `http://soft.vub.ac.be/amop`
Lab session material available `http://soft.vub.ac.be/~egonzale`

8.1 Idea

The purpose of this exercise is to experiment with language constructs for programming mobile distributed systems. More concretely, you will implement reflectively in AmbientTalk a language abstraction for group communication. Communicating with groups of remote objects is important when considering cooperation in decentralized networks such as mobile networks. Applications are often interested in communicating with only proximate devices, i.e. devices currently collocated. The application cannot know beforehand the number of such proximate devices as it may vary as the user moves about. In this exercise, we will implement a group communication abstraction called an *omnireference* which denotes all remote objects of the same interface which are available for communication. An omnireference allows 1-to-many interactions, i.e. a client object can interact with various remote objects with a single message send.

8.2 Design of the omnireference

We will implement an omnireference as a special referencing abstraction that transparently discovers and binds to all remote objects of a particular service type. In a previous session, you program part of a mobile game called *Flikken*, in which players broadcast their position to nearby players to orient themselves in the campus and coordinate their movements. Considering that players are service objects exported under the `Player` service type, an omnireference to discover proximate players can be created as follows:

```
def nearbyPlayers := omnireference: Player;
```

The `omnireference:` construct initiates a service discovery request for remote objects exported with the given service tag and immediately returns nil. In this example, the omnireference denotes the set of proximate objects exported to the network with the `Player` service type. Whenever a matching remote object is discovered, it is added to the *receivers* of the omnireference. The receivers of an omnireference are the set of objects that an omnireference binds to.

An omnireference follows the rules of inter-actor message passing and operate asynchronously. In our example, a policeman could use the `nearbyPlayers` omnireference to update his info with nearby team member whenever his position changes as follows:

```
nearbyPlayers <-receivePlayerPosition(myPlayerInfo);
```

When a client object sends a message to the service object, it does not wait for the message to be delivered to the omnireference's receivers. When the omnireference receives the message, it will forward the message to each receiver if the receiver is connected. If the receiver is disconnected upon message reception, it accumulates the message internally, and forwards it whenever it becomes reconnected at a later point in time.

8.3 Implementing the omnireference

We will implement an omnireference starting from a skeleton code shown below (also available in the session material):

```
def makeOmniRefMirror(serviceType){
  mirror: {
    def receivers := // to store available receivers matching the service type

    // override the necessary MOP methods


    // register service discovery of receivers
    whenever: serviceType discovered: { |potentialReceiver|
     //...
    };
  };
};

//public interface of the module
def OmniRefModule := object: {

  //creates an omnireference for a certain service type
  def omnireference: serviceType {
    //...
  };

};
```

An omnireference is implemented as an empty wrapper object which is mirrored by a custom mirror (created by invoking the `makeMultiRefMirror` function). Such mirror should override certain methods of the MOP to implement the explained design as follows:

(a) Complete the skeleton code to implement an omnireference which passes the `testAsyncBasicFunctionality` unit test provided with the skeleton code.

Note: Assume for now that the messages do not require return values.

(b) Client objects should be able to start sending messages to the omnireference before it is bound to any receiver, causing the omnireference to accumulate those messages until a receiver has been discovered. Each time a new receiver is discovered, it should get all message accumulated in the omnireference. Adapt your implementation to provide this functionality (if necessary) and check that your code passes the `testAsyncMessagesAreNotLostWhileUnbound` unit test that checks this.

(c) Extend the omnireference to deal with the results returned from an asynchronous message.

As you know, AmbientTalk provides futures to deal with return values in asynchronous message passing. AmbientTalk also supports *multifutures*, futures that can be resolved or ruined multiple times. A multifuture itself represents a collection of values and/or exceptions as a whole. A multifuture supports the same operations as a regular future. This means it acts as implicit callback for the messages sent and allows the registration of observer closures. There exist three different ways in which observers can be registered with a multifuture:

- **when:** observers are triggered only on the *first* value or exception with which the multifuture is resolved or ruined
- **whenEach:** observers are triggered on *each* value or exception of the multifuture
- **whenAll:** observers are triggered at most *once*, when the multifuture can guarantee that no further results will be gathered.

Make use of the multifutures abstraction (available at `at/lang/multifutures.at`) to handle return values of the messages sent to an omnireference. The `testAsyncMessagesWithReturnValue` unit test is provided with the skeleton code to help you test this functionality.

(d) As said before, when a receiver is discovered, it should get all messages accumulated to the omnireference. But, if a receiver disconnects and reconnects back in time it should only receive those messages which didn't receive during the disconnected time (not all the messages buffered in the omnireference). Add a unitest that checks that functionality.

Hint: The **disconnect: o** construct disconnects all far references to the given object triggering the **when:disconnected** listeners. If 'o' is already a far reference, the construct only disconnects this far reference.