# Abstracting connection volatility through tagged futures

Johan Fabry, Carlos Noguera

# Abstracting connection volatility through tagged futures

Johan Fabry and Carlos Noguera

INRIA Futurs - LIFL, ADAM Team
40, avenue Halley,
59655 Villeneuve d'Ascq, France
{johan.fabry|noguera}@lifl.fr

**Abstract.** The property of connection volatility, fundamental to the ambient intelligence (AmI) domain, makes it hard to develop AmI applications. The underlying reason for this is that the code for this concern is scattered and tangled with the core functionality of the application. In this paper we introduce the abstraction mechanism for connection volatility that we have created, which allows for this concern to be implemented in a non-tangled fashion. The core of our mechanism consists in extending the existing concept of futures with meta-data, i.e. *tags*, to specify values to be used in an offline state. The implementation of our abstraction mechanism, in Java, is called Spoon Graffiti. The meta-data of the futures is described using annotations and the intended behavior is achieved trough source-code processing, using the Spoon annotation processor. As a result of using tagged futures and Spoon Graffiti, the specification of offline behavior of an AmI application can be performed in a non-tangled way, which significantly eases development.

## 1 Introduction

Developing Ambient Intelligence (AmI) software is a non-trivial task. This is because, not only do we need to deal with many of the known issues of distributed systems, e.g., inherent concurrency and network latency, but also we face the fundamental problem of connection volatility. As ambient devices frequently come in and out of range of each other, connections will be constantly established and broken. Connection volatility is therefore a fundamental problem of AmI: whereas in non-AmI programs connections are assumed to be permanent, in AmI the inverse is the norm.

Developing applications that behave correctly in the presence of connection volatility is a difficult task. An important reason for this is that the code for this concern is scattered throughout the application, and tangled with the core functionality of the application. Furthermore, no abstraction mechanisms have yet been developed that provide an adequate amount of support for connection volatility. In this paper we introduce the concept of tagged futures as a valid abstraction mechanism for connection volatility in AmI applications. Tagged futures allow the specification of the offline behavior of the application in a

straightforward and non-tangled manner. Furthermore, our proposal includes support to semi-automatically transition from an online to an offline state, and vice-versa. Again this support is provided at a higher level of abstraction and is not tangled with the core application code.

A number of abstraction mechanisms have previously been developed for connection volatility [?,?,?,?,?,?,?], however, none of these provide adequate support for specifying offline behavior of the application. One such abstraction is the use of futures [?], as proposed in an ambient context by Dedecker et.al. [?]. We can use futures as empty place-holders for return values of network operations. Futures have the important advantage that they do not introduce any tangling of the connection volatility concern in the application. Their downside is however that in a disconnected setting they only allow an application to continue working in a very limited fashion. It is our opinion that the restrictions that are imposed are too strong, as we shall show in Sect. 2.2. We therefore propose to enrich futures, to allow them to be more amply useful. Tagged futures allow metadata, i.e., tags, to be attached to them. This metadata can then specify a mock value to be used during disconnected operations. As we shall show in this paper, the use of such metadata makes futures applicable in a more realistic setting.

## 2 Future Problems

*Futures*, also sometimes referred to as promises, essentially are placeholder values for an as yet undetermined object. When the actual value for that object has been determined, the future is automatically *resolved*. Resolving causes the future to transparently become the new object. Futures can be passed around as if they were the resolved value, without this affecting the behavior of the application. It is only when the future itself is accessed, e.g., through a method call or a field access that the behavior of the application differs. Accesses to a future *block* until the future is resolved. When the future is resolved, any blocked accesses are forwarded to resolved value for the future. An important advantage of futures is that, in the code, they are indistinguishable from the objects for which they are place-holders. As a result, this abstraction for connection volatility does not introduce any tangled code.

We can use futures as return values of network operations, allowing the application to continue to function in a disconnected fashion. As long as the future itself is not accessed, the application will function as normal. However, when a future is accessed, the application will block. The application will only continue after the future has resolved, in other words, only after the network link is established, the remote call has been executed and its return value is known.

### 2.1 The Shopping Application

We will employ a running example to illustrate an important limitation of futures, and show how our proposal can address this limitation. This running example is a shopping list application, a screen shot of which is shown in Fig. 1.

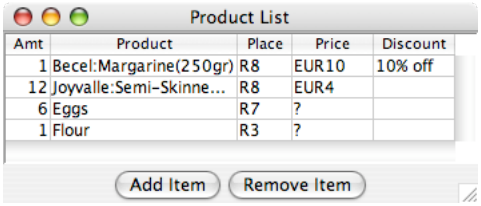


**Fig. 1.** Screen shot of the shopping application when inside a store.

The list can contain two kinds of products: generic products such as eggs and flour, and specific products that also identify a brand and container size. In Fig. 1, the first two items are specific products, and the last two are generic products. When inside a store, the list contains extra information. This is obtained using a network local to the store itself. The location of the products inside the store is shown, and for specific products their price and discounts, if any, are also displayed.

In a first step, we have implemented the shopping application as a non-AmI distributed system, taking care to have a clean modular decomposition of the application. Figure 2 shows the class diagram of this implementation, where we have omitted impertinent classes. The diagram is fairly self-explanatory. The only classes meriting an extra description are **Shop Product Info** and **Specific Info**: These classes contain the extra information for a given product that is displayed when inside a shop. Whenever the user wishes to add an item, the **Shopping List** firstly creates a **Product** or **Specific Product**, depending on the amount of information given. The shopping list then requests the **Shop** for the extra information for that product, and links this to the **Product** before adding the item to the list.

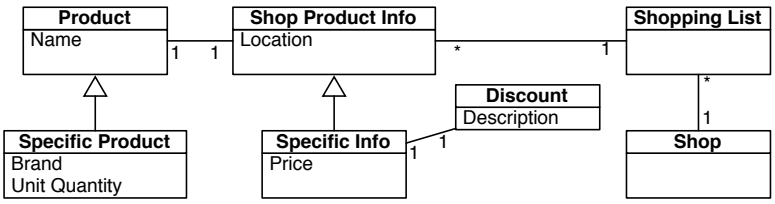


**Fig. 2.** Diagram of relevant classes of the shopping application

### 2.2 Features Missing From Futures

When the shopping list application is not connected to the server, using futures as placeholders for the **Shop Product Info** or **Specific Info** allows it to con-

tinue operating despite no such information being available. The future is linked to the **Product** or **Specific Product**, and the item is added to the shopping list. When entering a shop, the shopping list application will connect to a server and the future will be resolved, allowing the extra information to be shown. Consequently, it seems that futures are indeed a suitable abstraction for dealing with disconnected operations in this context.

The above scenario however does not take into account the behavior of the user interface (UI) shown in Fig. 1. Whenever an item is added to the list, the UI should, of course, reflect this. Therefore, after the item is added, the UI refreshes itself, reading out the required values for the different elements in the grid. When disconnected from the server, some of these values will be contained within futures, e.g., the place of an item. As a result these calls will block, blocking the UI and rendering the application unusable until a network connection is established, which resolves the future.

It is clear that the above behavior is not what a user would expect. It should be possible to add and remove products at all time, regardless of whether the application is connected or not. Furthermore, to enable this, the user will be willing to accept some information not to be available in the list, and to be replaced with mock values. For example, when disconnected, place and price of products may be represented by a question mark, and the discount may be empty. It is however essential that, once a connection has been established, such mock values are replaced with the true values as obtained from the server. Vice-versa, whenever the connection is lost, these mock values should be put in place again. This will allow new values to be obtained from a server when a connection is re-established. In our example, this will allow a user to wander from store to store, and always have the extra information for the current store being shown. When entering a store the connection will be re-established with the server for that store, which entails that the futures will resolve to the data for that store.

## 3 Tagged Futures

It is our intent to allow futures to be useful beyond what is currently possible when faced with connection volatility, as we have discussed above. To achieve this, we propose in this paper to extend futures as follows:

**Mock values:** can be specified as results of accesses to unresolved futures.
**Update mechanism:** when the futures are resolved, interested parties are informed and can take appropriate actions.
**Invalidation mechanism:** reverts a resolved future to its prior form on network disconnects.

The kernel of our proposal lies in adding tags, i.e., metadata to futures. Both the update and invalidate mechanism are a natural consequence of adding this metadata, as we discuss next.

### 3.1 Adding Metadata to Futures

The main contribution we present in this paper is the concept of adding metadata, as tags, to futures. This will alleviate some of the limitations of futures, therefore less restricting the applications' behavior in a disconnected setting.

Concretely, the first kind of metadata we add is mock values. These mock values are specified by the programmer of the application, in the class for which the future is a stand-in. These mock values will then be returned as a result of an access to the future, i.e., a method call or a field read. Note that we consider specifying such mock values as optional: if no mock value is given, the access will simply block.

As a result of this extension of futures, mock values will now be used by other objects in the system. Whenever futures are resolved, these mock values are no longer required and should also be replaced by the real values. Furthermore, any computation that has been performed using the mock values should be invalidated, and re-executed with the real values. To allow this, we propose the use of an update mechanism in addition to future resolution. This mechanism informs objects that use mock values that the future has been resolved. This allows them to perform any necessary updates, as they see fit.

The above two features provide support for a program to change from a disconnected to a connected state. To provide support for the inverse: changing from a connected to a disconnected state, we propose to use an invalidation mechanism. This mechanism invalidates all objects that are the result of the resolution of a future. As a result, these objects revert to their original future. In analogy to network connection, all computation dependent on these, now invalid, objects is invalid. The above update mechanism will again be triggered, allowing necessary updates to be performed.

### 3.2 Futures, Passive Futures, Possible Futures, and Future Observers

Conceptually, our introduction of tagged futures adds four new kinds of objects to a distributed system that serve to handle connection volatility. These new kinds of objects are Futures, Passive Futures, Possible Futures and Future Observers.

*Futures* are placeholders for objects that are unavailable due to the absence of network connections. When the connection is established, futures will automatically resolve to the real value. When the connection is dropped, the real value will automatically revert to the future. Methods and fields of futures may be tagged with mock values, to be returned when these are accessed. If no mock values are given, these accesses block until the future is resolved. In the shopping application, we can use futures for the **Shop Product Info** and **Specific Info** classes. When disconnected, these will return mock values for the location and price of objects, e.g., a question mark.

*Passive Futures* are a simplified version of futures. Passive futures do not have the ability to resolve to the real value when the network connection is established. Instead, passive futures let some other object assume responsibility for their resolution. The object responsible will usually be another future. We introduce passive futures to allow the resolution of multiple related futures to be handled by one coordinating authority. In the shopping application a passive future can be used for the **Discount**. When disconnected it will return an empty description for the discount. Upon connection, futures for **Specific Info** will handle the resolution of associated **Discount** instances.

*Possible Futures* are the objects that are substituted by futures or passive futures when the application is offline. In the shopping application, the classes **Shop Product Info**, **Specific Info** and **Discount** therefore are Possible Futures.

*Future Observers* are objects that may use a mock value of a future. These need to be notified when a future is resolved and also when an object is reverted to a future. This allows them to perform necessary updates. In the shopping application, the shopping list is a future observer. It observes all futures for **Shop Product Info** and **Specific Info** objects, and will refresh the UI after futures are resolved or reverted. As a result, when in a shop the additional information will be shown, and outside of a shop the mock values for this information.

## 4 Implementing Tagged Futures with Spoon Graffiti

We have chosen to implement our proposal using source-code transformations so that tagged futures have a minimal impact at runtime. The system we created is based on the Spoon transformation engine [**?**], and is called Spoon Graffiti[1]. Spoon allows the transformation of a program by means of successive processing rounds. These are implemented as visitors of a model derived from the program's abstract syntax tree. They are directed by the annotations present on various source code elements (classes, methods, fields, etc). Spoon is seamlessly integrated with the Eclipse IDE. This permits our tool to report errors on the definition of the tags in a transparent way, that is, errors can be presented just as compilation errors. This is specially useful when processing annotations that have Java expressions as arguments, as will be presented in the next section.

Thanks to using source-code transformations, the only overhead which remains at runtime is a class that reifies the online or offline state of the application. This minimal infrastructure is dependent on the distribution mechanism used, which currently is Java RMI. This class can however easily be re-implemented for a different distribution mechanism. The bulk of the behavior of the application, with regard to connection volatility, is implemented outside of this infrastructure, and we discuss it next.

[1] Because the future is tagged.

### 4.1 Tagged Futures as Annotations

To add support for connection volatility to a distributed application, a developer adds annotations to the code, as well as a number of additional methods. The use of annotations allows this extra behavior to be added without tangling it with the core behavior of the application.

The behavior of Futures and Passive Futures is realized by modifying the code of the classes of Possible Futures. Modifying the classes thus avoids issues regarding object identity, as the Future is the same object as the Possible Future. The downside of this is that, if the Futures are not passive, Possible Future classes need to implement a method for resolution. Similarly, generic behavior is added to classes that contain the annotations for a Future Observer. Methods that perform the actual update of the observer need to be implemented by the developer. We discuss this in more detail following the five types of annotations we have defined, which are shown in Tab. 1.

| | Type | Optional Argument | Usage |
|---|---|---|---|
| `@Future` | Method, Field | An expression | Future, Passive Future |
| `@Connect` | Method | | Future |
| `@ObservedFuture` | Local Variable | | Future Observer |
| `@Online` | Method | Class | Future Observer |
| `@Offline` | Method | Class | Future Observer |

**Table 1.** Defined annotations with their type and corresponding usage

The `@Future` annotation is added to a method or a field, making the class that contains it a possible future class. If no argument is given to the annotation, calls to that method or accesses to that field when the application is offline will block. Otherwise the result of evaluating the argument expression (which is encapsulated in a string literal) is returned. Note that in possible future class, methods and fields that have no such `@Future` annotations are not modified. This is so to support behavior which is unaffected by the presence of a connection.

Possible Futures that also contain the `@Connect` annotation will be replaced by futures that are not passive. The annotation declares the method that is called to resolve the future. The method will be called when a reconnection occurs. It should act as an initializer for the object: assigning to all fields the values it obtains over the network.

Future observers indicate which possible futures they observe using the `@ObservedFuture` annotation, which is given to instance variables of methods. At runtime, all values assigned to these variables will be considered as being observed. Upon network connection, these futures will be resolved by a call to their method annotated with `@Connect`. If a future is observed by multiple observers, this method will only be called once. Also, if a future is not observed, the method will not be called, i.e., it will not be resolved.

Future observers declare their interest in notification of future resolution or reverting to futures by annotating methods with `@Online` respectively `@Offline`. These methods should take one argument, of the type of the superclass of all possible future classes. After a future resolves respectively a possible future reverts, these methods will be called with the just changed object as argument.

## 5 The Shopping Application Revisited

We now illustrate how tagged futures and our implementation using Spoon Graffiti provides for a usable and non-tangled abstraction of connection volatility. We do this by revisiting the shopping application, of which the modifications are outlined in Fig. 3.

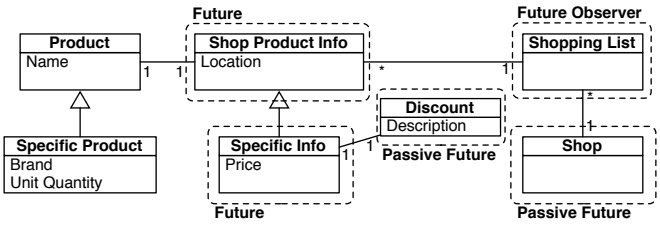


**Fig. 3.** Offline shopping application, with object kinds defined by the annotations.

We do not treat the entire application here, but instead focus on significant sections. We first discuss the Specific Info and Discount classes, before talking about the Shopping List and the Shop. Below is an excerpt of the code for the Specific Info and Discount classes:

```
public class SpecificInfo extends ShopProductInfo {
   private String price;   private Discount disc;

   @Future("\"TBD\"")
   public String getPrice() { return price; }
   public String getDiscount() {return disc.reduction_type; }

   @Connect
   public void become(){
      SpecificInfo realSPI = (SpecificInfo)
         Shop.getProductInfo(target_product);
      location = realSPI.location;
      price = realSPI.price; discount = realSPI.discount; }
   [... constructors omitted ...] }
public class Discount {
   @Future("\"TBD\"")
   public String reduction_type;
```

```
    [... constructors omitted ...] }
```

This code first shows how the `@Future` annotation can be applied to both methods and fields. Second it illustrates a use of a method without a `@Future` annotation tag, to delegate to a **Discount** object, which itself takes care of disconnected operations. A similar case is in **Shop Product Info**, which allows a reference to the product name to be obtained by the shopping list. Third, this code provides an example of how to resolve a future, in the `become()` method. This method obtains a new **Specific Info** from the server, and simply copies over all the relevant data, including the **Discount** object. As a result, futures for **Discount** objects can be passive. We omitted in the listing above the two constructors for each class: one for a normal instantiation used when online, and one for an 'empty' instantiation used when offline.

```
public class ShoppingList implements TableModel {
    public void addProduct(ShopProductInfo prod, Integer amt){
        @ObservedFuture ShopProductInfo p2 = prod;
        products.add(prod); prod_amounts.add(amt);
        this.changed(prod); }

    @Online
    @Offline
    private void changed(ShopProductInfo prod){
        for(TableModelListener listener : tml)
            listener.tableChanged(new TableModelEvent(this)); }
    [ ... fields and table model methods omitted ... ] }
public class Shop {
    @Future("ShopProductInfo.createEmptySPI(prod)")
    public static ShopProductInfo getProductInfo(Product prod){
        [... body omitted ...] }
    [... server implementation omitted ...]
}
```

The Shopping List class implements the Java Swing Table Model class, which allows it to be used in a Swing table, as shown in Fig. 1. Adding a product to the list, in the `addProduct` method implies that futures for it are observed, which is declared through the `@ObservedFuture` annotation. Future resolution, reverting to futures, as well as adding and removing products all trigger the `changed()` method. This method simply refreshes the UI.

The Shop is a passive future, that in an offline state returns empty Shop Product Info objects when queried, as indicated by its `@Future` annotation. The convenience method called in the argument of the annotation creates empty Shop Product Info or Specific Info objects. As the Shop itself contains no state that needs to be updated when the connection goes online or offline, it can be represented by a passive future when offline. Note that by having the Shop itself as a tagged future we do not need any extra mechanism for the creation of futures when the application is offline.

This concludes the revisit of the shopping application. When this application is offline, the extra information for a product will be displayed as `TBD`. When the

application goes online, the extra information will automatically be obtained from the server and displayed. To implement this behavior, no code needed to be added to, or changed in, methods that provide the core functionality of the application. As a result, this implementation shows that tagged futures, as implemented in Spoon Graffiti, are indeed a non-tangled abstraction that provides adequate support for connection volatility.

## 6 Related Work

Related work can be subdivided in to major categories: distributed languages and distributed middleware.

Using futures as return values of a synchronous call has previously been used in languages such as ABCL/f [**?**] and Argus [**?**] (where they are known as promises). However in both these languages accessing a future blocks, which yields the problem we have elaborated in Sect. 2.2. In the AmbientTalk language [**?**,**?**], calls are asynchronous, and a special `when` construct is used to delay execution of a block of code until the future is resolved. Again, as accessing an unresolved future blocks, this yields the problem described in Sect. 2.2. Furthermore, we consider the use of the `when` construct to produce code which is more tangled than our solution.

A significant amount of research has been performed on middleware for mobile networks, however to the best of our knowledge no system has yet been constructed that provides abstractions specifically for connection volatility in an AmI context. The most appropriate middleware solution seems to be Rover [**?**], as it allows for queuing of a remote message call in conjunction with weak replica management. While this can conceivably be used to implement behavior similar to the used of tagged futures, this would firstly not be encapsulated as one abstraction and secondly be unlikely to be tangled code. Similar to Rover, Coda [**?**] and XMiddle [**?**] also provide support for replica management but have no specific abstraction mechanism for connection volatility.

## 7 Conclusions and Future Work

In this paper we have proposed an extension to futures to provide better support for connection volatility in AmI applications. To the best of our knowledge, this is the first work performed to provide such an abstraction, allowing the specification of offline behavior in a non-tangled way.

Our proposal add tags to futures, specifying mock values to be used when offline, together with an update and invalidation mechanism for these mock values. We have discussed how we have implemented these extensions, and have shown though an example how they cleanly add support for connection volatility. We believe that tagged futures are an elegant abstraction for connection volatility which will significantly ease development for AmI applications.

Future work consists of exploring other kinds of metadata, e.g., instead of immediately reverting to a future when going offline, specifying a timeout, indicating a time-span in which this data is valid when offline. Furthermore, we consider adding support for writing to futures, so that when going online this data is written to the server. This amounts to replica management and will therefore entail a conflict detection and resolution mechanism, as in Coda [**?**], Rover [**?**] or XMiddle [**?**].

Spoon Graffiti, the full code of the shopping application example, as well as other examples can be obtained from: `http://spoon.gforge.inria.fr`