Towards a Domain-Specific Aspect Language for Leasing in Mobile Ad hoc Networks

Elisa Gonzalez Boix * Thomas Cleenewerk Jessie Dedecker Wolfgang De Meuter

Programming Technology Lab
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium
{egonzale,tcleenew,jededeck,wdmeuter}@vub.ac.be

Abstract

Leasing provides a robust mechanism to manage reclamation of remote objects in mobile ad hoc networks. However, applying the leasing semantics on each remote object reference places a considerable burden on developers. Low-level leasing management details can be abstracted away as much as possible by means of dedicated language support. This paper focusses on the software engineering issues that arise using language support for leasing. We observe that the concerns dealing with leasing are inherently crosscutting and argue in favour of a modularization of such concerns in an aspect. We propose a domain-specific aspect language (DSAL) for leasing which provides dedicated means to express the leasing concerns separately from the base functionality.

Categories and Subject Descriptors D.2.3 [*Software Engineering*]: Coding Tools and Techniques

General Terms Languages, Design

Keywords mobile ad hoc networks, leasing, domain-specific languages, aspect-oriented programming

1. Introduction

In mobile ad hoc networks, distributed programming is substantially complicated by the intermittent connectivity of the devices in the network and the lack of any centralized coordination facility. To deal with volatile connections, remote object references should tolerate network disconnections: a disconnected remote reference may always become reconnected when the network connection is restored. Because it is impossible to distinguish a transient network failure from a permanent (network or machine) failure, the lifetime of the remote object reference should be limited such that the remote object can eventually be reclaimed if the network failure persists.

Leasing provides a robust mechanism to manage reclamation of remote objects in a fault-tolerant fashion [10]. A *lease* denotes the right to access a resource for a limited amount of time. In a distributed object-oriented system, remote object references play the role of the lease and the objects they refer to play the role of the resource. In other words, client objects from other machines can reference remote objects by means of *leased object references*. However, applying the leasing semantics on each remote object reference places a considerable burden on developers. Rather than offering leased object references as a general library abstraction, we have chosen a language approach such that low-level memory management concerns can be cleanly incorporated into more high-level abstractions, decreasing the mental overhead for the developer. In previous work, we have designed dedicated language support for leasing [5].

This paper focusses on the software engineering issues that arise using such language support for leasing. We observe that leased object references are inherently crosscutting: developers must encode the distributed memory management semantics (i.e. how long a leased reference lasts and what happens once it is expired) at different places in the application. In this paper, we argue that concerns dealing with leasing should be cleanly separated from the functional code. Expressing the leasing semantics separately also allows developers to define the same leasing semantics to groups of objects which share certain properties and to express dependencies amongst leased object references in a structured way. We subsequently introduce a domain-specific aspect language (DSAL) which provides dedicated means to express the leasing concerns by encapsulating them as an aspect. Before describing the DSAL, we first discuss the crosscutting nature of leased object references and the domain specific semantics particular to leasing.

2. Background

To delimit the scope of our work, we first introduce some terminology and concepts from the area of distributed object-oriented computing. We assume an object-oriented system where objects can be *exported*, which makes them available on the network. Objects can either be exported explicitly, by means of a service discovery mechanism, or implicitly, by passing them as a parameter or return value in a message sent to a remote object. We denote such remotely accessible objects as *server* objects. Server objects can be referenced from other machines by means of *leased object references*. A leased object reference is a remote object reference that transparently grants access to a remote server object for a limited period of time. When a client first references a server object, a leased object reference is created and associated to the server object. From that moment on, the client accesses the server object transparently via the leased reference until it expires. In [5], we instantiated such leased object reference model in a distributed object-oriented pro-

* Author funded by the Prospective Research for Brussels program of the Brussels Hoofdstedelijk Gewest.

Workshop DSAL'08 April 1, 2008, Brussels, Belgium.

gramming language designed for mobile ad hoc networks called AmbientTalk [9]. The following sections briefly introduce AmbientTalk and the concrete instantiation of our leasing model.

2.1 AmbientTalk in a Nutshell

AmbientTalk is a prototype object-oriented distributed language. Consider the definition and use of a simple `Song` object in AmbientTalk:

```
def Song := object: {
  def artist := nil;
  def title := nil;
  def init(artist, title) {
    self.artist := artist; self.title := title;
  };
  def play() { /* play the song */ };
};
def s := Song.new("Garbage", "Stupid Girl");
```

In this example, a song object is assigned to the variable `Song`. A song object has two fields namely, a constructor (called `init` in AmbientTalk) and a method `play`. Sending `new` to an object creates a copy of that object, initialised using its `init` method.

AmbientTalk is a concurrent actor-based language [1]. AmbientTalk actors are based on the *communicating event loops* model [7] in which each actor is conceived as an event loop which *owns* a set of regular objects. Objects owned by the same actor communicate using sequential method invocation (expressed as `o.m()`) or using asynchronous message passing (expressed as `o<-m()`). Objects owned by different actors can only send asynchronous messages to one another.

2.2 Language constructs for leasing in AmbientTalk

AmbientTalk provides three different language constructs[1] for creating a basic leased object reference that expires after a certain timeout and two variations which transparently adapt their lease time under certain circumstances [4].

```
lease: timeout for: object
renewOnCallLease: timeout for: object
singleCallLease: timeout for: object
```

As shown above, a basic leased reference is created by means of the `lease` function which takes as parameter an initial time period and a server object to which the leased reference grants access. The first variant is a *renew-on-call* leased reference that automatically prolongs the lease upon each method call received by the remote object. The second variant is a *single-call* leased reference that automatically revokes the lease upon performing a method call on the remote object. Such leases are useful for objects which adhere to a *single call* pattern such as futures [6]. In AmbientTalk, an asynchronous message send immediately returns a *future* which is a placeholder for the actual return value. Once the return value is computed, it replaces the future object; the future is then said to be *resolved* with the value. A future actually acts as an implicit callback object which is remotely accessed only once with the computed return value.

3. Motivation

The main motivation for building a DSAL for leasing stems from the analysis of using our previously described language constructs in mobile ad hoc networking applications. Before arguing for leasing as an aspect, we first introduce our running example and then observe the crosscutting nature of the leasing concerns.

[1] These constructs are executed at the server side which then hands out the proper leased object reference to a client object.

3.1 Running example: the Mobile Music Player

We consider the case of the mobile music player [8], a small yet typical collaborative ad hoc networking application, as our running example. This application is meant to be used on the PDA or the cellular phone of the user. A mobile music player contains a library of songs. When two people using the music player enter one another's personal area network (defined for example by the bluetooth communication range of their cellular phones), the music players exchange their music library's list. After the exchange, the music player can calculate the percentage of songs both users have in common. If this percentage exceeds a threshold, the music player can e.g. warn the user that someone with a similar musical taste is nearby.

In this application, each music player is modelled as an actor which explicitly exports an *interface* object that can be used by other music players to start a communication session to exchange libraries by sending it the `openSession` message. The interface object implements this message as follows:

```
1  def openSession(sessionCallback) {
2    // store sender's music library in a set
3    def senderLib := Set.new();
4    def session := renewOnCallLease: minutes(10) for:
5      object: {
6        def downloadSong(artist, title) {
7          senderLib.add(Song.new(artist, title));
8          "ok"; //tell sender song was correctly received
9        };
10       def endExchange() {
11         revoke: session; //takes the session offline
12         // calculate match percentage with my library
13         def matchRatio := calcMatchRatio(senderLib);
14         if: (matchRatio >= THRESHOLD) then: {
15           // notify user of a match
16         };
17       };
18     };
       when: session expired: {
19       //clean the downloaded library of songs
       }
20   session; // return the session object
21 };
```

The `openSession` method asynchronously returns a new `session` object that implements two methods: `downloadSong` and `endExchange` which are used by a remote music player to send song information and to signal the end of the library exchange, respectively. The `session` object is clearly only relevant within the context of a single music library exchange. If – due to a persistent network partition or a crash – the exchange cannot be completed, this object and the resources it transitively keeps alive should be eventually reclaimed. To this end, the `session` object is exported using a lease for 10 minutes which is automatically renewed each time it receives a message. As long as the exchange is active, i.e. `downloadSong` messages are received, the session remains active. The leased reference is revoked either explicitly when a client sends the `endExchange` message to indicate the end of the library exchange, or implicitly if the lease time has elapsed. Once a leased reference expires, the **when-expired** observers will be triggered allowing client and server objects to properly react and release additional resources. In this example, a session clears the `senderLib` which stores the incoming songs.

3.2 Leasing as a Crosscutting Concern

The above code snippet illustrates a simple example of the usage of a leased object reference. The crosscutting nature of leasing already emerges when looking at only one single leased object reference. The implementation of the `session` object is tangled with two

concerns: a set of statements that defines the core functionality of the session object and a set of statements belonging to the leasing concern applied to the session object. In fact, conceptually the leasing concern encapsulates a number of sub-concerns: the establishment of the leased reference (line 4), its explicit revocation (line 11) and how the expiration of the leased reference is handled (lines 20-22). These sub-concerns are clearly scattered throughout the implementation of the session object.

Leasing concerns not only crosscut a single object but also groups of server objects sharing the same leasing semantics. As a concrete example, consider again the case of futures. As previously mentioned, a future acts as an implicit callback object which only serves to process the reply of an asynchronous message send. A future is thus implicitly exported in a message send to a client object which holds the only remote reference to it. If this reply does not arrive after some period of time, the future object should become garbage. In our running example, a remote player starts the exchange of their music library's list by sending the asynchronous openSession message as follows:

```
def openSessionFuture :=
    remotePlayer<-openSession()@Due(minutes(10));
when: sessionFuture becomes: { |session|
  // open session with remote player
} catch: TimeoutException using: { |e|
  system.println("unable to set up session.");
}
```

The timeout for the implicit lease on the openSessionFuture future object is set by means of the @Due(timeout) annotation. The when:become:catch function registers two event handlers with a future: the becomes closure is triggered when the future is resolved and the catch closure is triggered with a TimeoutException when the future's lease expires due to a timeout. Note that for each asynchronous message send returning a value, the application code gets polluted with (1) the annotation determining the timeout of the implicit lease on the future and (2) a catch closure defining how to handle the expiration of such lease. This clearly puts extra burden on developers which have to repeatedly encoded these two concerns along the entire application on every future message send. Although the concrete timeout to apply to a future may depend on the computational context of a message send, futures conceptually form a group of server objects sharing common leasing semantics: they either expire due to a timeout or upon the reception of a resolve message with the return value.

The scattering of leasing concerns also obfuscates the dependencies that exist among them. In our running example, once a music player receives a session object, it will start the exchange of songs by asynchronously sending downloadSong messages. The sender waits for the acknowledgment of a song information before sending the next one as follows:

```
def downloadSongFuture := session<-downloadSong(
  song.artist,song.title)@Due(leaseTimeLeft: session);
when: downloadSongFuture becomes: { |ack|
   // recursive call to send the rest of the songs
} catch: TimeOutException using: { |exception|
  // stopping exchange with remote player
};
```

Note that the future object attached to the downloadSong message sends can be discarded when either the acknowledgement arrives or the session times out. Therefore the timeout period to use should be derived from the session lease timeout, rather than picking an arbitrary timeout value. In order to properly encode these dependencies it is better to express leasing concerns separately rather than to have them scattered along the application.

3.3 Leasing as an Aspect

In the previous section, we observed that the usage of the language constructs which form the leasing concern, is clearly scattered along the application and tangled with the base functionality. We advocate to separate leasing concerns from the application's functionality and modularize them into a separate unit. Treating leasing concerns separately has the following advantages:

- The leasing concern is actually a complex concern which can be conceptually subdivided in a number of sub-concerns. Bringing them together in a single module allows developers to keep track of the life cycle of the leased object references.
- Leasing mechanisms are typically applied in a per-object basis. Expressing leasing concerns separately allows developers to reuse leasing semantics shared amongst a group of server objects and to better modularize the application of recursive leasing patterns.
- Dependencies between leasing concerns can be explicitly expressed in one module improving the readability of the code.
- Leasing concerns may evolve independently from the base functionality. For example, self-adaptive leasing algorithms have been proposed to dynamically vary lease periods in response to the system size [2]. Leasing semantics can be altered depending on hardware characteristics such as network latency. Expressing leasing concerns in a separate module allows developers to deal with unanticipated changes on the leasing semantics without having to adapt the entire application.

We believe that using aspects, the scattering of the definition of leased object references can be cleanly captured in a pointcut description. Aspects have been successfully applied to the memory management concerns at the level of the implementation of a virtual machine [3]. We propose the usage of aspects for distributed memory management concerns (i.e. leasing) at a programming language level. The basic idea is to grow our current language constructs into a domain-specific aspect language (DSAL) which encapsulates the leasing concerns in one module improving conciseness and readability of the code implementing the base functionality.

There are several reasons for proposing a DSAL rather than opting for a general purpose aspect language. First, all the semantics involved with leasing concerns are specific to the domain of memory management for remote references. Leasing concerns affect the process of establishing a leased reference, managing its life cycle and reacting to its expiration. Secondly, our language targets AmbientTalk applications: developers can express the memory management semantics of remote object references used in the base application. The language is a domain-specific language for distributed programming which employs an event-driven concurrency model. As such, an aspect language on AmbientTalk needs also to take into account several domain-specific concepts such as asynchronous communications and *ambient acquaintance management* (i.e. the discovery and management of proximate devices and their hosted services).

Based on our expertise in distributed programming languages for mobile ad hoc networking applications and the usage analysis of our language constructs for leasing, we describe a number of characteristics that a domain-specific language for leasing should support:

- The DSAL should allow developers to specify the leasing semantics concerning the management of the full life cycle of leased object references in a separate module. The management of leased object references should be open enough to make possible the definition of other actions than the standard renewals or revocations of leases.

- It should also provide means to define reusable leasing semantics which can be applied to a group of server objects.
- Dependencies between leasing concerns should be explicitly expressed.
- Due to the event-driven nature of mobile ad hoc networks, AmbientTalk adheres to a set of well-defined characteristics such as asynchronous communications and ambient acquaintance management. These characteristics (which are described in [9]) resulted in an event-driven programming language. This event-driven nature should be reflected in the join point model.

4. A DSAL for Leasing

This section proposes a DSAL for leasing. Our language declares the leasing semantics of a number of server objects. A leasing aspect encapsulates the leasing sub-concerns previously identified, i.e. the semantics of the leased object reference and the management of its life cycle since it gets established until its expiration, in a single module.

Aspects An aspect is a special kind of object which expresses leasing concerns on the base functionality. It consists of a pointcut which identifies the objects that will be passed with a certain leasing semantics and a set of advices managing the established leased references. Such advices can be sub-divided in two categories: default advices that control the creation and expiration of the lease, and custom advices which express other actions on the lease, e.g. revocation or renewal of the lease. The following code excerpt shows the definition of the aspect corresponding to the leased object reference for the `session` object introduced in the previous section.

```
1  def  leasedSession := aspect: {
2    //pointcut definition
3    capture: pass as: `session
4      and: { session.isTaggedAs(`sessionType) };
5    //default advices
6    on: referencing do: {
7       RenewalOnCallLease.new(minutes(10));
8    };
9    on: expired do: {
10     system.println("session " +remoteUser+ " expired");
11     senderLib.empty();
12   };
13   //custom advice
14   on: session.receive(msg)
15     and: {msg.methodName==`endExchange} do: {revoke()};
16 }
```

The above code defines an aspect by means of the **aspect:** construct and subsequently binds it to a local variable named `leasedSession`.

Pointcut Descriptors The pointcut of a leasing aspect is defined by means of the **capture:** function which takes as parameter an event to be observed. In this example, **capture:** observes the `pass` event which is triggered by the interpreter when an object is serialized (and becomes remotely accessible). Other events can be observed in the pointcut definition such as the discovery of a remote object. This allows developers to establish a leased reference when a server object is explicitly published via the service discovery.

The `pass` event returns an object which can be bound to a local variable by means of the **as:** construct. Additional conditions can be applied to a bound `session` object by means of the **and:** construct, e.g. in the above example whether the object is tagged with the `sessionType` type tag.

Advices Following the pointcut definition, a set of advices defines the management of the lease life cycle. They are expressed by means of the **on:do:** construct, which takes as parameter an event and the action to execute. **referencing** and **expired** are two default events that every leased reference exhibits. They reify how to handle resp. the creation and expiration of a leased object reference. Each leasing aspect can also provide custom default events. The advice on referencing establishes the type of the leased object reference, e.g, a *renew-on-call* lease while the advice on expired clears the `senderLib` as previously explained [2]. Lines 14-16 define a custom advice which triggers the explicit revocation of the leased reference upon the reception of the `endExchange` message.

Note that our DSAL can only issue two actions that modify the state of a leased object reference: a revoke (applied by means of the `revoke()` function as shown in line 16) and a renewal (applied by means of the `renewal(timeout)` function). Further research about changes of state of a leased reference is required.

4.1 Applying the DSAL to the running example

We described the syntax and semantics of our DSAL for leasing by means of the `leasedSession` aspect. Although the introduction of this aspect makes possible to factor out the lines of code for the identified three sub-concerns from the base code, this aspect is specific to the `session` object. The leasing semantics of futures are a more general and complex concern to modularise. The code excerpt below illustrates how a reusable leasing aspect for futures could look like:

```
def  future := aspect: {
  capture: send as: `msg
    and: { msg.isTaggedAs(`FutureMessage) };
  def timeout;
  def init(aTimeout) { timeout := aTimeout };
  on: referencing do: { SingleCallLease.new(timeout)}
}
```

As previously explained, all future objects share the following leasing semantics: they expire due to a timeout or upon the reception of a single message (either a `resolve` or `ruin` message). Therefore, the advice on referencing establishes a *single-call* leased reference. Note that the concrete timeout to apply to a future object may depend on the computational context of the message send. To this end, the future aspect defines a custom constructor which initialises the `timeout` field. In the context of our running example, the following aspect defines the future attached to the `downloadSong` messages.

```
def  downloadSongFuture := extend: future with: {
  capture: super.capture
    and: { msg.getMethodName().equals(`downloadSong)};
  def init() {
    super.init(msg.getReceiver().getLeaseCounter()) };
  on: expired do: { //stop the exchange }
}
```

In the example, the `downloadSongFuture` extends the leasing semantics of the `future` aspect by means of the **extend:with:** function which creates a new aspect (object) whose super slot is automatically set to the given parent. As such, the pointcut definition of the `downloadSongFuture` aspect uses its parent pointcut definition (accessed by means of the **super** variable) and adds an extra constraint, i.e. the message send name.

As said in 3.2, the future objects attached to each `downloadSong` message send can be discarded when either the acknowledgement arrives or the session times out. In order to express this

[2] We assume that the aspect is in the scope of the definition of the `session` object and it can thus see the `senderLib` variable.

dependency, the `downloadSongFuture` initializes the timeout field with the lease counter specified in the `session` object to which the `downloadSong` message is sent. The `downloadSongFuture` aspect defines how to handle the expiration of the lease since this may depend on a particular sequence of message sends. Note that the `downloadSongFuture` aspect is applied to a single message send, but future aspects could as well be applied to a group of asynchronous message sends, for example different messages involved in a database transaction.

5. Summary

This paper observes the crosscutting nature of the leasing concerns and gives several reasons why techniques for separation of concerns are a good candidate to implement dedicated language support for leasing. We take a language-oriented approach and advocate to express leasing concerns as a special domain of aspects using a DSAL. It simplifies the definition of the leasing semantics and the description of relationships between leasing concerns by encapsulating them into a separate module. This also allows developers to deal with unanticipated changes on the leasing semantics without adapting the entire application.

We are currently designing the DSAL described in this paper as an extension of the meta-object protocol of AmbientTalk. At a conceptual level, a number of challenging issues need also to be thoroughly explored such as how to access base variable definitions within the aspect or how to deal with the addition of new types of events that are not present yet in the system.

References

[1] AGHA, G. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[2] BOWERS, K., MILLS, K., AND ROSE, S. Self-adaptive leasing for jini. In *Inter. Conf. on Pervasive Computing and Communications (PERCOM)* (2003), IEEE Computer Society, pp. 539–542.

[3] GIBBS, C., AND COADY, Y. Aspects of memory management. In *Proceedings of the 38th Annual Hawaii International Conference on System Sciences (HICSS)* (2005), IEEE Computer Society, p. 275.2.

[4] GONZALEZ BOIX, E., VALLEJOS VARGAS, J., VAN CUTSEM, T., DEDECKER, J., AND DE MEUTER, W. Context-aware leasing for mobile ad hoc networks. In *3rd workshop on OT4AmI co-located at ECOOP* (2007).

[5] GONZALEZ BOIX, E., VAN CUTSEM, T., DEDECKER, J., AND DE MEUTER, W. Language support for leasing in mobile ad hoc networks. Tech. Rep. 07-08, PROG, VUB, 2007.

[6] HALSTEAD, JR., R. H. Multilisp: a language for concurrent symbolic computation. *ACM Trans. Program. Lang. Syst. 7*, 4 (1985), 501–538.

[7] MILLER, M., TRIBBLE, E. D., AND SHAPIRO, J. Concurrency among strangers: Programming in E as plan coordination. In *Symp. on Trustworthy Global Computing* (2005), Springer, pp. 195–229.

[8] VAN CUTSEM, T., DEDECKER, J., AND MEUTER, W. D. Object-oriented coordination in mobile ad hoc networks. In *9th International Conference on Coordination Models and Languages (COORDINATION)* (2007), vol. 4467 of *LNCS*, Springer-Verlag, pp. 231–248.

[9] VAN CUTSEM, T., MOSTINCKX, S., ELISA GONZALEZ BOIX, DEDECKER, J., AND DE MEUTER, W. Ambienttalk: object-oriented event-driven programming in mobile ad hoc networks. In *XXVI International Conference of the Chilean Computer Science Society (SCCC)* (2007), IEEE Computer Society.

[10] WALDO, J. Constructing ad hoc networks. In *IEEE Inter. Symposium on Network Computing and Applications (NCA)* (2001), p. 9.