A Leasing Model to Deal with Partial Failures in Mobile Ad hoc Networks

Elisa Gonzalez Boix *, Tom Van Cutsem **, Jorge Vallejos ***, Wolfgang De Meuter, and Theo D'Hondt

Programming Technology Lab - Vrije Universiteit Brussel - Belgium
{egonzale,tvcutsem,jvallejo,wdmeuter,tjdhondt}@vub.ac.be

Abstract. In mobile ad hoc networks (MANETs) many partial failures are the result of temporary network partitions due to the intermittent connectivity of mobile devices. Some of these failures will be permanent and require application-level failure handling. However, it is impossible to distinguish a permanent from a transient failure. Leasing provides a solution to this problem based on the temporal restriction of resources. But to date no leasing model has been designed specifically for MANETs. In this paper, we identify three characteristics required for a leasing model to be usable in a MANET, discuss the issues with existing leasing models and then propose the *leased object references* model, which integrates leasing with remote object references. In addition, we describe an implementation of the model in the programming language AmbientTalk. Leased object references provide an extensible framework that allows programmers to express their own leasing patterns and enables both lease holders (clients) and lease grantors (services) to deal with permanent failures.

Key words: mobile ad hoc networks, partial failures, leasing, remote object references, language design

1 Introduction

The recent progress in the field of wireless technology has proliferated a growing body of research that deals with *mobile ad hoc networks* (MANETs): networks composed of *mobile* devices connected by *wireless* communication links with a limited communication range. Such networks have two discriminating properties, which clearly set them apart from traditional, fixed computer networks [13]: intermittent connectivity of the devices in the network (called the *volatile connections* phenomenon) and lack of any centralized coordination facility (called the *zero infrastructure* phenomenon). The volatile connections phenomenon states

* Author funded by Prospective Research for Brussels program of the Institute for the encouragement of Scientific Research and Innovation of Brussels (IWOIB-IRSIB).
** Postdoctoral Fellow of the Research Foundation - Flanders (FWO).
*** Author supported by the VariBru project of the ICT Impulse Programme of IWOIB-ISRIB and the MoVES project of the Interuniversity Attraction Poles Programme of the Belgian State, Belgian Science Policy.

that a disconnection should not be treated as a "failure" by default: due to the limited communication range of wireless technology, devices may move out of earshot unannounced. The resulting disconnections are usually *transient*: the devices may meet again requiring their connection to be re-established and allowing their collaboration to be continued where they left off. The zero infrastructure phenomenon states that it is more difficult to rely on server infrastructure (e.g. a name server for service discovery) since devices spontaneously join and disjoin the network due to their physical mobility.

Our research focuses on distributed programming language support for mobile ad hoc networks. In this paper, we explore support to deal with the effects engendered by partial failures. Due to the above mentioned phenomena, it can be expected that many partial failures in MANETs are the result of temporary network partitions. However, not all network partitions are transient, e.g. a remote device has crashed or has moved out of the wireless communication range and does not return. Such *permanent* failures should also be dealt with by means of compensating actions, e.g. application-level failure handling code. Because it is impossible to distinguish a transient from a permanent failure [15], some arbitrary criteria should be agreed upon so that a device can determine when the *logical* communication with another remote device has terminated.

Leasing [5] provides a solution to this problem based on the temporal restriction of resources [15]. A *lease* denotes the right to access a resource for a specific duration that is negotiated by the owner of a resource and a resource claimant (called the *lease grantor* and *lease holder*, respectively) when the access is first requested. The advantage of leasing is that allow both lease grantor and holder to distinguish a transient from a permanent failure by approximating permanent failures as disconnections that exceed the agreed time interval. However, to date no leasing model has been designed to operate in a mobile ad hoc network setting where leasing needs to (1) be combined with computational models that deal with transient failures, (2) provide different leasing patterns for the different kinds of collaboration that can be set up in MANETs, and (3) allow both lease holders and grantors to deal with permanent failures.

This paper proposes a leasing model specially designed for MANETs. Our contribution lies in integrating leasing as a special kind of remote object reference, called *leased object references*, which tolerate both transient and permanent failures. The leased object references model incorporates different leasing patterns at its heart and allows programmers to schedule clean-up actions at both client and server side of a leased reference upon expiration. In addition, we describe a concrete instantiation of such a model in a distributed programming language, called *AmbientTalk* [13], as an extensible framework in which many leasing patterns can be expressed.

2 Leasing in Mobile Ad hoc Networks

In this section, we discern a number of criteria that need to be exhibited by a leasing model for dealing with partial failures in MANETs and describe how ex-

isting leasing models for distributed computing fail to deal with them. We derive these criteria from the analysis of an illustrative ad hoc networking application.

Throughout this paper, we assume an object-oriented system where devices can be abstracted as a container for a set of objects implementing some functionality. Objects can be *exported* in the network either explicitly or implicitly by passing them as parameter or return value in a message sent to a remote object. We denote such remotely accessible objects as *service* objects. Service objects can be referenced from other machines by means of *remote object references*.

2.1 Running example: the Mobile Music Player

Consider a music player running on mobile devices. The music player contains a library of songs. When two people using the music player enter one another's personal area network (defined by for example the bluetooth communication range of their cellular phones), the music players set up a so-called ad hoc network and exchange their music library's list (not necessarily the songs themselves). After the exchange, the music player can calculate the percentage of songs both users have in common. If this percentage exceeds a certain threshold, the music player can e.g. warn the user that someone with a similar musical taste is nearby.

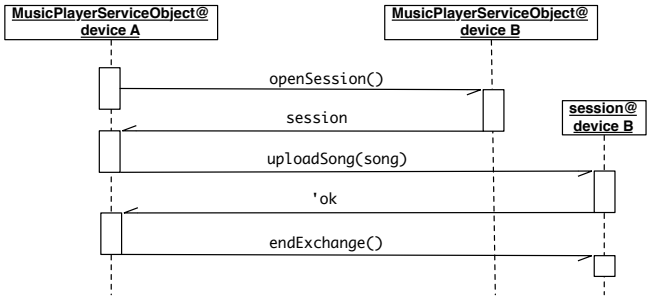

Fig. 1. The music library exchange protocol

Figure 1 gives a graphical overview of the music library exchange protocol modeled via a distributed object-oriented system where communication between devices is asynchronous. The figure depicts the stages of the protocol from the point of view of the music player on the device `A`. In fact, this protocol is executed simultaneously on both devices. Once both devices have discovered each other, the music player running on `A` asks the remote peer `B` to start a session to exchange its library index by sending an `openSession` message. In response to it, the remote peer returns a new `session` object which implements methods that allow the remote music player to send song information (`uploadSong`) and to signal the end of the library exchange (`endExchange`).

2.2 Analysis

This section describes a set of criteria that need to be exhibited by a leasing model to be used for dealing with partial failures in MANETs. While some of the above described criteria can be observed in different leasing models for distributed computing platforms and frameworks, to the best of our knowledge, no single leasing model exhibits all of the criteria presented in this section.

Leasing an Intermittent Connection. A lease denotes a time restriction on the logical connection between lease holder and grantor. At the software level, a logical connection is represented by a communication link. Because of the volatile connections phenomenon in MANETs, communication links are often intermittent: devices often disconnect for an unknown period of time and then reconnect. However, that does not imply that the logical connection should be terminated. In our running example, once the service objects that represent the music player application have discovered one another, they need to set up a session to exchange their music libraries. Such a session is *leased*, such that both music players can gracefully terminate the exchange process in the face of a persistent disconnection. However, if a user moves temporarily out of range, the resulting transient disconnection should not cause the exchange to fail immediately as the user may reconnect. A leasing model for MANETs must take this into account: the disconnection of a device does not indicate that resources associated with the logical connection can already be cleared, since the communication link may be restored later.

Leasing Patterns. Mobile ad hoc networking applications are conceived as a collection of several devices setting up a *collaboration*. As different kinds of collaboration can be set up, different kinds of leasing patterns are also possible. In our running example, devices collaborate in the exchange of their music library indexes. As long as the exchange is active, i.e. `uploadSong` messages are received, the session should remain active. The session could be thus exported using a lease which is automatically renewed each time it receives a message. The lease should be then 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. Other collaborations may involve objects adhering to a *single call* pattern such as callback objects. For example, in asynchronous message passing schemes, callback objects are often passed along with a message in order for service objects to be able to return values. These callback objects are typically remotely accessed only once by service objects with the computed return value. In this case, a lease which is automatically revoked upon a method call is more suitable. A leasing model for MANETs should be flexible enough to allow developers to specify different leasing patterns to manage the lifetime of leases.

Symmetric Expiration Handling. Leasing allows lease grantors to remain in control of the resource by maintaining the right to free the resource once the lease expires. In MANETs, both communicating parties should be aware of a lease expiration since it allows them to detect a permanent disconnection. Once a lease expires, both lease holder and grantor should be able to

properly react and do some clean-up actions in order to gracefully terminate their collaboration. In our running example, the `session` object is clearly only relevant within the context of a single music library exchange. If the exchange cannot be completed (e.g. due to a persistent network partition), the session object and the resources allocated during the session, e.g the partially uploaded library index, should be eventually reclaimed. A leasing mechanism for MANETs should allow both lease holder and lease grantor to deal with the termination of their logical connection.

2.3 Related Work

Leases were originally introduced as a fault-tolerant technique in the context of distributed file cache consistency [5]. Jain and Kircher introduced the concept of leasing as a software design pattern to simplify resource management in [6]. In distributed object-oriented systems, leasing has been used to describe the lifetime of remote objects in frameworks like Java RMI and .NET Remoting [10], and as a general abstraction for resource management in platforms like CORBA [2] and Jini [14]. In this section, we further evaluate these leasing mechanisms in the light of the criteria for a leasing model in MANETs.

Java RMI In Java RMI leases are tightly coupled to the distributed garbage collector (DGC) and are used as a way to manage the lifetime of remote objects in a fault-tolerant fashion. Although Java RMI integrates leasing with remote references, leases are only used to reclaim unused connected remote references. In fact, its leasing model is combined with a synchronous communication model (RPC) which does not decouple objects in time or synchronization [3], which makes it unsuitable for MANETs. Leases are transparent to the programmer as a part of the DGC and the lease duration is controlled by means of a system property. If developers need to deviate from the default leasing behaviour, the system provides an interface to the DGC based on so-called *dirty* and *clean* calls. Leasing patterns need to be built on top of these low-level operations. For example, automatic renewal of leases can be only accomplished by making low-level dirty calls on the remote references. Expiration handling is not provided upon lease expiration in Java RMI. If a client attempts to access a service object whose lease expired, an exception is raised. This allows clients to schedule some clean-up actions in the exception handler, but forces them to install exception handlers on every remote call.

.NET Remoting framework The .NET Remoting framework incorporates leasing in combination with the concept of *sponsorship* for managing the lifetime of remote objects [9]. Sponsors are third-party objects which are contacted by the framework when a lease expires to check if that party is willing to renew the lease. Clients can register a sponsor on a lease and thus decide on the lifetime of server objects. Similar to JavaRMI, the .NET Remoting framework leases are used to reclaim unused connected remote references. In contrast to the simplicity of the language constructs offered in JavaRMI, the .NET Remoting

framework incorporates a leasing pattern at the heart of its design. Leases are automatically extended on every call on the remote object by the time specified in the `RenewOnCallTime` property. If that property is not set, lease renewal can be achieved by registering a sponsor. Variations on the integrated pattern need to be built on top of sponsor and lease interface abstractions. The lease interface provides methods for overriding some leasing properties (e.g. `RenewOnCallTime`), renewing the lease, and the registration of sponsors. However, no means are provided for explicit revocation of a lease. Expiration handling is not provided either in the .NET Remoting framework. Although the system does indeed contact sponsors upon lease expiration, there are no guarantees that the system will contact the sponsor of a specific client as it may ask several sponsors until it finds one willing to renew the lease.

CORBA Leasing has been introduced to CORBA as a technique for resource management [2]. A broad definition of resource is adopted: a resource can be practically any CORBA entity. In order to provide reusable leasing functionality for different CORBA-based applications, leasing is modeled as a dedicated CORBA service [2]. A resource is expected to implement two methods used by leases to start and stop the use of the resource. A *resource claimant* has to obtain a lease from a lessor in order to use such a resource. A lease offers methods to be renewed and revoked. Two types of leases are granted depending on the type of claimants. *Observed* claimants receive a lease which observes the claimant so that if it terminates, the lease gets cancelled. The object is periodically queried to detect if it is still alive. Due to the volatile connections phenomenon, such leases do not seem appropriate for MANETs: the claimants may only be disconnected temporarily, causing the lease to be cancelled erroneously. *Notified* resource claimants receive a lease which notifies the claimant as soon as it expires. The lease is then automatically renewed once at server side to give the claimant sufficient time to renew the lease if necessary. Leasing patterns need to be built on top of the above mentioned architecture, e.g. automatic renewal of leases can be accomplished by making explicit renew calls on the lease interface. Expiration handling can be achieved at client side using notified claimants.

Jini Jini is a framework built on top of Java which allows clients and services to discover and set up an ad hoc network. Jini provides support to deal with the fact that clients and services may join and leave the network at any time, unannounced. Leasing was introduced to allow clients and services to leave the network gracefully without affecting the rest of the system. However, Jini relies on the communication model of Java RMI [15]. This means that transient disconnections are not supported, i.e. a disconnection blocks the connection between objects. Although Jini's architecture is flexible enough to accommodate a leasing model which takes into account intermittent connections, to the best of our knowledge, Jini does not implement this functionality. Jini advocates the use of leasing to mediate the access to any kind of resource: objects, files, certificates that grant the lease holder certain capabilities or even the right to request for some actions to execute while the lease is valid. By default, leases have been

only integrated in the lookup service. Services get a lease when they advertise themselves with the lookup service which must be explicitly renewed; if they cannot, the lookup service will remove the service advertisement such that it does not provide stale information. Jini provides a data structure for the systematic renewal of a set of leases. Leasing patterns can be built using a lease renewal service to implement the protocol to communicate with the remote server. Expiration handling can be achieved at client side by registering an event listener on a lease renewal set. When the lease is about to expire, an expiration warning event is generated notifying all registered listeners for that set.

	Leasing an Intermittent Connection	Leasing Patterns	Symmetric Expiration Handling
Java RMI	N (RPC)	N (need to be built)	N (no notification upon lease expiration)
.NET Remoting Framework	N (RPC)	Y (integration of a leasing pattern)	N (notification not guaranteed)
CORBA	N (RPC)	Y (using notified and observer claimants)	Y (only at client side)
Jini	N (reliance on JavaRMI)	N (need to be built)	Y (only at client side)

Table 1. Summary of related work

Table 1 summarizes our related work. We observe that no single approach adheres to all of the criteria for a leasing model in MANETs. This shortcoming forms the main motivation for our work. In particular, no approach takes into account the intermittent connections criterion since they rely on synchronous communication by RPC, which is not designed for MANETs. However, these approaches provide a foundation on which we base our leasing model.

3 Leased Object References

To address *all* the criteria discussed in the previous section, we introduce our leasing model for MANETs where remote object references play the role of the lease and the service objects they refer to play the role of the resource, leading to the concept of *leased object references*.

A leased object reference is a remote object reference that transparently grants access to a service object for a limited period of time. When a client

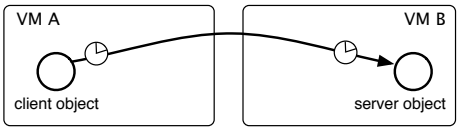

Fig. 2. A leased object reference

first references a service object, a leased reference is created and associated to the service object. From that moment on, the client accesses the service object transparently via the leased reference until it expires. Figure 2 illustrates an allocated leased reference. Each end point of the leased reference has a timer initialized with a time period which keeps track of the lease time left. When the time period has elapsed, the access to the service object is terminated and the leased reference is said to *expire*. The lifetime of leased references can be explicitly controlled by renewing or revoking them before they expire. Once a leased reference expires, *both* the client object and service object know that access to the service object is terminated. Leased object references provide dedicated parameter-passing semantics to ensure that all remote interactions to a service object are subject to leasing.

Leasing an Intermittent Connection In order to abstract over the transient disconnections inherent to MANETs, a leased reference decouples the client object and the service object it refers to in time. This means that a client object can send a message to the service object even if the leased reference is disconnected at that time. Client objects can only send messages to service objects *asynchronously*: when a client object sends a message to the service object, the message is transparently buffered in the leased reference and the client does not wait for the message to be delivered [4]. Figure 3 shows a diagram of the different states of a leased reference. When the leased reference is connected and active, i.e. there is network connection and the lease has not yet expired, it forwards the buffered messages to the remote object. While disconnected, messages are accumulated in order to be transmitted when the reference becomes reconnected at a later point in time. When the lease expires, the client loses the means of accessing the service object via the leased reference. Any attempt in using it will not result in a message transmission since an expired leased reference behaves as a *permanently* disconnected remote reference.

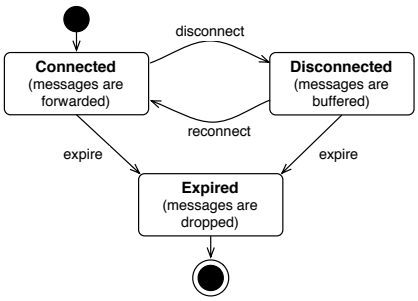

Fig. 3. States of a leased object reference

[4] We are not the first ones on providing buffering of messages, this behaviour can be also found in the Rover toolkit [7]

Leasing Patterns Leased object references incorporate two leasing variants on leased object references which transparently adapt their lease period under certain circumstances. The first variant is a *renew-on-call* leased reference which automatically prolongs the lease upon each method call received by the service object. This pattern has been inspired by the `renewOnCall` property of the .NET Remoting framework [9]. As long as the client uses the service object, the leased reference is transparently renewed by the interpreter. The second variant is a *single call* leased reference which automatically revokes the lease upon performing a successful method call on the service object. Such leases are useful for objects adhering to a *single call* pattern, such as callback objects. As previously explained, callback objects are often used in asynchronous message passing schemes in order for service object to be able to return values. These callback objects are typically remotely accessed only once by service objects with the computed return value. Other variants are definitely possible and can be built on top of the leased object reference abstraction as we illustrate later in the implementation of a concrete instantiation of the leased object references model.

Symmetric Expiration Handling We adopt a Jini-like solution for expiration handling based on event listeners which can be registered with a leased reference at both client and server side. When the reference expires, the registered listeners are notified asynchronously. This allows client and service objects to treat a failure as permanent (i.e. to detect when the reference is permanently broken) and to perform appropriate compensating actions. At server side, this has important benefits for memory management. Once all leased references to a service object have expired, the object becomes subject to garbage collection once it is no longer locally referenced.

Because both sides of a leased reference have a timer, no communication with the server is required in order for a client to detect the expiration of a leased reference. However, having a client-side and server-side timer introduces issues of clock synchronisation. Keeping clocks synchronised is a well known problem in distributed systems [12]. This issue is somewhat more manageable with leases since they use time intervals rather than absolute time and the degree of precision is expected to be of the magnitude of seconds, minutes or hours. Once the leased reference is established, the server side of the reference periodically sends the current remaining time by piggybacking it onto application-level messages. At worst, the asynchrony causes a leased reference to be temporarily in two inconsistent states: either the client-side of the reference expires while the server-side is still active, or the client-side of the reference is active while the server-side expired. In the first case, a client will not attempt a lease renewal and thus, the server-side timer will eventually expire as well. In the second case, when a client requests a lease renewal, the server will ignore it and the client-side timer will expire soon thereafter. When the server-side timer is expired, the client perceives the remote object as disconnected due to a network failure.

4 Leased Object References in AmbientTalk

We have implemented leased object references in AmbientTalk, an object-oriented programming language designed for distributed programming in mobile ad hoc networks [13]. Before describing the concrete instantiation of the leased object reference model, we first briefly introduce AmbientTalk. We will use the mobile music player example introduced in Section 2.1 to illustrate the language and the language support for leased object references.

4.1 AmbientTalk in a Nutshell

AmbientTalk is a prototype-based object-oriented distributed language. The following code excerpt shows the definition 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("Mika", "Relax");
```

A prototypical song object is assigned to the variable `Song`. A song object has two fields; 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.

Distributed programming AmbientTalk is a concurrent actor-based language [1]. AmbientTalk's actors are based on the *communicating event loops* model of the E language [11] in which each actor *owns* a set of regular objects. Objects owned by the same actor communicate using sequential message sending (expressed as `o.m()`), as in Java. Objects owned by different actors can only send asynchronous messages to one another (expressed as `o<-m()`). This ensures by design that all distributed communication is asynchronous.

Additionally, an actor can explicitly *export* objects which can then be discovered by remote objects. Objects can acquire a remote reference to an object when the remote object is implicitly parameter-passed as argument or return value in a asynchronous message sent. Additionally, an actor can explicitly *export* objects representing certain services which can then be discovered by remote objects. In AmbientTalk, a service object is always exported together with a *service type*, a descriptor used to categorise which objects export what kinds of services. One can define event handlers that are triggered whenever a remote object of a certain service type has become available in the network. AmbientTalk's remote references by default mask partial failures: messages may be sent to a disconnected reference, where are buffered until the remote reference becomes reconnected.

4.2 Leasing in AmbientTalk

We now describe a concrete instantiation of the leased object reference model introduced in Section 3 in the AmbientTalk language. A description of how leased references have been implemented in AmbientTalk is postponed until Section 5.

Our language support features three different language constructs for creating leased object references which correspond to basic leased object references and the two variants described in Section 3. The most basic form of a leased reference is created by the **lease:for:** construct which requires two parameters: an object corresponding to the service object to which the leased reference grants access, and an initial time period (in milliseconds). In our running example, the `session` object that represents the exchange process between two music players should be subject to leasing in order for both music players to gracefully terminate the exchange process in the presence of network failures. Such a `session` object can be leased as follows:

```
def leasedRef := lease: minutes(10) for: session
```

The leased reference created with the **lease:for:** construct is valid for the given time period unless a renewal or revocation is explicitly issued. After this time period, access to the session is terminated and the leased reference expires. Explicit manipulation of the lifetime of a leased reference is provided by means of the **renew:** and **revoke:** constructs. The **renew:** construct requests a prolongation of the specified leased reference with a new interval of time which can be different than the initial time while the **revoke:** construct cancels the given leased reference. Cancelling a lease is in a sense analogous to a natural expiration of the lease, but it requires communication between the client and server side of the leased reference.

Note that the **lease:for:** construct and the other two constructs (described in the next section) are executed at the server side. The virtual machine hosting the service object hands out the proper leased object reference to a client object. In our running example, a music player application asks a remote peer to start a session to exchange its library index by sending it an `openSession` message which returns a new session object. The music player can then send song information to the remote peer via the obtained leased reference as follows:

```
session<-uploadSong("Mika", "Relax", ...);
```

Because we chose to model leasing by means of a special kind of remote object reference, the client can use the leased reference as if it were the service object itself. The use of leasing is thus made transparent to the client.

4.3 Language constructs for Leasing Patterns

In order to create renew-on-call and single-call leases explained in Section 3, we provide the **renewOnCallLease:for:** and **singleCallLease:for:** constructs,

respectively. The **renewOnCallLease:for:** construct creates a leased reference which is automatically prolonged on every remote method invocation on the service object. When no renewal is performed due to a network partition or in the absence of utilization, the leased reference expires once its lease time elapses. In the running example, once a music player establishes a session with another music player to exchange their music library index, the session should remain active as long as the exchange is active, i.e. `uploadSong` messages are received. A renew-on-call lease can be used for the `session` object to model that kind of collaboration as follows:

```
def openSession(sessionCallback) {
  def senderLib := Set.new(); // store sender's music library in a set
  def session := renewOnCallLease: minutes(10) for: ( object: {
    def uploadSong(artist, title, ackCallback) {
      senderLib.add(Song.new(artist, title));
      ackCallback<-ok(); // tell sender that song was successfully received
    };
    def endExchange() {
      revoke: session;
      def matchRatio := calculateMatchRatio(senderLib);
      if: (matchRatio >= THRESHOLD) then: { // notify user of match };
    };
  } );
  sessionCallback<-receive(session); // return the session object
};
```

As previously mentioned, the `openSession` message is sent by a music player to a remote peer which returns a session object that can be used to start a library exchange. A session implements the `uploadSong` method to send song information and the `endExchange` method to signal the end of the library exchange. The session object is exported using a lease for 10 minutes which is automatically renewed each time it receives a message. The renewal time applied on every call is the initial interval of time specified at creation. 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. Since the session object was only referred to by the leased reference, it can be reclaimed once the lease has expired. Any resources it transitively occupied such as the partially uploaded library of songs (i.e. `senderLib`) can be reclaimed as well.

The **singleCallLease:for:** construct allows developers to create leased references that remain valid for only a single call. In other words, the leased reference expires after the service object receives a single message. However, if no message has been received within the specified time interval, the leased reference also expires. As shown in the code above, the `sessionCallback` is parameter-passed in the `openSession` message to asynchronously receive a `session` object. A single-call lease can be used for unexporting this callback object upon receipt of the `receive` message as follows:

```
remotePlayer<-openSession(
  singleCallLease: minutes(10) for: ( object: {
    def receive(session) {
     /* start to exchange of its library via the session (explained later) */
```

```
    }
}));
```

A lease time of 10 minutes is specified to wait for the reply of the `openSession` message. If a disconnection would occur after the message was sent but before the `receive` reply was received, the session object could have already been allocated. Since a session's lease only lasts 10 minutes by default, it does not make sense to wait any longer for the reply. If the session callback's lease expires, the music library exchange terminates before it was actually started, requiring no additional cleanup code.

4.4 Integrating leasing with future-type message passing

In the previous section, we used an explicit callback object to return the result of the `openSession` asynchronous message. This is motivated by the fact that in AmbientTalk, an asynchronous message send has no return value by default (i.e. it returns **nil**). To avoid forcing programmers to rely on explicit, separate callback methods to obtain the result of an asynchronous computation, future-type message passing [16] was introduced in AmbientTalk. Futures are a classic technique to reconcile asynchronous message sends with return values, by making an asynchronous send immediately return a *future* object. A future is a placeholder for the return value of an asynchronous message send which allows the sender of an asynchronous message to access the return value of that message at a later point in time [16]. In our running example, we have used callback objects to circumvent the lack of return values in asynchronous message sends. With the introduction of futures, explicit callbacks are no longer necessary: the future serves as an implicit callback. The asynchronous invocation of `openSession` can be rewritten using futures as follows:

```
def sessionFuture := remotePlayer<-openSession();
when: sessionFuture becomes: { |session|
  // open session with remotePlayer
}
```

We have integrated leasing into futures by parameter-passing a future attached to an asynchronous message via a singe-call lease which either expires due to a timeout or upon the reception of the computed return value. The timeout for the implicit single-call lease on a future can be set by annotating the asynchronous message with a **@Due** annotation as follows:

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

In AmbientTalk, it is possible to register a block of code with a future which is executed asynchronously when the future becomes resolved with a return value

by means of the **when:becomes:catch:** construct. If the future is resolved to a proper value, the block of code in the **becomes:** argument is applied. If the asynchronously invoked method raises an exception, the **catch:** argument is applied to the exception. A `TimeoutException` is raised when the future's lease expires. If the future is resolved, the `session` variable stores a leased object reference to the remote session object. A music player then sends all of its own songs one by one to this session object as follows:

```
def sessionFuture := remotePlayer<-openSession()@Due(minutes(10));
when: sessionFuture becomes: { |session|
    def iterator := myLib.iterator(); // iterate over own music library
    def sendNextSong() {  // auxiliary function to send each song
      if: (iterator.hasNext()) then: {
        def song := iterator.next();
        def ackFut := session<-uploadSong(song.artist,
                                  song.title)@Due(leaseTimeLeft: session);
        when: ackFut becomes: { |ack|
          sendNextSong(); // recursive call to send next song info
        }catch: TimeoutException using: { |e|
          notification("stopping exchange: " + e)
        };
      } else: { session<-endExchange(); };
    };
    sendNextSong();
  };
};
```

As already observed, the `uploadSong` method can be directly sent to the session variable storing the leased reference as if it were the service object itself since leasing is transparent to the client. The auxiliary function `sendNextSong` sends the music player's songs one by one to the remote `session` object. This serial behaviour is guaranteed because each subsequent `uploadSong` message is only sent after the previous one returned an acknowledgement. Since the return value of the `uploadSong` message is only useful in the context of the current library exchange session, it only makes sense to wait for the future resolution for the remaining duration of the session (which can be acquired from a leased reference by means of the **leaseTimeLeft:** construct). If the future's lease expires, the library exchange is stopped without requiring additional cleanup code.

The integration of futures and leasing by means of the **@Due** annotation, illustrates that low-level memory management concerns (e.g. the callback objects) can be cleanly incorporated into more high-level abstractions, decreasing the mental overhead for the developer.

4.5 Supporting Expiration Handling

In order to allow both client and service objects to properly react to the expiration of a leased reference and schedule clean-up actions, the **when:expired:** construct is provided. A music player can detect when a session with a remote music player expires as follows:

```
when: session expired: {
  system.println("session timed out.");
```

```
  // clean the partially received music library
}
```

The construct takes as parameters a leased reference and a block of code that is asynchronously triggered upon the lease expiration. In the example, **when:expired:** is installed at the server side so that if the exchange cannot be completed the resources a session transitively keeps alive (i.e. the `senderLib` set storing incoming songs) can be cleared.

Note that in the integration of leasing with futures, specifying a **catch:** block for the **TimeoutException** is equivalent to install a **when:expired:** observer on the future's (server-side) lease.

5 Implementation

Leased object references have been implemented as part of the AmbientTalk language[5]. The language has been implemented as an interpreter written on top of the Java Virtual Machine which runs on the J2ME platform. The mobile music player has been implemented and tested on HTCP3650 Touch Cruise smartphones connected by a WiFi network. In this section, we describe the necessary features of the implementation of leased object references to show how custom leasing patterns can be expressed.

Leased references have been built *reflectively* in the AmbientTalk language itself. They have been implemented as remote object references whose default semantics has been altered using the language's meta-object protocol (MOP) [8]. The implementation of leased references basically applies two changes to this semantics. First, the lifetime of a remote reference is limited by means of a timer which is initialized when the remote reference is created and associated to a service object. Second, any asynchronous message received in a leased reference is managed as shown in figure 3.

5.1 Leased Object References

A leased object reference is a unidirectional communication link from a client to a service object as depicted in figure 4 with a dotted line. At the implementation level, as also shown in figure 4, a leased reference actually consists of an ensemble of object references and two set of objects at client and server side: a `lease` object which implements methods for managing the life cycle of a leased reference, and an `interceptor` object which intercepts the messages sent to the server object and exposes the different variation points of a leased reference.

The Lease Object. A lease object contains a timer and implements methods to handle the life cycle of a leased object reference. Figure 5 (on the left-hand side) shows the API of the `lease` object. The `expire` method terminates the remote

[5] The language is available at `http://prog.vub.ac.be/amop/at/download`

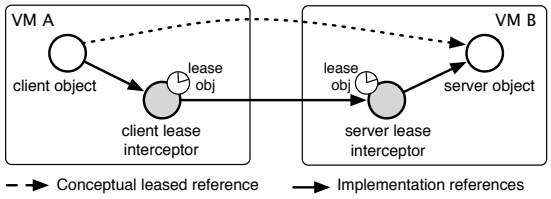

Fig. 4. Implementation of a leased object reference

access to the service object by taking offline the leased reference. The `revoke` method is analogous to the expire method but it does not notify the expiration observers. The `renew` method prolongs the lease timer with a specified renewal time (by default set to the initial time). The given time is not directly added to the initial time interval but, a new interval is calculated by taking the maximum of the current time left and specified the renewal time. Finally, the `whenExpired` method registers a block of code to be applied upon lease expiration.

```
def lease := object: {
  def expire();
  def revoke();
  def renew(renewalTime);
  def whenExpired(block);
};
```

```
def interceptor := object: {
  def receive(msg, lease);
  def pass(lease);
  def resolve(lease);
};
```

Fig. 5. The lease object API (left) and the interceptor API (right)

The Interceptor API. Figure 5 (on the right-hand side) shows the API of the interceptor object. `receive` is called every time an object receives a message. By default, the server lease interceptor overrides `receive` to forward the messages to the service object to which a leased reference grants access as shown below.

```
def receive(msg, lease) {
  if: !(lease.isExpired) then: { forward(msg, lease.serviceObject) }
}
```

The `pass` and `resolve` methods are called when an object is marshalled and unmarshalled to another device, respectively. Interceptors override `pass` and `resolve` methods to modify the parameter-passing semantics of the service object referenced to ensure a *pass-by-lease* semantics. More concretely, these methods are overridden by interceptors to create the client and server side of a leased object reference when a service is parameter passed.

The client side of a leased reference behaves slightly different than its server counterpart. The key difference is that it does not grant access to a service object but to the server side of the leased reference pointing to the actual service object (as shown in figure 4). Messages intercepted by the client lease interceptor are forwarded to the server lease interceptor. The client side of a leased reference

also has a lease object (which maintains its own timer kept in synchronization with its server counterpart) and its own **when:expired:** observers (which allow clients to be notified upon the lease expiration reference without requiring communication with the server). Another difference is that the client side of a leased reference does not adhere to the described *pass-by-lease* semantics since they do not provide access to a service object but to the server side of a leased reference.

Integrated Leasing Patterns A leased object reference created by means of the **lease:** construct uses the default interceptors explained above. We now describe how such default interceptors has been extended to implement the single-call and renew-on-call variants explained in Section 3. The single-call-lease and renew-on-call interceptors override `pass` and `resolve` with their own specific strategies to ensure *pass-by-single-call-lease* and *pass-by-renew-on-call-lease* semantics, respectively. `receive` is overridden in single-call and renew-on-call interceptors to provide automatic renewal and revocation of the leased reference upon message reception, respectively as follows:

```
// Renew-on-call server interceptor
def receive(msg, lease) {
  if: !(lease.isExpired)  then: {
    lease.renew(lease.renewalTime);
  };
  super.receive(msg);
};
```

```
// Single-call server interceptor
def receive(msg, lease) {
  def result := super.receive(msg);
  if: !(lease.isExpired)  then: {
    lease.revoke();
  };
  result;
};
```

Both server lease interceptors delegate the forwarding of the message to the default interceptor by means of a super-send. In the case of a renew-on-call lease, it renews its timer before delegating the forwarding of the message. In the case of a single-call lease, it cancels its lease upon receiving a first message by calling the `revoke` method, which also sets the `isExpired` property to `true` so that future received messages are dropped.

Other Variations on Leased Object References Using the provided APIs developers can extend the default leasing semantics to encode custom leasing patterns. For example, in our running example, the session object is leased with a renew-on-call lease which needs to be explicitly revoked upon the `endExchange` message. We have implemented a custom interceptor and lease object to encode this behaviour. The `receive` method for the interceptor is shown below:

```
// server interceptor
def receive(msg, lease) {
  if: !(lease.isExpired) then: {
    if: (lease.isRevoke(msg)) then: { lease.revoke(); }
    else: { lease.renew(lease.renewalTime) };
  }
}
```

The lease object needs has been extended with the `isRevoke` method which checks if the received message should automatically revoke the lease (in the case of the running example: if it is an `endExchange` message).

6 Discussion

Now that the leased object references model and its instantiation in AmbientTalk have been properly described, we evaluate them in the light of the criteria for leasing in MANETs identified in Section 2.

Leasing an Intermittent Connection. Leased object references combine leasing with asynchronous communication into one coherent language abstraction that deals with both transient and permanent disconnections. A leased reference defines a connection which supports intermittent disconnections by default: client objects can send messages via a leased reference as long as it is not expired, independently of the state of the connection, because a leased reference buffers messages while disconnected.

Leasing Patterns. Useful leasing patterns have been made available in the form of dedicated leased object references, e.g. renew-on-call and single-call leases. These lease variants illustrate that managing the lifetime of leased references can be done implicitly by means of message passing reducing the programming effort for the developer. In addition, the lease object and interceptor API explained in the implementation section form an extensible framework in which developers can plug in their own custom leasing patterns.

Symmetric Expiration Handling. By means of the registration of dedicated listeners triggered upon the expiration of a leased object reference, both sides of the reference can gracefully deal with the termination of their logical connection and schedule the appropriate compensating actions.

We consider the mobile music player application to be an illustrative example which exhibits a set of key issues that are typical in collaborative ad hoc networking applications. Its implementation, shown in Section 4, demonstrates how developers can concisely define and manipulate leased references and how the language support eases the development of mobile applications that deal with both transient and permanent disconnections and properly reclaim their service objects. Thanks to the language constructs presented, its implementation counts merely 90 lines of code. We have implemented a music player application which exhibits similar semantics in Java RMI which counts no less than 462 lines [6].

	Memory management code	Concurrency control code	Failure handling code	Application code	Total lines of code
Java RMI	145 (31,38%)	148 (32,03%)	78 (16,88%)	91 (19,69%)	**462**
AmbientTalk	7 (7,77%)	7 (7,77%)	6 (6,66%)	70 (77,77%)	**90**

Table 2. Summary of the lines of code (and %) for the music player application

Table 2 summarizes the lines of code for both implementations according to four different concerns [7]: 1) memory management: includes the code to setup a

[6] Both implementations are available at `http://code.google.com/p/ambienttalk/downloads`

[7] The code for service discovery has not been taken into account in this comparison.

renew-on-call lease for the music session and reclaim the used data structures upon lease expiration, 2) concurrency control: includes the code to ensure the responsiveness of the application in the face of transient disconnections, 3) failure handling: includes the code to have time-based delivery policy guarantees on remote messages and 4) application-level code. While the application code has a similar magnitude in both implementations, the Java RMI implementation has required to manually deal with the following issues:

- First, programmers have to manually deal with the impact of the volatile connections phenomenon on remote references. Upon a disconnection, the thread executing a remote method call blocks (as Java RMI uses synchronous communications). This makes the application unresponsive to GUI events and to discovery events notifying new music player peers in the network. To solve this issue, two different threads need to be spawned: a *transmission* thread that sends the remote messages and a *callback* thread that handles the return values. These threads need to communicate with each other by means of *message* objects which wraps a remote call. Five message classes were encoded corresponding to the different remote messages shown in figure 1. In addition, the transmission thread must ensure that messages are buffered while the reference is disconnected. This is not required in AmbientTalk as leased references themselves abstract the connection state.
- Second, the lifetime of Java RMI leases is controlled by the `leaseValue` property which is applied to *all* remote objects in the entire VM. It is not possible to specify lease periods on a per-application, per-class or per-object basis to provide application-specific policies. This needs to be manually encoded with timers. This property is also associated with the socked connection timeout which controls when a remote message send fails. Thus, *every* remote message send has the same timeout. In the music player application, `uploadSong` and `openSession` messages have different delivery guarantees (specified by the **@Due** annotation in our approach). In order to have similar semantics in Java RMI, the application needs to implement its own timers and modify the transmission thread to take into account the message expiration.
- Third, in Java RMI a remote reference is considered in use as long as the client holds it. When the client stops using the reference, the runtime sends an "unreferenced" control message to the server side which may be then able to garbage collect the object (once it is no longer remotely nor locally referenced). To stop using a remote reference, clients have to explicitly make sure to clear local references to the remote reference. For example, in the music player application, code was added to stop the transmission and callback threads and to remove the session from the hashmap storing active opened sessions. If clients do not add this code, the remote object cannot be collected unless there is a disconnection exceeding the `leaseValue` timeout. This code is avoided with renew-on-call or single-call leases in AmbientTalk. Leases created by means of **lease:** construct need to be manually revoked. To revoke a lease, only one revoke message must be sent rather than having to wait for the system to detect when the remote reference is no longer used.

– Fourth, Java RMI does not provide a renew-on-call lease pattern as the one used in the music player application. Rather, the client-side runtime system renews the lease to a remote object implicitly once the `leaseValue` reaches half of its value. This leads to unnecessary network traffic since control messages are sent to keep a remote object alive which may not be used anymore. To implement a renew-on-call pattern in Java RMI, the system provides the `dgc` interface which is not extensible. The interface is not meant to be used by application programmers, thus the pattern needs to be implemented explicitly requiring repetitive renewal code at client and server-side.
– Finally, no notification is performed in Java RMI upon lease expiration. Client objects are notified of the expiration of a lease only if they issue a remote method call upon an expired lease (which throws an exception). At server side, only when *all* clients disconnect, the `unreferenced` method is called on a remote object. In the music player application, the registration of listeners with leases had to be explicitly encoded. In AmbientTalk, dedicated language constructs are provided for the registration of expiration listeners at both sides of the leased reference.

7 Conclusion and Future Work

This paper focuses on the use of leasing for dealing with partial failures in mobile ad hoc networks. We identify a number of criteria for a leasing model specially designed for MANETs. We require a leasing model that (1) takes into account the volatile connections phenomenon, (2) provides different leasing patterns to manage the lifetime of leases, and (3) allows both lease holder and grantor to react to and schedule clean-up actions upon lease expiration. We subsequently propose the *leased object reference* model which exhibits such criteria. The contributions of leased object references are threefold: we provide leasing as a special kind of remote reference which tolerates both transient and permanent disconnections, we design leased references as an extensible framework which integrates useful patterns, e.g. renew-on-call and single-call leases, and we provide means for allowing client and service objects to detect and react to a permanent failure. The applicability of the language constructs have been assessed by means of the development of a typical collaborative ad hoc networking application.

By making use of leased references, programmers can distinguish transient from permanent disconnections and react accordingly. Disconnections that exceed the lease period are considered permanent requiring the collaboration between two communicating parties to be for example restarted in the music player application. However, leasing can only provide an approximation of when a disconnection is permanent. The quality of the approximation depends on the accuracy of the selected lease period. Selecting a suitable lease period is not straightforward and it requires to consider the behaviour of mobile devices in the physical world and factors such as the frequency of disconnections and reconnections. Any leasing mechanism has to deal with this issue but it is exacerbated in MANETs due to the unpredictability of the mobility patterns of the end-users. In future

work, we would like to allow the system to choose an appropriate lease timeout for the developer based on previously observed mobility patterns. Questions also arise in a leasing system regarding when to renew a lease. This is in a high degree application-specific. Although renew-on-call and single-call leases alleviate this problem by providing automatic renewal or revocation of leases, we would like to explore the use of leasing patterns which dynamically adapt the lease period under certain circumstances [4], e.g. changes in the observed mobility pattern.

References

1. Agha, G. *Actors: a Model of Concurrent Computation in Distributed Systems.* MIT Press, 1986.
2. Aleksy, M., Korthaus, A., and Schader, M. Realizing the leasing concept in corba-based applications. In *Proc. of Symp. on Applied Comp.* (2005), pp. 706–712.
3. Eugster, P. T., Felber, P. A., Guerraoui, R., and Kermarrec, A.-M. The many faces of publish/subscribe. *ACM Comput. Surv. 35*, 2 (2003), 114–131.
4. Gonzalez Boix, E., Vallejos, J., Van Cutsem, T., Dedecker, J., and De Meuter, W. Context-aware leasing for mobile ad hoc networks. In *ECOOP Workshop on Object-Oriented Technology for AmI and Pervasive Comp.* (2007).
5. Gray, C., and Cheriton, D. Leases: an efficient fault-tolerant mechanism for distributed file cache consistency. In *SOSP '89: Proceedings of the twelfth ACM symposium on Operating systems principles* (NY, USA, 1989), pp. 202–210.
6. Jain, P., and Kircher, M. Leasing. In *Proceedings of the 7th Patterns Languages of Programs Conference (PLoP)* (2000).
7. Joseph, A. D., de Lespinasse, A. F., Tauber, J. A., Gifford, D. K., and Kaashoek, M. F. Rover: a toolkit for mobile information access. In *Proc. of the 15th ACM Symposium on Operating Systems Principles* (1995), pp. 156–171.
8. Kiczales, G., Rivieres, J. D., and Bobrow, D. G. *The Art of the Metaobject Protocol.* MIT Press, Cambridge, MA, USA, 1991.
9. Lowy, J. Managing the lifetime of remote .net objects with leasing and sponsorship. *MSDN Library* (December 2003).
10. McLean, S., Williams, K., and Naftel, J. *Microsoft .Net Remoting.* Microsoft Press, Redmond, WA, USA, 2002.
11. 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.
12. Tanenbaum, A. S., and Steen, M. V. *Distributed Systems: Principles and Paradigms.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.
13. 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 *Int. Conf. of the Chilean Comp. Science Society* (2007).
14. Waldo, J. The Jini Architecture for Network-centric Computing. *Commun. ACM 42*, 7 (1999), 76–82.
15. Waldo, J. Constructing ad hoc networks. In *IEEE Inter. Symposium on Network Computing and Applications (NCA)* (2001), p. 9.
16. Yonezawa, A., Briot, J.-P., and Shibayama, E. Object-oriented concurrent programming in ABCL/1. In *Proceedings on OOPSLA* (1986), pp. 258–268.