Language Support for Leasing in Mobile Ad hoc Networks

Elisa Gonzalez Boix Tom Van Cutsem * Jessie Dedecker Wolfgang De Meuter

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

Abstract

In mobile ad hoc networks, distributed programming is substantially complicated by the fact that nodes in the network only have intermittent connectivity and the lack of any centralized coordination facility. Because transient disconnections are so omnipresent in mobile networks, we assume a distributed object-oriented programming model in which remote object references abstract over network disconnections by default. However, this language design decision has repercussions on distributed memory management, as disconnected remote references can prevent an object from being reclaimed. To address this issue, we integrate memory management based on *leasing* directly into the remote object reference abstraction, leading to the concept of a *leased object reference*. We explore the language design issues, the integration with other language features and illustrate the applicability of the language construct by means of a concrete example.

Categories and Subject Descriptors D.3.3 [*Programming Languages*]: Language Constructs and Features; D.3.4 [*Programming Languages*]: Processors—Memory management (garbage collection)

General Terms Languages, Design, Experimentation

Keywords mobile ad hoc networks, distributed garbage collection, leasing, language design, remote object references

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*. Such networks have two discriminating properties, which clearly set them apart from traditional, fixed computer networks: applications are deployed on *mobile* devices which are connected by *wireless* communication links with a limited communication range [12]. Example mobile network applications range from modest, already commonplace applications like collaborative text-editors, to more futuristic pervasive and ubiquitous computing [23] scenarios. Mobile ad hoc networks exhibit a number of phenomena which are rare in their fixed counterparts [6]:

Volatile Connections. Mobile devices equipped with wireless media possess only a limited communication range such that two communicating devices may move out of earshot unannounced. The resulting disconnections are not always permanent: the two devices may meet again, requiring their connection to be re-established. Often, such *transient* disconnections should not affect an application, allowing both parties to continue their collaboration where they left off. Because transient disconnections are so omnipresent in mobile networks, a disconnection should no longer be treated as a "failure" by default.

Zero Infrastructure. Mobile networks are often unadministered: devices usually spontaneously join with and disjoin from the network due to their physical mobility. As a result, in contrast to stationary networks, it is more difficult to rely on server infrastructure (e.g. a name server to manage service discovery) which may not be available when some devices meet and set up spontaneously a so-called ad hoc network. A peer-to-peer collaboration model is often more appropriate due to the limited infrastructure and communication range.

Any application designed for mobile ad hoc networks has to deal with the above phenomena. Because the phenomena are universal, an appropriate computational model should be developed that eases distributed programming in a mobile network by taking these phenomena into account from the ground up. One important observation is that, to deal with volatile connections in an object-oriented model, remote object references should tolerate network disconnections: a disconnected remote reference is not a "dangling" reference, it may always become reconnected when the network connection is restored. This paper specifically focuses on the repercussions of this design decision on distributed memory management and subsequently introduces the concept of leasing [9] into remote object references to cope with them, leading to the concept of *leased object references*.

We describe an abstract model for leased object references in section 3. In previous work, we have described the distributed object-oriented programming language AmbientTalk, which is designed specifically for mobile networks [6]. We describe an instantiation of the leased object reference model in this language in section 5. We use this language to illustrate leased object references as it already provides remote object references that tolerate network disconnections.

2. Motivation

The main motivation for incorporating leasing into the programming language stems from the characteristics that set mobile ad hoc networks apart from traditional computer networks. Before discussing the repercussions of mobile networks on distributed memory management by means of a concrete scenario, we first intro-

* Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

duce some terminology and concepts from the area of distributed object-oriented computing.

We assume an object-oriented system where objects can be *exported* in the network either explicitly 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 *remote object references*.

2.1 Case Study: the Mobile Music Player

In order to illustrate the issues of distributed garbage collection in mobile ad hoc networks on a concrete example, we first present a small case study in the form of a music player running on mobile devices such as PDAs or cellular phones. Consider a mobile music player containing a library of songs. When two people using the music player enter one another's personal area network (delineated by e.g. the bluetooth communication range of their cellular phones), the music players exchange their music library's index (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 is high enough, the music player can e.g. warn the user that someone with a similar taste in music is nearby. This example, although relatively small, is a typical collaborative ad hoc networking application and illustrates some of the key properties of collaborations in mobile ad hoc networks:

Discovery The so-called *service objects* that represent the music player application have to discover one another in a peer-to-peer manner. Once they have discovered one another, they need to set up a *session* to exchange their music libraries.

Communication Once the service objects have established a session, they need to transmit their library index in the face of volatile connections. A transient network partition should not cause the exchange to fail immediately. Hence, remote object references between both music players should tolerate transient disconnections by default.

Failure handling If a network partition does persist, remote object references still referring to the session should be eventually cleared such that it becomes possible to reclaim the allocated session object together with other resources allocated during the session, such as e.g. the partially downloaded library index of the remote party.

An implementation of this example and a more thorough discussion on the distributed garbage collection issues is postponed until section 4.

2.2 Reclaiming Remote Objects in Mobile Networks

A distributed garbage collector has to make trade-offs between *soundness* and *completeness*: when a client machine containing references to a server's objects disconnects from the network, the server either has to keep the objects exported until the client reconnects (sacrificing completeness – the client may never reconnect, keeping the object from being reclaimed) or it eventually takes the object offline such that it can be reclaimed (sacrificing soundness – the client may reconnect and still refer to the object) [1]. These problems are exacerbated in mobile ad hoc networks because due to the characteristics outlined in the previous section, network failures arise much more frequently and are often only temporary.

Traditionally, remote object references do not abstract over temporary network failures: when a network failure occurs, the remote reference becomes unusable and is considered "dangling". This semantics has the advantage that the remote server object can be unexported if all remote object references that referred to it have become disconnected. For example, in a distributed garbage collector based on reference listing, the reference can be directly removed from the reference list upon disconnection so that the remote object can be eventually reclaimed [16]. Because of the frequent disconnections in mobile networks, remote object references that do tolerate network failures are much more suitable as they remain valid during a disconnection, decoupling client and server objects in time. The expected behaviour is that such *time-decoupled* remote references are reconnected once the network connection is reestablished. However, this means that a disconnected remote object reference can no longer be regarded as "dangling", i.e. the server object it points to should not be reclaimed prematurely.

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 [21]. A *lease* is granted by a lease grantor to a lease holder, and grants the holder access to a resource managed by the lease grantor for a limited period of time that is negotiated by grantor and holder when the access is first requested. The advantage of leasing is that the lease grantor remains in control of the resource by maintaining the right to reclaim the resource once all of its leases have expired. Because of the lease time associated with a lease, the lease holder knows when its access rights have expired meaning it can no longer access the resource. Leasing solves the tension between soundness and completeness by weakening the notion of soundness: expired leases essentially denote "dangling references", but they are now admitted as a valid state of the distributed system.

2.3 Language Support for Leasing

We observe that on the one hand, leases have already been integrated into programming languages for managing the lifetime of remote objects (e.g. in Java RMI [18] and .NET Remoting [13]). However, in these approaches leases are not used for managing disconnected remote references. Rather, they are used to reclaim unused connected references. These systems do not provide time-decoupled remote references that tolerate failures. On the other hand, distributed computing frameworks (e.g. Jini [20] and one.world [10]) do employ leases for managing resources in the face of network failures. However, these approaches do not provide time-decoupled references either, and do not integrate leasing in the underlying language. Our contribution lies in integrating leases as they are used in e.g. Jini into the programming language as is done in e.g. .NET Remoting to specifically allow remote object references to tolerate network failures. In other words, we combine leasing with time-decoupled remote object references and provide the resulting leased object references as a first-class language abstraction.

Our goal is to provide language support for leasing, such that low-level leasing management details can be abstracted away as much as possible. Therefore, the focus of this research is on what features of a leasing framework to expose to the programmer, and what features to integrate in the language. As is always the case with language abstractions, a trade-off must be made between a decrease in customisability for the programmer and an increase in conciseness and properties that can be enforced by the language. Integrating leases into the language as leased object references, rather than offering them as a general library abstraction has the following advantages:

- Tedious boilerplate code, such as explicit lease renewal code, is no longer required. Rather, lease renewal is done implicitly by means of other language constructs, such as message sending, as will be described later.

- The language can enforce desirable properties and semantic constraints. For example, leased object references have dedicated parameter-passing semantics; expired leased references are treated uniformly as disconnected remote references; a programmer can express the semantic constraint that an object should be "pass-by-lease", ensuring that only leased references can remotely refer to the object. All of these properties are also explained in more detail later.
- A dedicated syntax for specific concerns in a language improves conciseness and readability of code, making the code easier to understand.

3. Leased Object References

In this section, we describe our leased object reference model which integrates leasing into time-decoupled remote object references. Although the description of this model is quite abstract, we will instantiate it in a concrete programming language in section 5.

A leased object reference is a remote object reference that transparently grants access to a remote service object for a limited period of time. When a client first references a server object that is leased, 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. Figure 1 illustrates an allocated leased reference. Each side 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 server object is revoked, i.e. the leased reference expires. A server object can have explicit control of its leases' lifetime by renewing or revoking them explicitly before they expire. When no renewal is performed due to a network partition or merely in the absence of utilization, the leased reference expires once its lease time elapses. Once a leased reference expires, both client and server object know that the client access to the server object is terminated.

In order to abstract over the transient disconnections inherent to mobile ad hoc networks, a leased reference decouples the client object and the server object it refers to in time. This means that a client object can send a message to the server object even if the leased reference is disconnected at that time. Client objects can only send messages to server objects *asynchronously*: when a client object sends a message to the server object, the message is transparently buffered in the leased reference and the client does not wait for the message to be delivered. 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 server 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.

Clients objects can register a listener with a leased reference, which is notified asynchronously when the reference expires. This is especially useful for client objects since it allows them to clear other resources and to perform failure handling if necessary. Multiple leased object references with different lease times can refer to the same server object. Only when all of the leased object references to a server object have expired can the server object become a candidate for local garbage collection.

Interactions with a leased server object are always subject to leasing. When a lease to a server object is passed to a client, both sides of the leased reference keep track of the remaining lease time as depicted in figure 1. Hence, no communication with the server is required in order for a client to detect the expiration of a leased reference. However, having both a client-side and server-side timer introduces issues of clock synchronisation. Keeping clocks synchronised is a well known problem in distributed systems [19]. 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. At worst, the asynchrony causes the server to revoke the lease too early. To the client, it will appear as if the remote object has disconnected due to a network failure and the lease will expire soon. Once the leased reference is established between the two parties, the server periodically sends the current remaining time by piggybacking it onto application-level messages.

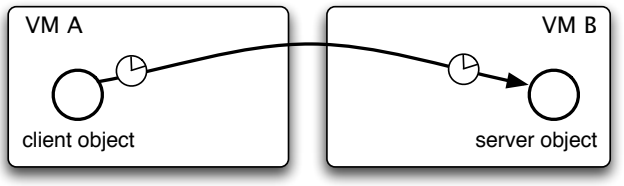

Figure 1. A leased reference

A client renewal request is delegated to the server-side which ultimately decides the actual renewal time since it is responsible for the server object's lifetime. If the renewal time has not been returned to the client side within a certain period of time, the client-side timer remains untouched and expires when the original time interval elapses. A client revocation request results in the cancelation of both the client and the server timers. If the client cannot communicate the revocation to the server, the server-side timer will eventually expire, causing the lease to be revoked anyhow.

As is the case in other leasing mechanisms, determining the proper lease renewal period is not straightforward and may even depend on system parameters such as the number of clients. We describe two variants on leased object references which transparently adapt their lease time 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 server object. They are based on similar leases in the .NET Remoting framework [13]. As long as the client uses the server object, the leased reference is transparently renewed by the interpreter. The default call time renewal is the initial time period of the leased reference. The second variant is a *single call* leased object reference which automatically revokes the lease upon performing a successful method call on the server object. Such leases are useful for objects which adhere to a *single call* pattern such as callbacks. For example, in asynchronous message passing schemes, callback objects are often passed along with a message in order for server object to be able to return values. Such callback objects are typically remotely accessed only once by server objects with the computed return value.

4. Mobile Music Player Implementation

In the previous section, we have introduced a model for integrating leasing into time-decoupled remote object references. Before describing an instantiation of this model in the AmbientTalk language, we discuss the distributed garbage collection issues in more detail by means of a concrete implementation of the mobile music player described in section 2.1. We first briefly introduce AmbientTalk in the following section.

4.1 The AmbientTalk Language

AmbientTalk is an object-oriented distributed language. The following code excerpt shows 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");
s.play();
```

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

In AmbientTalk, concurrency is not spawned by means of threads but rather by means of actors [2]. AmbientTalk actors are based on the *communicating event loops* model of the E programming language [14]. Each actor *owns* a set of regular objects. Objects owned by the same actor communicate using sequential message sending, as in Java or Smalltalk. Objects owned by different actors can only send asynchronous messages to one another. AmbientTalk borrows from the E language the syntactic distinction between sequential message sends (expressed as `o.m()`) and asynchronous message sends (expressed as `o<-m()`).

In order to make some objects available to remote actors and their objects, an actor can explicitly *export* objects that represent certain services. In AmbientTalk, a service object is always exported together with a *service type*. Service types serve as general descriptors merely used to categorise which objects export what kinds of services. In the music player example, each music player is modelled as an actor which exports an `interface` object that can be used by other music players to start a communication session to exchange libraries. This object is exported with the service type `MusicPlayer`, as follows:

```
deftype MusicPlayer;
def interface := object: {
  def openSession(sessionCallback) {
    // return a session object (explained later)
  };
};
export: interface as: MusicPlayer;
```

From the moment an object is exported by its actor, it is discoverable by other actors by means of its service type. One can define event handlers that are triggered whenever a remote object of a certain service type has become available in the network. For example, when a music player actor discovers another one in the local ad hoc network, it opens a session to exchange its music library index:

```
whenever: MusicPlayer discovered: { |remotePlayer|
  system.println("discovered new music player");
  // open a session to remotePlayer
}
```

The `remotePlayer` parameter of the event handler is a remote object reference to the exported `interface` object of another music player. The event handler can use the remote reference to asynchronously communicate with the discovered object. AmbientTalk's remote object references are time-decoupled. A remote reference may either be connected to or disconnected from the remote object. While disconnected, asynchronous messages sent via the reference are buffered and transmitted when the reference reconnects. This enables client objects to abstract over transient network failures.

Note that the code that exports the `interface` object, and the code above that discovers other such objects is executed by all music player actors in the network. Hence, music players engage in peer-to-peer communication: when a music player `A` and a music player `B` enter one another's communication range, `A` will discover `B`'s interface object and `B` will discover `A`'s interface object.

4.2 The Mobile Music Player

In this section, we describe the implementation of the library exchange protocol between two music players. The implementation described here, however, will not deal with distributed memory management yet. At the end of this section, we will describe the garbage collection issues that the example application exhibits. Language support for dealing with these issues in AmbientTalk is then introduced in section 5.

In the music player example, once one music player has a reference to the `interface` object of another music player, it can ask the remote player to open a library exchange session by sending it the `openSession` message. The `interface` object implements this message as follows:

```
def openSession(sessionCallback) {
  // store sender's music library in a set
  def senderLib := Set.new();
  def session := object: {
    def downloadSong(artist, title, ackCallback) {
      senderLib.add(Song.new(artist, title));
      ackCallback<-acknowledge();
    };
    def endExchange() {
      // calculate match percentage with my library
      def matchRatio := calcMatchRatio(senderLib);
      if: (matchRatio >= THRESHOLD) then: {
        // notify user of match
      };
    };
  };
  // return the session object
  sessionCallback<-receive(session);
};
```

The `openSession` method asynchronously returns a new session object which implements two methods which are used by a remote music player to send song information (`downloadSong`) and to signal the end of the library exchange (`endExchange`). A music player sends all of its own songs one by one to this session object after it has discovered a music player:

```
whenever: MusicPlayer discovered: { |remotePlayer|
  system.println("discovered new music player");
  remotePlayer<-openSession(object: {
    def receive(session) {
      // iterate over own music library
      def iterator := myLib.iterator();
      // auxiliary function to send each song
      def sendNextSong() {
        if: (iterator.hasNext()) then: {
          def song := iterator.next();
          session<-downloadSong(song.artist,song.title,
            object: {
              def acknowledge() {
                // recursive call to send next song info
                sendNextSong();
              }
            });
        } else: {
          session<-endExchange();
        };
      };
      sendNextSong();
    }
  });
};
```

The argument to the `openSession` message send is an anonymous callback object which is designated to receive the remote music player's `session` object, as shown previously in the code for `openSession`. 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 `downloadSong` message is only sent after the previous one returned an acknowledgement. Each acknowledgement is handled by an anonymous callback object.

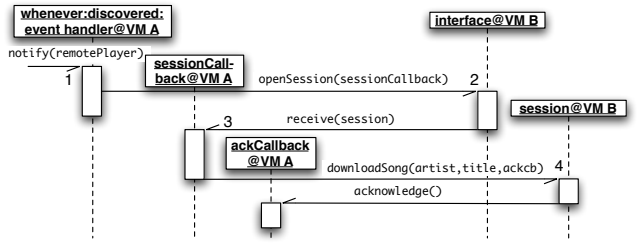

Figure 2. The library exchange protocol

Figure 2 gives a graphical overview of the library exchange protocol. Note that for purposes of clarity the figure only describes the protocol from the point of view of the music player on virtual machine `A`. In reality, this protocol is executed simultaneously on both virtual machines. The numbers correspond to the acquisition of a new remote reference to an object. They are discussed in the following section.

4.3 Distributed Garbage Collection Issues

Four different kinds of remote object references, and hence four different kinds of remotely accessible objects, are involved in the example application introduced above:

- The `remotePlayer` variable contains a remote reference to the explicitly exported interface object of the other music player.
- The `session` variable in the previous code excerpt contains a remote reference to the session object created by the remote music player.
- The `sessionCallback` variable in the method `openSession` contains a remote reference to the anonymous callback object that is used to asynchronously receive the session object.
- The `ackCallback` variable in the method `downloadSong` contains a remote reference to the anonymous callback object used to process the acknowledgement.

Figure 3 gives a graphical overview of each of these remote references. The numbers correspond to those in figure 2 and indicate at what stage of the protocol the remote reference is created.

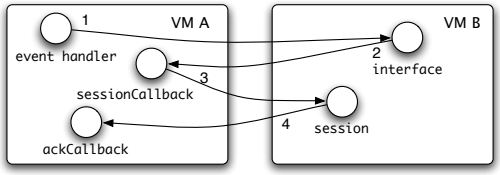

Figure 3. The exchanged remote object references

For each of these remote references and the objects they point to, we should consider whether and when they can be correctly reclaimed. First, the interface object of each music player has been explicitly exported. Note that from the moment such objects have been exported, they will not become subject to garbage collection unless explicitly unexported. Hence, no special memory management provisions must be taken for the `remotePlayer` reference. The `session` object, on the other hand, 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 (such as the `senderLib` variable to store incoming songs) should become garbage. This indicates that the remote reference to this session object should be *leased*, such that both music players can gracefully terminate the exchange process if the lease expires.

Because the anonymous objects serving as callback objects are parameter-passed in the `openSession` and `downloadSong` messages, they too are implicitly exported, such that the remote music player may refer to them. This means that they introduce additional memory management concerns. The anonymous object referred to by the `sessionCallback` variable should not be put online indefinitely. Rather, the callback only serves to process the incoming reply. If this reply does not arrive after some period of time, it makes sense to take the object offline and further ignore the music player, as the library exchange has not even started. Hence, the `sessionCallback` object should either be taken offline when the reply comes in or after a certain timeout period.

For the `ackCallback` object the situation is only slightly different. This object should also only be exported for a limited period of time. The callback can be taken offline and discarded when either the acknowledgement arrives or the session times out. The difference with the `sessionCallback` is that the timeout period to use can be derived from the session timeout, rather than picking an arbitrary timeout value.

The example code shown previously does not take these issues into account at all: when two music players disconnect, the objects referred to by the remote music player will remain exported and hence kept alive by the pointers in the actor's object table. This behaviour is due to the design decision of making remote object references resilient to network failures by default. In the next section, we describe the required changes to this default behaviour and apply the concept of leased object references to the example in order to take the above distributed garbage collection issues into account.

5. Language support for leasing

We now describe a concrete instantiation of the leased object reference model introduced in section 3. We have added support for leased object references to the AmbientTalk language. We describe AmbientTalk's leased object references by applying them to the case study to solve the garbage collection issues identified in the previous section. A description of how leased object references have been implemented in AmbientTalk is postponed until the next section.

Our language support features three different language constructs for creating leased object references which correspond to basic leased object references and the two variations described in section 3. The most basic form of a leased reference is created by the **lease** construct. As shown below, the construct requires two parameters: an object corresponding to the server object to which the leased reference grants access, and an initial time period.

```
lease: timeout for: object
```

The leased reference created with this **lease** construct only lasts for the given time interval unless a renewal or revocation is explicitly issued. The dedicated language constructs for explicitly renewing or revoking a leased reference are detailed later.

In order to create renew-on-call and single-call leases, our language support also provides the **renewOnCallLease** and **singleCallLease** constructs, respectively. The **renewOnCallLease** construct creates a leased reference which is automatically prolonged on every message invocation on the server object. In the mobile music player previously introduced, a renew-on-call lease can be used for the `session` object that represents the exchange process between two music players as follows:

```
def session := renewOnCallLease: minutes(10) for: (
  object: {
    def downloadSong(artist, title, ackCallback) {
      /* as before */
    };
    def endExchange() {
      revoke: session;
      /* as before */
    };
  };
});
```

As explained in section 4, once a music player establishes a reference to another music player, it can ask the remote player to open a library exchange session by sending it the `openSession` message which returns a session object. The `session` object should be subject to leasing in order for both music players to properly terminate the exchange process in the presence of network failures. The session object is exported using a lease for 10 minutes that 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. Since the anonymous 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 downloaded library of songs can be reclaimed as well.

By default, the renewal time applied on every call is the initial interval of time specified at creation. However, developers can also determine their renewal time by means of an extended version of the construct which takes as parameter a given renewal time.

The **singleCallLease** construct allows developers to create leased references that remain valid for only a single call. In other words, the leased reference expires after the server 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, an `ackCallback` object is parameter-passed in the `downloadSong` message in order for a `session` object to acknowledge that a song has been properly downloaded. A single-call lease can be used for unexporting this callback object upon receipt of the `acknowledge` message as follows:

```
session<-downloadSong(song.artist, song.title,
  singleCallLease: (leaseTimeLeft: session) for: (
    object: {
      def acknowledge() { /* as before */ }
    }));
```

Since the callback is only useful in the context of the current library exchange session, it only makes sense to export the callback for the remaining duration of the session (which can be acquired from a leased object reference by means of the `leaseTimeLeft` construct). The callback can become candidate for garbage collection either once it has processed its `acknowledge` method or when the session expires. If the callback's lease expires, the library exchange is stopped without requiring additional cleanup code.

Similar to the acknowledgement callback, the `sessionCallback` is parameter-passed in the `openSession` message to asynchronously receive a `session` object. The session callback object can also be exported with a single call lease as follows:

```
whenever: MusicPlayer discovered: { |remotePlayer|
  system.println("discovered new music player");
  remotePlayer<-openSession(
    singleCallLease: minutes(10) for: ( object: {
      def receive(session) { /* as before */ }
    }));
};
```

A lease time of 10 minutes is specified to wait for the reply. If a disconnection would occur after the `openSession` message was transmitted 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 library exchange protocol terminates before it was actually started, again requiring no additional cleanup code. By default, any message received revokes a single-call leased reference. However, an extended version of the **singleCall** construct exists that allows developers to specify explicitly which messages cause an immediate revocation.

Explicit manipulation of the lifetime of a leased reference is provided by means of the **renew** and **revoke** language constructs. The **renew** construct requests a prolongation of the specified lease reference with a new interval of time which can be different than the initial time. When a lease is renewed, the specified renewal time is not directly added to the initial time interval. Rather, the renewal time is used to determine a new expiration time by taking the maximum of the current time left in the leased reference and the renewal time specified. The construct looks as follows:

```
renew: aLeasedRef for: period
```

The **revoke** construct cancels the given leased reference. Canceling 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. The **revoke** construct has been already introduced in the code example for the `session` object where it is used to revoke the session's lease when a client signals the end of the library exchange.

In order to allow client objects to properly react to expired leased references, the **when-expired** construct is provided. The code below shows how a music player can detect when a session with a remote music player expires.

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

The construct takes as parameters a leased reference and a block of code that is executed upon the expiration of the lease. When the leased reference expires, all of the registered **when-expired** event handlers are notified and their associated block of code is triggered. These event handlers are only notified when the lease has expired, not when it has been revoked explicitly.

The case study of the mobile music player application illustrates how developers can concisely define and manipulate leased references. This case study features a simple yet representative example how language support for leasing eases the development of mobile ad hoc applications that properly reclaim their server objects. Without the language constructs presented, coding this application would have required to manually deal with concerns such as the renewal of the session leased reference or the revocation of the lease when a reply is received in the callback objects.

6. Implementation

Leased object references have been implemented as part of the AmbientTalk language[1]. The language has been implemented as an interpreter written on top of the Java Virtual Machine. It runs on the J2ME platform such that the language can be used in actual mobile networks. The mobile music player used as a case study in this paper has been implemented and tested on QTek 9090 smartphones connected by a WiFi network.

AmbientTalk does not have a built-in distributed garbage collector. Instead, a minimal set of low-level primitives for distributed memory management have been implemented in its kernel in Java. Rather than defining one specific garbage collector directly in the kernel, we have opted to enable the experimentation of distributed garbage collection techniques from within AmbientTalk itself. To this end, leased references have been implemented *reflectively* on top of the low-level support of the kernel. Before explaining the reflective implementation of leased references, we first describe the interpreter support for distributed memory management.

6.1 Remote Object References

The AmbientTalk data structures for distributed memory management have been based on reference listing [17] and the network objects framework [5]. Similar to these techniques, remote object references are implemented by means of a proxy at client-side, which encapsulates a *wire representation* of the exported object and whose methods are stubs that transform local message sends into distributed message sends. Messages sent to the server include method invocation information, as well as the wire representation of the receiver. Each actor maintains an *object table* which maps wire representations onto local references to exported objects. This table is used to transform distributed message sends back into local message sends.

As explained in section 4.1, AmbientTalk's remote references are by default resilient to disconnections. At a distributed garbage collection level, this implies that an exported server object remains part of the root set in the presence of disconnections since remote references remain valid and still refer to the server object while disconnected. We have implemented kernel support in order to make server objects subject to garbage collection in spite of such time-decoupled remote references. As the goal of our work is to uncover the necessary language abstractions for distributed garbage collection in mobile networks, the AmbientTalk kernel provides two low-level primitives named **takeOffline** and **when-takenOffline** to manipulate the object table.

The **takeOffline** primitive takes as parameters an object which is removed from the export table of the actor where the code is executed. When the object is removed from the export table, it no longer belongs to the set of root objects and as such, it can be eventually reclaimed by Java's local garbage collector once it is no longer locally referenced. Although the actual reclamation of an unexported object may be triggered at a later point in time, any attempt to access it via a remote reference will result in an `ObjectOffline` exception. At the client side, remote references to an object taken offline behave as *permanently* disconnected references. Despite having network connection, messages sent to the remote references are thus not forwarded to the server object. We further illustrate the usage of the **takeOffline** primitive in the implementation of leased references described in the next section.

AmbientTalk's asynchronous message sending semantics ensure that a message will be eventually received, but it does not ensure that the message is actually executed by the receiver of the message and returns a result. For AmbientTalk developers, a remote reference to an object taken offline behaves as a disconnected reference. However, at the meta-level, developers can distinguish between a disconnection engendered by a network partition and one engendered by a remote object that was taken offline. The **when-takenOffline** primitive allows developers to install event handlers on a remote reference that are notified when the object pointed to is taken offline.

The **takeOffline** primitive can be considered the equivalent to the so-called `delete` operation provided by some sequential languages without built-in local garbage collection. As previously mentioned, we have opted for a minimal distributed memory management support in the interpreter so that leased object references can be reflectively built in the language. Regular AmbientTalk developers should make use of higher-level abstractions, rather than using the **takeOffline** primitive.

6.2 Leased Object References

As explained in section 3, a leased reference conceptually is a unidirectional communication link from a client to a server object as depicted in figure 4 with a dotted line. At the implementation level, a leased reference actually consists of an ensemble of object references as also shown in figure 4. More specifically, a leased reference is composed of a proxy object in the client host, denoted as the *client lease proxy (CLP)*, referring to a lease object which grants the actual access to the server object, denoted as the *server lease proxy (SLP)*. Note that both client and server lease proxies are transparent to the regular AmbientTalk developer.

Figure 4 illustrates the internal details of the remote reference between CLP and SLP according to the implementation of remote references explained in section 6.1. On the client side, the CLP encapsulates the wire representation of the server object denoted by the `SLW` key. On the side of the server object, the object table maps the wire representation `SLW` to the SLP.

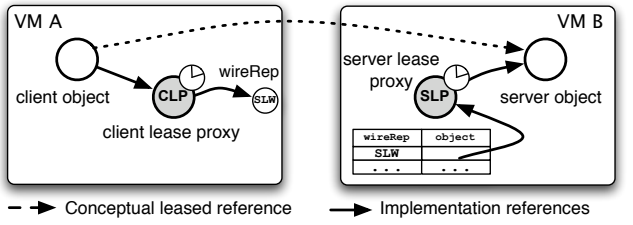

Figure 4. Implementation of a leased object reference

A CLP is responsible for transparently intercepting client messages sent to the server object and for forwarding them to the SLP. In addition, it also manages the client **when-expired** subscriptions. As also shown in figure 4, the client lease maintains its own timer which is kept in synchronization with the SLP's timer. When the timer expires, the CLP cleans the reference to the wire representation of SLP.

On the other side, the SLP intercepts the messages received from the client lease and forwards them to the actual server object. The SLP also maintains a timer and removes its entry in the object table once its time elapses. The expiration of the server timer terminates the actual access to a server object. This has been implemented by means of the **takeOffline** primitive as follows:

```
timer.schedule(timeout,
  object: {
    def run() {
      notifyExpiredEventHandlers();
      expired := true;
      takeOffline: slp;
    }
  }
);
```

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

The above excerpt of code schedules a new task object passed as parameter which is triggered when `timeout` elapses. More precisely, the `run` method of the task object is executed after the specified time interval. The `run` method is responsible for notifying the **when-expired** event handlers and unexporting the server lease proxy `slp`. At the implementation level, a server object itself is not exported to clients. Rather, its server lease proxies are the ones remotely accessible on behalf of the server object. All interactions with a server object are ensured to be subject to leasing and as such, the server object can be eventually reclaimed. Since each time a client establishes a remote reference to a server object, a new leased reference will be created, once all leased references expire, the server object is no longer referenced from any SLP and hence, from the object table.

Server lease proxies respond to a set of methods to manipulate the lease life time, i.e. to get the time left, renew and revoke a lease, and install the **when-expired** observers. Note that **when-expired** observers can also be installed on server lease proxies so that server objects are able to clean other resources created during the interaction with a client object once its leased reference expires.

Leased reference proxies obey a *pass-by-lease* semantics to ensure that interactions with a leased server object is always subject to leasing. When a SLP to a server object is passed to a client, the server lease is wrapped in a placeholder object which upon arrival at the client side creates the CLP referring to the SLP. In order to properly implement such expected behaviour, a SLP overwrites certain methods forming part of the metaobject protocol of AmbientTalk, namely **receive** and **pass** meta methods which reify the receipt of messages and the serialization of objects, respectively.

The **receive** method is called at the meta level each time an object receives an asynchronous message. The SLP created by means of the **lease** construct, replaces the default AmbientTalk semantics by the following code so as to forward the messages buffered to the referenced server object provided that the lease has not already expired.

```
def receive(msg) {
  if: !(expired) then: {
   forward(msg, serverObject)
  }
}
```

The **pass** meta method is called when a SLP is serialized when handed over to a remote client object. On the client side, the **resolve** meta method is called when an object is received in order to properly deserialize it. Server lease proxies override the **pass** meta method to apply *pass-by-lease* semantics instead of the default pass-by-reference semantics applied to regular AmbientTalk objects. When a lease server object is handed over to the client object, **pass** returns a pass-by-copy wrapper object whose only behaviour is to overwrite the **resolve** meta method that is executed upon receipt at the client virtual machine. When **resolve** is called at the client side, such wrapper object creates a CLP that points to the actual SLP residing at the server side.

As explained in section 3, our leasing model also provides single-call and renew-on-call leased references. At the implementation level, the common behaviour to all kinds of leased references is represented by a core leased reference which corresponds to those created by means of the **lease** construct. The core leased reference is implemented by the SLP and CLP described above. Single-call and renew-on-call server and client lease proxies are implemented as an extension to the core server and client lease proxies, respectively. More concretely, such proxies overwrride the receipt of messages and the serialization of objects inherited from the *core* objects with specific strategies.

In the case of a renew-on-call leased reference, its SLP renews its timer before delegating the forwarding of the message to the core object. The renew-on-call lease object overrides **receive** as follows:

```
def receive(msg) {
  self.renew(renewalTime);
  super.receive(msg);
};
```

As we mentioned in section 5, the renewal time is by default the initial lease timeout. Similar to renew-on-call leases, a single-call SLP delegates the forwarding of the message to the core object. As shown in the code below, the single-call lease then revokes the leased reference either if the selector of the message corresponds to a message name contained in the `forSelectors` array or if the array is empty which means that any message selector revokes the lease.

```
def receive(msg) {
  def result := super.receive(msg);
  if: !(expired) then: {
     if: (forSelectors.isEmpty().or: {
        forSelectors.contains(msg.selector)}) then: {
       self.revoke();
     }
  };
  result;
};
```

Renew-on-call and single-call server leases override the **pass** meta method as well in order to create the CLP that use their equivalent strategies at the client side. Client lease proxies behave slightly different than their server counterparts. The key difference is that client lease proxies do not adhere to the pass-by-lease semantics since they are not actually granting access to the server object but to the SLP.

Finally, we detail the interaction between client and server proxies in terms of renewal and revocations originated by the client object. Intercepted client renewals or revocations on the CLP are delegated to the server lease reference. If a client revokes the leased reference, this does not pose much challenges: the client lease first cancels its timer and then flushes the request to the server lease. If the request is lost, the server lease will in any case eventually be collected when the original time interval elapses. However, client renewal requests only prolong the timer of the CLP upon successful renewal acknowledgment from the SLP.

Due to space constraints, a comprehensive explanation of the reflective implementation of leased references cannot be covered in this paper. However, the complete reflective source code can be found in the system library shipped with AmbientTalk.

7. Discussion and Future work: Variations on Leasing

As explained in the introduction, our goal is to incorporate leasing into the programming language. In section 5 we have illustrated three concrete instances of such language support. However, these particular language constructs do not always suit the need of the developer. Other variations on the semantics of leased object references are definitely possible. Because leased references have been implemented reflectively, defining variations on the described language constructs is relatively easy, as it can be done in the high-level language itself. Our future work consists of **a)** investigating other language constructs for leasing, **b)** finding novel, useful renewal strategies of leased references, and **c)** integrating leased references with other language constructs. We give a concrete example of each of these research directions below.

The language abstractions for leased references described in this paper are applied on a *per-object* basis. However, identifying each remote reference and applying language constructs separately still places considerable burden on developers. To alleviate this, we are currently exploring new language constructs to allow developers to delimit a scope of action so that all objects exported within this scope are transparently exported as leased references.

In the spirit of renew-on-call leases, we are investigating leased references with built-in renewal policies where the renewal of the lease depends on changing context parameters of the system. For instance, a leased reference can be created that remains valid only while the battery level of the device hosting the server object is above an acceptable limit. This is particularly relevant to the development of context-driven adaptations in mobile ad hoc networks. We believe that changes in context not only require adaptation in the behaviour of the application but also permeate to distributed memory management. In future work, we will thus explore self-adaptive leasing techniques [7] to dynamically adapt the lease time and the renewal frequency according to context parameters.

As an example of integrating leases with other language constructs, we are currently exploring how to integrate leased object references with the concept of *futures* [3] or *promises* [11]. Futures are a recurring language abstraction in concurrent languages with asynchronous message sends (e.g. ABCL [24], E [14], Argus [11]). A future is a placeholder for the return value of an asynchronous message send. It allows the sender of an asynchronous message to access the return value of that message at a later point in time. The receiver either *resolves* the future with a return value or *ruins* the future with an exception. In AmbientTalk, futures have been added to the language reflectively, just like leased object references.

In our case study application, we have used callback objects to circumvent the lack of return values in native asynchronous message sends. With the introduction of futures, explicit callbacks are no longer necessary: the future serves as an implicit callback. We integrate futures with leasing by exporting a future attached to an asynchronous message using a **singleCallLease** which either expires due to a timeout or upon the reception of a `resolve` or `ruin` message. When the future's lease expires due to a timeout, the future is automatically ruined with a `TimeoutException`. For example, the asynchronous invocation of `openSession` in section 5 can be rewritten using futures as:

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

The timeout for the implicit **renewOnCall** lease on the future can be set by annotating the asynchronous message (the `@Timeout` annotation). AmbientTalk inherits from E the notion of *non-blocking* futures [14]. It is impossible for code to block on a future until its value is known. Rather, it is possible to register an event handler with a future that is asynchronously triggered when the future is resolved or ruined (by means of the **when-becomes-catch** construct). This example shows how more low-level memory management concerns can be cleanly incorporated into more high-level abstractions, decreasing the mental overhead for the developer.

8. Related Work

Distributed garbage collection has been thoroughly investigated for traditional distributed systems [1, 16]. Most contemporary distributed garbage collection (DGC) algorithms stem from one of the two well-known families derived from centralized systems, namely *tracing* and *reference counting*. DGC algorithms based on tracing often require a substantial amount of cooperation among the nodes in the network. Because of the lack of reliable infrastructure and administration in a mobile network, tracing algorithms are impractical in this context. DGC algorithms derived from reference counting [15, 8, 4, 22] or reference listing [5, 17], on the other hand, offer more scalable solutions but they are not directly applicable in mobile ad hoc networks because they do not abstract over temporary network failures: remote references break upon a network disconnection.

We have already argued that leasing provides a robust mechanism to manage reclamation of remote objects in mobile ad hoc networks. Leases were originally introduced as a fault-tolerant approach in the context of distributed file cache consistency [9]. In the context of distributed computing platforms for ad hoc networks, Jini [21, 20] was built from the ground up with the notion of leasing. In Jini, leases are offered as a general programming abstraction which can be used 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. In distributed object-oriented frameworks like Java RMI [18] and .NET Remoting [13] on the other hand, leases are tightly coupled to the garbage collector and are used to describe the lifetime of remote objects. However, non of these approaches provide time-decoupled references that tolerate transient failures inherent to mobile ad hoc networks. In the remainder of this section, we discuss these leasing approaches and compare them with the leased object references presented in this paper.

In Java RMI and .NET Remoting, leases were introduced as a way to manage the lifetime of remote objects. Similar to our approach, Java RMI has been built based on Birrell's network objects [5]. However, its leasing mechanism is less open to the developer since the lease time is controlled by the system by means of a `leaseValue` parameter set to 10 minutes by default. Moreover, language support is also more limited since all the interactions with DGC are entirely based on the so-called *dirty* and *clean* calls. However, such methods are intended to be used by the DGC algorithm. Unlike our approach, no dedicated language constructs are provided to developers for managing renewals of leases which can be only accomplished by making additional dirty calls on the remote references. Related to this, .NET Remoting framework incorporates the concept of *sponsorship* in its distributed memory management scheme. 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 the lifetime of server objects. In contrast to .NET Remoting, in our leasing model the life time of leased references is always ultimately decided by the server lease object which does not ask client objects for renewals. This was a deliberate design choice motivated by volatile connections and scarce infrastructure of mobile ad hoc networks. Since client and service provider devices may not encounter again after a disconnection, service provider devices are expected to have the final decision on the reclamation of their exported objects. The .NET Remoting framework, on the other hand, provides a `RenewOnCallTime` property to automatically extend the lease on every call which formed the basis for our renewal-on-call leases.

Jini provides direct support to deal with the fact that clients and service providers may join with and disjoin from the network at any time, without any prior warning. This phenomenon motivated the introduction of the leasing so as to allow client and services to leave the *Jini federation* easily without disrupting other members. However, as previously explained, their leasing mechanism is part not only of their distributed memory management model, but it is

also employed in other kinds of interactions. For instance, in order for the lookup service to deal with unheralded disconnections, services must explicitly renew their lease with the lookup service; if they cannot, the lookup service will remove the service advertisement such that it doesn't provide stale information. Likewise, clients should interact with services on the basis of a lease such that a service may reclaim any resources allocated for the client session whenever either one disjoins from the network. However, Jini is a set of conventions to allow services and clients to from a flexible distributed system built on top of Java RMI [20]. In other words, Jini extends the semantic model of RMI with guidelines to use leasing. This implies that although no conventions are placed on the implementation of remote references, the communication model of remote references is synchronous as in Java RMI. Instead of just providing rules or guidelines for leasing as Jini, in our approach the integration of leasing at a language level is a crucial design decision in order to enforce the semantics that leased references must exhibit to properly determine the reachability of the remote objects referred to.

9. Conclusion

This paper has described leased object references: time-decoupled remote object references with built-in leasing semantics. The language construct has been designed to overcome the difficulties of distributed programming in mobile ad hoc networks. We have described the benefits of time-decoupled remote object references in this context, but also how this design decision impacts distributed memory management.

Leased object references incorporate leasing directly into the remote object reference abstraction. Rather than providing a general leasing framework in which useful patterns can be expressed, we have chosen a language approach where the useful patterns are made available in the form of dedicated leased object references, e.g. renew-on-call and single-call leases. We have described the language construct's design and (reflective) implementation in the AmbientTalk language, and discussed other variations on the semantics of the leased object references presented in this paper. The applicability of the language constructs have been assessed by means of a small but representative case study.

References

[1] ABDULLAHI, S. E., AND RINGWOOD, G. A. Garbage collecting the internet: A survey of distributed garbage collection. In *ACM Computing Surveys* (1998), vol. 30, pp. 330–373.

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

[3] BAKER JR., H. G., AND HEWITT, C. The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages* (1977), vol. 8 of *ACM Sigplan Notices*, pp. 55–59.

[4] BEVAN, D. I. Distributed garbage collection using reference counting. In *Parallel Architectures and Languages Europe* (1987), Springer-Verlag, pp. 176–187.

[5] BIRELL, A., EVERS, D., NELSON, G., OWICKI, S., AND WOBBER, E. Distributed garbage collection for network objects. Tech. Rep. 116, Digital Equipment Corp. Research Center, 1993.

[6] DEDECKER, J., VAN CUTSEM, T., MOSTINCKX, S., D'HONDT, T., AND DE MEUTER, W. Ambient-oriented Programming in Ambienttalk. In *Proceedings of the 20th European Conference on Object-oriented Programming (ECOOP)* (2006), pp. 230–254.

[7] DUVVURI, V., SHENOY, P., AND TEWARI, R. Adaptive leases: A strong consistency mechanism for the world wide web. *IEEE Transactions on Knowledge and Data Engineering 15*, 5 (2003), 1266–1276.

[8] GOLDBERG, B. Generational reference counting: A reduced communication distributed storage reclamation scheme. In *Programming Languages Design and Implementation* (1989), vol. 24, ACM SIGPLAN, pp. 313–321.

[9] 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* (New York, NY, USA, 1989), ACM Press, pp. 202–210.

[10] GRIMM, R., ANDERSON, T., BERSHAD, B., AND WETHERALL, D. A system architecture for pervasive computing. In *EW 9: Proceedings of the 9th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2000), ACM Press, pp. 177–182.

[11] LISKOV, B., AND SHRIRA, L. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN conference on Programming Language design and Implementation* (1988), ACM Press, pp. 260–267.

[12] MASCOLO, C., CAPRA, L., AND EMMERICH, W. Mobile Computing Middleware. In *Advanced lectures on networking*. Springer-Verlag New York, Inc., 2002, pp. 20–58.

[13] MCLEAN, S., WILLIAMS, K., AND NAFTEL, J. *Microsoft .Net Remoting*. Microsoft Press, Redmond, WA, USA, 2002.

[14] MILLER, M., TRIBBLE, E. D., AND SHAPIRO, J. Concurrency among strangers: Programming in E as plan coordination. In *Symposium on Trustworthy Global Computing* (April 2005), R. D. Nicola and D. Sangiorgi, Eds., Springer, pp. 195–229.

[15] PIQUER, J. M. Indirect reference counting: A distributed garbage collection algorithm. In *Proceedings of the Conference on Parallel Architectures and Languages Europe* (1991), vol. 505 of *LNCS*, Springer-Verlag.

[16] PLAINFOSSÉ, D., AND SHAPIRO, M. A survey of distributed garbage collection techniques. In *Proceedings of International Workshop on Memory Management* (1995).

[17] SHAPIRO, DICKMAN, AND PLAINFOSSE. Robust distributed references and acyclic garbage collection. In *Proceedings of the 11th Symposium on Principles of Distributed Computing (PODC)* (August 1992).

[18] SUN MICROSYSTEMS. Java RMI specification, 1998. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[19] TANENBAUM, A. S., AND STEEN, M. V. *Distributed Systems: Principles and Paradigms*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[20] WALDO, J. The Jini Architecture for Network-centric Computing. *Commun. ACM 42*, 7 (1999), 76–82.

[21] WALDO, J. Constructing ad hoc networks. In *IEEE International Symposium on Network Computing and Applications (NCA'01)* (2001), p. 9.

[22] WATSON, P., AND I.WATSON. An efficient garbage collection scheme for parallel computer architecture. In *Parlallel Architectures and Languages Europe* (1987), Springer-Verlang, pp. 432–443.

[23] WEISER, M. The computer for the twenty-first century. *Scientific American* (september 1991), 94–100.

[24] YONEZAWA, A., BRIOT, J.-P., AND SHIBAYAMA, E. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications* (1986), ACM Press, pp. 258–268.