Distributed Proxies as Delegation-based Descendants

Tom Van Cutsem[*] Stijn Mostinckx[†] Wolfgang De Meuter
Jessie Dedecker[*] Theo D'Hondt
Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel
Pleinlaan 2 - 1050 Brussels - Belgium
{tvcutsem,smostinc,wdmeuter,jededeck,tjdhondt}@vub.ac.be

ABSTRACT
The contemporary object distribution spectrum consists of proxy-based middleware solutions on the one hand and distributed programming languages on the other hand. Middleware suffers from numerous technical problems due to its inability to hide distribution and current day languages fail precisely because they hide distribution too much. The paper presents a distributed programming language feature that occupies middle ground by treating proxies as delegation-based descendants in the prototype-based sense. Several distribution idioms are shown to be elegantly expressible using the proposed feature.

Categories and Subject Descriptors
D.3.2 [**Programming Languages**]: Language Classifications—*distributed languages, prototype-based languages*; D.3.3 [**Programming Languages**]: Language Constructs and Features—*delegation*

General Terms
Design, Languages

Keywords
Distributed objects, proxies, prototypes, delegation

1. INTRODUCTION
In the past decade, the notion of mobile computing has made its definitive entrance in society. Portable devices such as cellular phones, smartphones, laptops and PDA's are becoming ever cheaper, smaller and more powerful. Moreover, in the past few years, these devices have been equipped with fast and reliable wireless networking hardware. One does not have to be a big prophet to foresee a near future in which distributed software will run on dynamically defined networks, the nodes of which consist of portable devices. This new hardware constellation will almost certainly have its repercussions on the software side as well. Developing distributed applications that can deal with this kind of dynamicity will put ever higher demands on programming languages and library support concerning flexibility and adaptability.

The contemporary technique for building distributed applications consists of deploying a standard object-oriented programming language (such as Java, Smalltalk or C#) in conjunction with a suite of tools commonly referred to as *middleware* (such as CORBA [27], SOAP-RPC [40], J2EE and Java RMI [31]). Middleware typically consists of precompilers and libraries which programmers can use to enhance their programming language with distributed objects, name servers, remote message sending and other concepts related to distributed object computing (DOC). A characteristic common to state-of-the-art DOC middleware is its reliance on what is known as *remote object proxies*. A proxy (also called a *stub*) is a local stand-in which forwards messages to an object located in a remote object space. Typically, proxy code is synthesized by the middleware's tools and has the same interface as the object for which it is a stand-in. The idea is to allow the application code to send messages to a proxy as if it were sending them directly to the object it represents, because – from a modelling point of view – the object and its proxy designate the same entity. However, from a technical point of view, the object and the proxy are two idiosyncratic objects. In the paper we show that this discrepancy is a fundamental problem of the middleware approach because it heavily interacts with other object-oriented language features such as static typing, reflection and object identity.

An approach to DOC that radically differs from the middleware approach is to incorporate the issues related to remote objects in the programming language. Example languages are Argus [23], Emerald [3, 16], Obliq [7], dSelf [33] and distributed objects in BETA [4]. In the language-based approach, communicating parties send messages to local objects which technically *are* the same object as the remote objects they represent. In this approach, programmers cannot distinguish between an object and a proxy. Unfortunately, such full transparency has its problems too [37, 19, 13], as e.g. supported by Guerraoui and Fayad in their CACM column: *"being aware of the fundamental difference between distributed object invocation and local object invocation has the merit of giving up the myth of distribution transparency and the claim that an OO concurrent language is enough because distribution is only an im-*

[*]Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)
[†]Author funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen)

plementation detail" [13]. Full transparency precludes developers from exploiting their knowledge about a system in order to control some of its non-functional aspects such as performance, fault tolerance and quality of service. Moreover, in the context of the current paradigm shift from stationary networks towards mobile networks, full transparency gets even more problematic because the presence of distribution becomes even tangible at the application level [26].

This paper takes the stance that a new generation of programming languages is required, situated in the middle of the distributed object spectrum currently defined by the "handicraft" middleware and the fully transparent programming language extremes. In this context we present a novel distributed object model that conceives proxies as delegation-based descendants of the objects they represent. In other words, proxies are objects that transparently delegate (in the original prototype-based sense proposed by Lieberman [21]) messages over the network to the objects they represent. These ideas are embodied in an experimental distributed programming language called ChitChat. ChitChat thus exploits the parent-child delegation relationship between objects in a networked context. The paper discusses the technical consequences of this idea in great detail and presents a suite of programming idioms that elegantly show the expressivity of distributed proxies as delegation-based descendants.

2. MOTIVATION

At the heart of both DOC approaches outlined above is a *referencing* mechanism (consisting of explicit proxies or implicit distributed object references) that allows an object in one object space to refer to an object in another object space. The object being referred to is called the *subject* of the remote reference. Upon receiving a message, it is the task of a remote reference to forward that message to its subject after having established a network connection with the host of that subject. Hereby, actual arguments have to be *marshalled*, potential return values have to be *unmarshalled* and synchronisation between the sender and the receiver may have to be taken care of.

This section argues that both DOC approaches are problematic in the way they tackle these issues. The problems with explicit proxies of middleware solutions are discussed in section 2.1. The shortcomings of implicit references advocated by distributed programming languages are discussed in section 2.2.

2.1 Problems with Proxies

The realisation of proxies in most programming languages, even modern ones such as Java and C#, is a source of subtle problems. Although the bulk of application development is done in statically typed languages, we distinguish between problems that are related to static typing and problems that have to do with object identity. The latter are more fundamental and also manifest themselves in dynamically typed languages.

2.1.1 Type-related problems

In order to illustrate the problems associated to proxies in a statically typed language, consider part of a typical[1] implementation of complex numbers in Java:

```
public class Complex {
   private float re, im;
   public Complex(float re, float im) {
      this.re = re; this.im = im;
   };
   public boolean equals(Object o) {
      if (o instanceof Complex) {
         Complex c = (Complex) o;
         return (this.re == c.re)&&(this.im == c.im);
      } else
         return false;
   };
   public Complex add(Complex c) {
     return new Complex(this.re+c.re,this.im+c.im);
   } }
```

[1]The use of `instanceof` is considered bad object-oriented programming style, yet it is widely used in practice, especially in methods like `equals`.

Imagine an application that uses complex numbers on different network nodes. Assume that, for reasons of simplicity and object state consistency, objects are never copied or physically moved across the network. This means that, upon passing back and forth arguments and results associated to remote message sends, these objects will be represented by proxies instead of being moved or copied. If the `Complex` class is to be distributed under these assumptions, it will be paired with a `ComplexProxy` class that represents its remote instances. Whether this proxy class is automatically generated or not, depends on the specific DOC middleware at hand. In Java RMI, to make its methods remotely invocable, the `Complex` class needs to implement an interface `ComplexI` that extends `java.rmi.Remote`. Only the methods declared in the `ComplexI` interface will be available to remote peers. The Java RMI stub compiler will automatically create the proxy class `ComplexProxy`[2] which also implements `ComplexI`. It is important to see that the middleware will have to replace references to the original `Complex` type by references to `ComplexI` since both `Complex` and `ComplexProxy` are valid complex numbers. If this were not done, we would, for example, be unable to add complex numbers that reside in different object spaces.

Given this setup with both local and remote complex numbers, the code as specified above breaks in several ways. For example, one might send the `equals` message to a local complex number and pass along a proxy (i.e. a `ComplexProxy` object) as a parameter. The `instanceof` test then fails because the type of the formal parameter `o` is `ComplexProxy`, not `Complex`. The proxy scheme interferes with static typing unless all code involving `Complex` objects is re-typed with the `ComplexI` interface. This requires invasive changes to the application code. But these changes do not solve all problems as shown by the binary `add` method. Replacing the parameter type by the interface type makes the body of the method access instance variables of an interface type; an impossibility. The only way to solve this is to change the original class with public methods to 'get' all instance variables that are read in this way (which is essentially the technique behind C# properties). Needless to say, this is a total bypass of encapsulation which is particularly problematic in dynamically defined networks where potentially malicious nodes can join a network.

2.1.2 Identity Problems of Proxies

The fundamental problem in the above example is that a subject and its proxies conceptually denote the same object but technically are different objects. At the programming language level, both objects have their own object identity. Since the proxy's object identity does not coincide with the subject's identity, operators such

[2]In Java RMI, the generated class would actually be named `Complex_Stub`.

as pointer comparison (`==` in C++ and Java) and downcasts will produce different results depending on whether they are applied to the proxy or the subject. The problem *can* be circumvented by *strictly* adhering to message passing such that a proxy can forward all operations (e.g. only using `equals` instead of `==` in Java), but mainstream languages do not enforce this.

Another language feature that severely interferes with proxies is reflection. When a Java programmer explicitly names classes using strings (e.g. `Class.forName("Complex")`), there is no way for middleware or libraries to ensure that proxies are integrated consistently into the system. A bit less far-fetched, invoking `obj.class.getFields()` returns different results depending on whether `obj` is a proxy or not. The problem is that the proxy has no means of redirecting such accesses to the subject.

These issues with operators, identity and reflection are to be taken seriously. Object identity is important for nearly any non-trivial application, and many Java libraries and extensions employ Java's introspection API. Distributed applications, glued together using proxies which introduce ambiguous object identities, will be more difficult to understand, maintain and develop because the problems outlined above lead to subtle bugs and result in tangling functional code with proxy related code.

2.2 Problems with Transparent Distribution

An alternative to the middleware approach with explicit proxies consists of distributed programming languages that advocate full transparency of remote object references. Such languages defer all non-functional aspects of distribution to the programming language infrastructure. However, as already hinted at in the introduction, the paradigmatic shift from closed stationary networks towards open mobile networks renders full transparency ever less tenable. Indeed, the complexity of such open mobile networks can no longer be dealt with by a predefined language infrastructure [26]. Instead, a developer will have to enhance the distributed behaviour of his remote object references with extra knowledge. In the middleware approach, such enhanced remote objects are known as *smart proxies* [25]. Typically, they are introduced for the following reasons:

Caching Smart proxies can cache return values of remote operations when those values change infrequently or not at all. This can reduce network traffic substantially.

Failure Handling A smart proxy can specify meaningful behaviour to handle a failure of the network connection or the device hosting the subject. The proxy can e.g. batch the messages until the connection can be restored or discard the broken connection and replace as much functionality of the subject as possible.

Load Balancing and Fault Tolerance Smart proxies can be made to represent multiple subjects and balance the number of requests each subject has to handle. Likewise, the proxy can use the fact that multiple subjects exist in order to provide fault tolerance.

Security Smart proxies can be used to perform additional validity checks or pass along extra arguments to the subject, such as security certificates needed to authenticate the request.

State Sharing A subject may have several proxies in different object spaces. Conceptually, all proxies share the subject's state. Technically, this might be realised by equipping smart proxies with advanced replication machinery.

Because of the way middleware solutions make proxies explicit, they can be adapted to incorporate this kind of behaviour. In CORBA, for example, proxies are generated by the middleware. A smart proxy can be created by inheriting from the generated proxy class, overriding certain methods to achieve supplementary behaviour [38]. The drawback of CORBA's smart proxies is that their customised code must be present at the client site (because the client needs a smart proxy at run-time) which requires the client to be modified whenever changes are made to the server code (because it is the server issuing the smart proxy code). In open networks where new clients may appear unheraldedly, this is problematic.

Java's DOC solution, Java RMI, also automatically generates proxies using Java's RMI stub compiler. Smart proxies can be created in Java in much the same way as in CORBA, with the same drawbacks. A better way to implement smart proxies in Java that avoids these drawbacks is to exploit Java's class loader to download new proxy classes from a server at run-time [39]. A drawback with this approach is that the application might have to substantially refactor its class hierarchy in order to meet the class and interface requirements for such smart proxies. Also, dynamic class loading is not always allowed because of constraints imposed by the security manager as exemplified by the applet sandbox model.

In spite of the technical problems associated with smart proxies in Java RMI and CORBA-like middleware, smart proxies are a good abstraction technique because they provide a convenient place to encapsulate advanced functionality. This makes DOC applications much easier to reason about. Unfortunately, in existing distributed languages, remote object references are fully transparent which precludes the creation of such smart proxies.

2.3 Summary

The problems revealed in this section seem to have locked us in a stalemate. Either one chooses for a distributed object-oriented programming language that deals with remote object references in a transparent way, or one reverts to proxy-based middleware on top of a standard object-oriented programming language. The former defers all distribution aspects to the programming language infrastructure precluding developers to fully exploit their knowledge. The latter is a source of numerous problems because the discrepancy between proxies and their subjects interacts with basic object-oriented language features such as static typing, reflection and object identity.

In the remainder of the paper we show that there is a way out of this apparent stalemate by conceiving proxies as delegation-based descendants of their subjects. We solve the problem by making the delegation feature of prototype-based languages applicable to objects that reside in distributed object spaces. We have explored this language feature by incorporating it in an experimental distributed toy language called ChitChat, which is the topic of the rest of the paper.

3. LANGUAGE DESIGN ASSUMPTIONS FOR OPEN NETWORKS

Before delving into the technical details of ChitChat, we first list the fundamental assumptions that have shaped it. These assumptions are based on the properties of the dynamically formed networks we described in the introduction. Our most fundamental assumption is that we have chosen our language to be object-oriented and that the object is the basic unit of distribution such that communication between distributed parties basically happens through message sending. The additional considerations that have influenced the design of ChitChat are:

Concurrency: Because the objects in a network typically reside on different hardware devices, concurrency is a natural phenomenon we have to deal with. One of the key decisions in designing a concurrent object-oriented language is the choice between synchronous and asynchronous message sending. Synchronous message sends are associated with a thread-based concurrency model such as that of Java. In synchronous message sending, the sender is blocked until the execution of the remote method finishes. This model is conceptually easy to understand, yet extremely error-prone [20]. The epitome of asynchronous message sending is the actor model [1]. Although extremely elegant, this model is, in its pure form, considered to be unpracticable because of the lack of mutable object state. The notion of an **active object**, as e.g. present in ABCL/1 [41], is considered a compromise between both approaches. Active objects have their own thread and message queue, just like actors, except that they *do* have mutable state. Active objects autonomously handle messages asynchronously.

In an open dynamic network environment, asynchronous remote message sends have advantages over synchronous ones. The connections in such networks are often wireless and unstable: a connection between two devices can break easily. If a device were in the process of handling a remote message synchronously, this can render the sender blocked for a long period of time, which is unacceptable. Moreover, asynchronous messages decouple the sender and receiver of a message in time, meaning that both do not necessarily have to be connected to the network at the same time in order to communicate [26]. Section 5.1 provides a more detailed discussion on ChitChat's active objects.

Flexible Object Graph Definition: Distributed object graphs glued together by proxies can become very complex. This is even more so in hardware constellations that consist of machines that dynamically form networks and start sending messages around in an object-graph the topology of which is not statically determined. This has led us to opt for a **dynamically typed prototype-based language**. First, in statically typed languages, it is cumbersome to inject new types of objects (i.e. new classes) at run-time. Even if it is possible (as in a restricted form with CORBA's Dynamic Invocation Interface or with Java classloaders) their static types will be abstract (i.e. `Object`) since the concrete subtypes are unknown at compile-time. This nullifies the benefits of static typing and has led us to a dynamically typed language. Second, in the given hardware context, classes are another source of problems. The creation of objects requires a class which inevitably leads to copying classes over the network. This poses many problems related to the fact that classes share their class-variables and methods among their instances. When classes are copied over a network, the former lead to replication problems and the latter to versioning problems. Classless objects encapsulating their own methods and state have proven to be more flexible in distributed applications. Indeed, with the exception of Argus [23] – which was not designed for *open* network environments – most well-known distributed programming languages such as Emerald [3, 16], Obliq [7] and dSelf [33], are all classless.

Language Level Security: Security is an essential ingredient of distribution. This is even more so if networks are defined dynamically and potentially malicious nodes might enter the network. As analyzed by Thorn [32], security is an issue that affects all levels, varying from the operating system and network layers up to the programming language in which one writes distributed systems. Vitek [36] defines a secure system as "[...] a programming system that prohibits insecure programs". This means that languages where the objects are not *de facto* protected from external encapsulation breaches by the language are flawed right from the start. This is particularly important when designing a prototype-based language. Indeed, Snyder [28] already showed that inheritance in a class-based language is an encapsulation-breaching operation because inheriting code necessarily "sees" more details of a superclass than client code. In [30] this was shown to be a fundamental problem of prototype-based languages: because inheritance is defined directly on objects, objects are constantly vulnerable to breaches of encapsulation. As shown below, ChitChat solves these issues by adhering to a principle we call **extreme encapsulation**. The key idea is that objects are fully encapsulated and are only subject to message passing, excluding "operators" such as inheritance, cloning, slot modification, parent assignment and so on. How this extremely restricted object model still enables phenomena such as inheritance, cloning and object creation, is described in more detail in the following section.

4. CHITCHAT'S OBJECT MODEL

Before delving into distribution and concurrency in section 5, the foundations of extremely encapsulated objects are introduced by situating ChitChat in the realm of prototype-based languages.

4.1 Prototypes and Delegation

Along with Self [34] and ACT1 [22], ChitChat belongs to the delegation-based branch of the prototype-based language family. Thus, ChitChat object extensions can delegate messages at run-time to their parent object [21]. Unlike Self, ChitChat features a more rigid object inheritance scheme where each object has exactly one parent. Furthermore, the delegation link between parent and child is implicit and fixed at object creation time. When a message is sent to an object but no corresponding method is found, it is implicitly delegated to its parent. If an implementation is found in a parent, self-sends will properly be sent to the original receiver, a property referred to as *late binding of self*. An important property of delegation-based systems is *parent sharing* which means that the same object can be a shared parent for two or more descendants. Changes in the parent state made by one descendant are then visible to other descendants too.

4.2 Extreme Encapsulation

As mentioned before, language operators such as cloning, object extension, slot addition or deletion, etc. breach object encapsulation because they allow one to bypass an object's message sending interface. This security concern has caused us to conceive ChitChat as a prototype-based language that features message passing as the *only* operator applicable to objects. Other language operators are simulated by allowing objects to declare different kinds of *methods*. Depending on the kind of a method, extra actions are undertaken before and after its body is executed. The language distinguishes between ordinary methods, *view* methods and *cloning* methods. A view (resp. cloning) method is declared by prefixing its method name by `view` (resp. `cloning`). In the following code excerpt, `incr` and `decr` are ordinary methods, `new` is a cloning method and `makeCounter` and `makeProtected` are view methods. The entire expression is the declaration (denoted by `::`) of a view method (denoted by the prefix `view`) called `makeCounter`.

```
view.makeCounter(n) :: {
  incr(v) :: { n := n+v };
  decr(v) :: { n := n-v };
  cloning.new(x) :: { n := x };
  view.makeProtected(limit) :: {
    incr(v) :: {
        if(n+v > limit,
           error("overflow: limit reached"),
           super.incr(v) )
    }
  }
}
```

As can be expected, when method lookup determines that a message sent to an object is implemented by an ordinary method, the body of the method is simply executed.

Upon invoking a view method, before the method body is executed, an extension of the receiver of the message is created. The extension is called a *view*, the original receiver is called the parent and the delegation link established between them is fixed. The view method's body is subsequently executed in the scope of the view. The return value of the method is the newly created view. Hence, it is not far wrong to think of a view method as a constructor. In the example above, `makeCounter` is a view method which spawns extensions on the root object by sending e.g. `root.makeCounter(3)`. As can be read from the body of the `makeCounter` view method, such newly created objects will contain a variable `n` and will respond to four messages, to wit `incr`, `decr`, `new` and `makeProtected`. `makeProtected` can be used in its turn to dynamically generate extensions of a counter object. Notice that a protected counter overrides the `incr` method of its parent.

Executing a cloning method leads to the creation of a clone of the receiver. The body of the cloning method is executed in the scope of the clone, allowing for the internal state of the clone to be initialized correctly. In the example, invoking `c.new(5)` on a previously created counter object `c` results in a clone whose instance variable will be set to 5.

With the special kinds of methods, ChitChat gives objects the explicit control over all critical operations because there are no additional language operators defined on objects apart from message passing. We say that objects are extremely encapsulated since they can decide independently whether they can be extended or cloned. Objects that do not contain view and/or cloning methods cannot be tinkered with. Since an object provides the code of a view or cloning method itself, it can also precisely prescribe how its descendants or clones will behave. ChitChat adopts this extremely encapsulated object model from Agora [29, 30] where view methods were originally termed mixin-methods. Section 7 further comments on the implications of this model on software evolution.

5. DISTRIBUTED DELEGATION IN CHITCHAT

In section 3 we argued in favour of a secure concurrent prototype-based language as the basis for distribution. The model based on extremely encapsulated prototypes discussed in the previous section covers the security requirement. We now enhance this model with active objects, the basic unit of concurrency and distribution in ChitChat. As explained in the introduction, the key insight of the paper is that remote object proxies are nothing but delegation-based descendants of their subjects. This is the topic of section 5.3. First we explain how active objects behave and how they participate in delegation hierarchies.

5.1 Active Objects

Active objects in ChitChat are largely based on those of ABCL/1 [41] and, as shown in section 3, occupy the middle ground between the thread-based and actor-based extremes. They encapsulate a message queue and a computational thread which perpetually processes messages from that queue and executes the associated methods, one at a time. This excludes what is known as intra-object concurrency [5] such that race conditions on the internal object state are avoided. Messages sent to an active object are handled asynchronously but immediately return a promise (a.k.a. a future [2, 14, 24]). Such a promise acts as a placeholder for the result to be computed. Accessing this placeholder before the result is known will temporarily block the execution of the accessor. When the result is known, the promise is transparently replaced by its value and the accessor continues its execution. This scheme maximizes concurrency and ensures synchronous communication only *when strictly necessary*. Moreover, it has the advantage of allowing a concurrent program to be read as if it were a sequential one.

The following example illustrates active objects. Analogous to the way ordinary objects are created, active objects are spawned by executing an *active* view method (prefixed by `aview`). In the example, invoking `fib` creates an active object that contains one single method `do` whose body spawns concurrency by creating active objects and asynchronously sending them `do` as well. The `+` operator blocks until both promises of the concurrent tree recursion have been fulfilled.

```
aview.fib(n) :: {
  do():: {
    if(n<2,
       1,
       athis.fib(n-1).do()+athis.fib(n-2).do())}}
```

Because of the assumptions outlined in section 3, active objects are the unit of distribution in ChitChat. This implies that only active objects can be referred to over the network and that all network traffic consists of asynchronous messages. The relation between active objects and the ordinary objects of section 4 is that every ordinary object is reachable by at most one active object. In other words, every ordinary object is owned by exactly one active object. This guarantees ordinary objects to be free of race conditions because two different threads necessarily belong to two different active objects and can therefore never enter the same ordinary object. To satisfy this constraint that ordinary objects are never shared by active ones, ChitChat's *object passing rules* prescribe that ordinary objects are copied every time they are used as arguments or results in a message sent to an active object. Active objects, on the contrary, can be safely passed by reference.

5.2 Active Objects vs. Delegation

As the method prefix `aview` (for active view) suggests, active objects can participate in delegation hierarchies in the same way ordinary objects do. Active objects have exactly one parent object which has to be an active object in its turn[3]. Upon receiving a message that it does not implement, an active object will delegate that message to its parent. Since active objects are the unit of distribution, the method lookup algorithm may cross network boundaries while traversing the active delegation chain. When an active object is found that implements the method corresponding to the message, the message is scheduled in the queue of this particular object. Hence, when messages are delegated between active objects, it is the *provider* of the method that will eventually run the method. Arguments for the method and (a promise for) its result follow the object passing rules prescribed earlier. During the execution of the method, messages sent to the pseudo-variable `athis` end up in the queue of the original receiver which means that distributed delegation satisfies late binding of self, even when `athis` refers to a remote descendant. As can be expected, messages sent to the `asuper` pseudo-variable are explicitly delegated (asynchronously) to the parent and also satisfy late binding of `athis`.

Distributed delegation hierarchies of active objects emerge whenever one invokes an active view method on a remote active object. When this happens, the body of this method is dynamically downloaded on the host of the sender. The method is subsequently executed in the context of a newly spawned active object which resides on the machine of the sender and whose parent is the original receiver. Thus, messages sent to a remote active object that are implemented by active view methods lead to the creation of local active descendants. In conclusion, active views are *always* created on the site of the *sender* of the message that triggered the active view method. If sender and receiver are located on different virtual machines, an active view with a remote parent is created.

5.3 Distributed Proxies as Delegation-based Descendants

Active objects combined with distributed delegation form the basis for remote proxies in distributed applications. The key insight is that a proxy in ChitChat is conceived as an *empty view* on a remote object. In accordance with the object passing rules of section 5.1, this empty active view is automatically created when an active object is passed across network boundaries. The proxy automatically delegates (in the original prototype-based sense of the word) each request to its subject (read: parent) if it does not override that request. This behaviour is exactly the one shown by conventional proxies. But as argued below, the delegation-based approach has many advantages over the aggregation-based proxies discussed in section 2. Apart from the automatic message forwarding and the overriding capabilities, the late binding of self principle between a subjects and their proxies is respected. Applications hereof are postponed to section 6.

Proxies as empty delegation-based descendants enable a number of elegant DOC solutions. First, stub code that manually forwards each method invocation across the network is no longer required because this is the default behaviour of delegation. Second, it becomes trivial to augment proxies with dedicated behaviour, turning them into what were called smart proxies in section 2.2. All one has to do is to equip the subject with an active view method that overrides and/or adds certain methods in order to obtain that dedicated behaviour. Once an initial (automatically created and empty) proxy is obtained on the subject[4], it can be sent a message that invokes the active view method thereby spawning the proxy that implements the dedicated behaviour. In other words, ChitChat allows one to add methods in the proxy. Because these can override methods in the subject, some operations can be handled differently or locally (i.e. without delegating over the network). Figure 1 illustrates how a reference to a remote counter object can be used to spawn a local extension that applies these ideas. We will use the technique in section 6.1 to devise smart proxies that display such customised behaviour. First, we evaluate our approach in the light of the analysis presented in section 2.

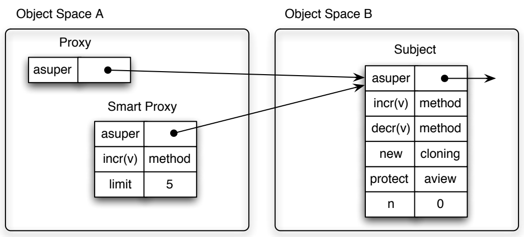

Figure 1: Modelling Remote Object Proxies as Delegation-based Descendants

5.4 Evaluation

As summarised in section 2.3 current DOC approaches require a programmer to choose between fully transparent distributed programming languages (excluding advanced DOC techniques like smart proxies), and the use of middleware technologies (whose proxies heavily interfere with basic object-oriented language features).

The goal of middleware is that application code using proxies should not be able to distinguish a proxy from its subject. However, middleware fails to conceal its proxies because language features beyond its control are able to reveal them. ChitChat avoids the discrepancy between proxies and subjects due to dynamic typing and extreme encapsulation which prescribes that all operations on an object are funnelled to the object's message sending interface. Hence, the idiosyncratic behaviour of e.g. object identity operators (such as `==`) is avoided, because ChitChat enforces the use of late-bound messages (such as `equals`). In general, extreme encapsulation allows a proxy to intercept any operation on its subject.

The pitfall of *full* transparency in other language-based approaches lies in the fact that their proxies are not only transparent to *client code*, but also entirely oblivious to the *programmer* who may wish to enhance their behaviour. This impediment is alleviated by conceiving proxies as (potentially empty) delegation-based descendants. These allow one to encapsulate the non-functional aspects of distribution, yielding the smart proxies we introduced in section 2.2. Concrete examples of such smart proxies in ChitChat are given in section 6.1.

ChitChat's dynamic download mechanism for active view code can be regarded as an object-based variant of Java's class loading mechanism. Both techniques achieve the incorporation of new behaviour

[3]We have experimented with hierarchies consisting of ordinary objects mixed with active ones. It turns out to be very hard to come up with a comprehensible semantics for this.

[4]ChitChat's service discovery mechanism, albeit outside the scope of the paper, allows one to obtain an initial empty view on a primeval subject. Other references are subsequently obtained through message sending.

at the client side at run-time. Hence, customized proxy code does not have to be known to all client sides at compile-time, but is downloaded just-in-time. This benefits the extensibility of a program since clients will automatically download newer versions of the proxy code if the subject is upgraded. Also, the addition of a new method to a subject is gracefully handled by existing proxies, since the default behaviour will delegate the new requests to the subject. There is no need for the proxy to manually forward the request unless specific proxy behaviour is required.

Compared to Java's class loading mechanism, which can be heavily restricted for security reasons, ChitChat does not constrain the download mechanism of proxies. Due to extreme encapsulation, the downloaded proxy can tamper with its subject only. This does not introduce security breaches since the subject prescribes the proxy's behaviour, ruling out the creation of malicious descendants.

5.5 Parent Sharing and Scoping Rules

Before we present a number of ChitChat programming idioms resulting from representing proxies as delegation-based descendants, more technical insight in ChitChat's delegation mechanism is required. As indicated in section 4.1, parent sharing is a natural phenomenon of delegation-based languages. Parent sharing also occurs when a subject has multiple proxies (possibly spawned by one or more active view methods). However, delegation hierarchies consisting of active objects easily lead to race conditions when two active objects have access to the same parent [6]. In order to prevent those race conditions, delegation hierarchies of active objects require special scoping rules. In traditional delegation-based languages, descendants have privileged scoping in the sense that they have direct access to their parent's variables. This becomes problematic if two active objects concurrently read and write variables in a shared active parent, as exemplified by the following code:

```
aview.makeParent(x) :: {
  aview.makeChild() :: {
    incr() :: { x := x + 1 }
  }
}
p :: makeParent(0);
c1 :: p.makeChild();
c2 :: p.makeChild(); ‘c1 and c2 share p as a parent‘
```

If `c1` and `c2` concurrently handle an `incr()` message, race conditions may occur: `c1` might read the value for `x` and find it bound to `0`. Meanwhile, `c2` reads out `x`, binding it to `0` as well. Next, `c1` writes `1` back to `x`. Finally, `c2` also stores `1` in `x`.

ChitChat solves this problem by restricting the scope of active descendants. An active descendant has *no* direct access to the slots of its parent. Rather, the scope of a descendant's method is restricted to the descendant itself and, of course, the local scope of the method. Hence, a method can only manipulate local variables of the method and of the descendant itself. In Java terminology, parent variables in ChitChat are private instead of protected. To access state in the parent, the scope must be widened *explicitly* using the *scope opening message* `asuper(code)`. The meaning of an expression of the form `asuper(code)` is to request the parent to execute the expression `code` in its scope. As such, descendants can still read and modify parent variables. Reconsidering the above example, the body of `incr` is not valid ChitChat code since `x` is a variable belonging to the parent. The method must therefore be implemented as `asuper(x:=x+1)`.

A characteristic of the scope opening message `asuper(code)` is that it actually schedules a request to execute the code in the parent's message queue. The absence of intra-object concurrency in ChitChat implies that only one such request is served at a time. This ensures that requests from multiple descendants are always executed serially such that race conditions in the parent are precluded. The descendant-parent communication based on scope opening messages confirms the philosophy that active objects are autonomous: even descendants cannot *directly* change the state of another active object. This is a key characteristic of delegation in ChitChat.

As with other messages sent to active objects, the `asuper(code)` expression immediately returns a promise that will eventually be replaced by the value of `code` once it has been executed by the parent. Furthermore, the block of code sent to the parent respects late binding of self such that occurrences of `athis` in the block refer to the descendant that sent the code. Analogous to the way code is transmitted to the parent using `asuper(code)`, the parent can in turn send code back to the descendant using the scope opening message `athis(code)`. By nesting such scope opening messages, children can access parent variables and vice versa. The subtleties of the pseudo-variables `athis` and `asuper` (explained in section 5.2) and the scope opening messages `asuper(code)` and `athis(code)` are depicted in figure 2 and further exemplified by the idioms of the following section.

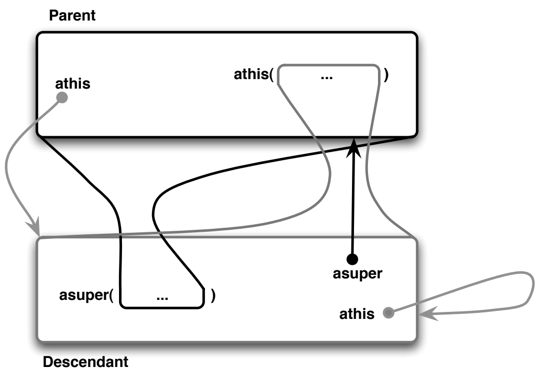

Figure 2: Scope Opening Messages vs. Pseudo-variables

Notice that the dynamic code uploading mechanism introduced by `asuper(code)` may at first seem to introduce security breaches because code is injected by a descendant into its parent. However the code that is uploaded this way was originally defined by that parent because the descendant itself was spawned by that parent. Thanks to extreme encapsulation, descendant objects can only be created by (active) view methods and therefore a parent has full control over its descendants.

6. CHITCHAT PROGRAMMING IDIOMS

We now illustrate how ChitChat's unique features can be used to implement a collection of quite complex DOC phenomena in an elegant and expressive way. We start by showing the ease with which smart proxies are conceived. Section 6.2 shows how ChitChat generalizes Java's popular applet technique in a secure way. Finally section 6.3 sheds some light on how distributed delegation forms the basis for a programming language with simple built-in transactions.

6.1 Example Smart Proxies

In section 2.2, fully transparent approaches to DOC have been criticised because they preclude programmers from using their knowledge to fine tune the behaviour of their proxies. Smart proxies were advocated to implement such tailor-made remote references. We now illustrate how the network performance of an application can be improved using smart proxies by means of two examples, to wit caching and request batching.

6.1.1 *Caching*

We illustrate caching smart proxies using a simplistic stock exchange network service. The service is presented as a remotely accessible active object that encapsulates a database of stock quotes. It might be created by invoking the active view method given in the following code excerpt[5].

```
aview.makeStockExchange(database) :: {
  proxies : makeList();
  requestStockPrice(symbol) :: {
    database.find(symbol) };
  updateStockPrice(symbol, price) :: {
    database.update(symbol, price);
    proxies.iterate({elt.refresh(symbol, price)})}
  topStocksTradedToday() :: { ... };
  aview.makeCachingProxy() :: {
    cache : makeMap();
    requestStockPrice(symbol) :: {
      if (cache.contains(symbol),
          cache.get(symbol),
          { price: asuper.requestStockPrice(symbol);
            cache.put(symbol, price);
            price }) };
    refresh(symbol, price) :: {
      if (cache.contains(symbol),
          cache.set(symbol, price)) };
    asuper(proxies.add(athis)) } }
```

The server spawned by this method is an active object that has two private variables (`database` and `proxies`) and that implements four methods (`requestStockPrice`, `updateStockPrice`, `topStocksTradedToday` and `makeCachingProxy`). Other active objects on the network will typically act as clients and use ChitChat's service discovery mechanism (which is outside the scope of the paper) in order to obtain a network reference to the server. Remember from section 5.3 that such a reference is nothing but an empty descendant of the object. Messages sent to this empty view (read: proxy) are automatically delegated to the parent (read: subject). Examples of such messages are `requestStockPrice` which queries the server for a stock price given a ticker tape symbol, and `topStocksTradedToday` which returns today's top deals.

The focus of our attention is the `makeCachingProxy` active view method. Upon invoking it, a descendant of the server is created on the site of the sender of the `makeCachingProxy` message. The new object contains a private variable `cache`, overrides the `requestStockPrice` method, inherits `topStocks-TradedToday` and adds the `refresh` method. This is the smart proxy that implements the caching behaviour: instead of forwarding each request for a stock quote, it caches previous request results in `cache`. In order for the proxy to be notified when a stock price is updated, it registers itself with the stock service which happens in the last expression of the active view method. In order to have access to the `proxies` variable, this expression needs to be evaluated at the server site using the `asuper` scope opening message. Because `athis` is late bound, it refers to the smart proxy.

Whenever the server is updated upon processing an `updateStock-Price` request, the `iterate` form is used to loop[6] over the `proxies` variable in order to notify all smart proxies by sending them `refresh`. This will cause the smart proxies to update their cached prices.

6.1.2 *Request Batching*

The second illustration of smart proxies that reduce network traffic is known as request batching. The idea is that a smart proxy accumulates a number of actions on a remote subject such that the accumulated actions can be sent in bulk to the subject. The code below shows the technique in the context of an oversimplified distributed drawing board application.

```
aview.makeDrawingBoard() {
  draw(figure) :: { ... };
  aview.makeBatchProxy() {
    batchlist : makeList();
    draw(figure) :: { batchlist.append(figure)};
    flush() :: {
      snapshot : batchlist.copy();
      batchlist.clear();
      asuper(
        athis(snapshot).iterate({draw(elt)}) )
    } } }
```

A drawing board has a `draw` method which can be used by a client to draw something on a centralized whiteboard. Clients can download a batch proxy onto their machine by sending `makeBatch-Proxy` to the drawing board which will buffer all `draw` requests until `flush` is invoked[7]. `flush` creates a snapshot of the buffered requests such that the buffer can be cleared. `asuper` is used to invoke the buffered requests. The code contained within `asuper` is uploaded to and executed by the drawing board server. The server then requests a copy of the snapshot and iterates over the list locally. Hence, instead of having to transmit each figure in a separate remote method invocation, they are effectively transmitted in batch.

6.2 Connected Applets

Remember from section 5.2 that an `aview`'s method body is dynamically downloaded if the sender and the provider of the method are not co-located. This mechanism resembles the behaviour of Java applets, whose code is dynamically downloaded from a server and executed locally by the virtual machine in the client's web browser. In this setup it is very common that, after downloading the applet code and starting the execution of the new applet, a connection is established between the applet and its spawning server in order to retrieve additional data. An extreme instance of this is the popular Volano[8]chat server which spawns chat applets that immediately establish a lifetime connection with their server. These *connected applets* are trivially modeled in ChitChat by programming

[5]Notice that ChitChat object slots are either public, immutable slots (declared using `::`) or private, modifiable slots (declared using `:`). The rationale behind this is unravelled in [12].

[6]The `iterate` form takes an expression over `elt` and evaluates that expression with `elt` bound to each element in the collection. See [11] for more details.

[7]Typically, other methods of the smart proxy will invoke `flush` thereby avoiding pollution of client code.

[8]http://www.volano.com

them as descendant objects of their server. The code of the server and the applets can use the `asuper` and `athis` scope opening messages to communicate. We illustrate the technique by presenting the framework for a distributed chat system written in ChitChat.

The chat application is modelled as an active server object that spawns — upon request — remote active clients as distributed delegation-based descendants. The fact that the clients are conceived as descendants of the server might seem awkward at first, but only for those who associate the terms 'client' and 'server' with client-functionality (e.g. opening connections to a server) and server-functionality (e.g. polling or listening for requests). From a conceptual point of view, the chat server merely contains the information shared by the chat clients. In ChitChat it is therefore implemented as a shared parent object of those clients. The local client is literally a view on that shared state. We will get back to this in the discussion of section 7.

The overall structure of the code is shown below. As explained, a chat server is an active object (created by invoking `makeChatServer`) that can spawn chat clients as delegation-based descendants. This is accomplished by sending `registerClient` to a remote chat server.

```
aview.makeChatServer(channelName) :: {
   clients : makeList();
   aview.registerClient(name) :: {
        ...
   } };
```

The detailed implementation of the `registerClient` method is given below. Upon invoking it remotely, the method body is downloaded and executed in the context of a newly created object (at the site of the sender) that has the chat server as its parent. As can be seen from the source text, executing the method body will install methods `receiveMsg` and `sendMsg` (which is expanded further on) in the client and will register that client in the server using the same technique that was used to register the caching smart proxies in the previous section. In the code this is resembled by the fact that the server keeps track of its clients in the `clients` variable. Updating this variable to register a new client is done with the `asuper` scope opening message. It uploads and schedules the registration code in the server's message queue.

```
aview.registerClient(name) :: {
   receiveMsg(from,msg) :: {print(from,": ",msg)};
   sendMsg(msg) :: { ... };
   asuper(clients.add(athis))
};
```

A client's `receiveMsg` method will be invoked by the server whenever another client has sent a message to the chat channel. It will simply print the identification of that other client on its GUI along with the message. The other way around, a GUI at a client's site is expected to send `sendMsg` to its chat client object whenever the chatter types a line. The implementation of `sendMsg` also resides in the client object and broadcasts the message to all connected clients. This is shown below. Upon receiving `sendMsg`, a client asks its parent to broadcast the message to all the clients contained in the `clients` list using the `asuper` scope opening message. The server, in turn, requires the name of the client that sent the message. It is obtained by the `athis(name)` expression yielding a promise which is used when `receiveMsg` is sent to all connected clients. Because of asynchronous message sending, the iteration will not wait for a result before entering its loop again.

```
sendMsg(msg) :: {
  asuper({
    from: athis(name);
    clients.iterate({elt.receiveMsg(from,msg)})})};
```

The chat example presented here is not production code because it lacks such things as a GUI and is not fault tolerant. Nevertheless, it clearly illustrates the expressiveness of ChitChat's features: the above 10 lines of code allowed us to effectively run the core functionality of a distributed chat application in our prototype implementation of ChitChat.

6.3 Serialized Transactions

A final idiom that illustrates the elegance and expressiveness of ChitChat's distributed delegation consists of expressing simple serialized distributed transactions. As in section 6.1, a simplistic distributed drawing application in which clients share a remote white board will be used as an example to illustrate the technique. Typically, some concurrently running client requests made to the white board require mutually exclusive access. For example, a call to `drawLine` may not proceed in parallel with a call to `getBoardContent`. In Java, this is accomplished by marking those methods with the `synchronized` modifier. In ChitChat, this is the default behaviour because active objects exclude intra-object concurrency.

For the sake of the example consider a client that needs an operation to draw *two* lines on the white board atomically such that other clients always see both lines rendered, or none at all. The solution in a typical client-server architecture would be to add another synchronized method `draw2Lines` to the server that does the job. This illustrates that a synchronized method has to be added to the server for every combination of requests that one wants to see handled in an atomic way. For example, a client that wants a rectangle (as four lines) without concurrent interference of other clients will require the server to be extended with ad-hoc methods such as `drawRectangle`. This approach of extending the server with every possible meaningful operation that might have to run atomically does not scale. The same phenomenon occurs in the context of databases. Since the database designer cannot know beforehand what operations (table insertions, deletions, queries) have to be performed atomically, database management systems put the burden of synchronisation with the clients by requiring them to manually group their actions in transactions. This is precisely the vision endorsed by ChitChat. The idea is that a client (read: descendant) can ask a server (read: parent) to execute a chunk of code atomically simply by uploading that code using the `asuper(...)` scope opening message. This is exemplified by the following code excerpt which illustrates how a client might ask a server to accomplish the same task that would require us to add a method `draw2Lines` in other languages.

```
asuper({ drawLine(line1,color1);
         drawLine(line2,color2) })
```

This kind of client-side synchronisation might also be achieved in existing client-server applications by employing locks. Indeed, a client could explicitly lock and unlock the server object and perform the necessary requests in between. However, this is quite error-prone since the technique of manually managing locks puts an extra cognitive burden on the programmer. Also, if a malicious or erroneous client takes a lock on the server and does not release it, the server could be rendered useless unless the server is provided with sophisticated protection code. The transaction-based solution of ChitChat does not suffer from this problem. It is the server (parent) itself that executes the code contained in an `asuper` block. The parent will automatically serve new requests when it finishes executing the block. Furthermore, because the client that uses the `asuper(...)` scope opening message was itself spawned by the server, security is guaranteed.

7. DISCUSSION AND FUTURE WORK

This section contemplates the solutions presented in the paper and discusses the possibilities for improvement and further research.

Delegation Versus Synchronisation

In one of the 'classic' papers on object concurrency, Briot and Yonezawa [6] meticulously explain that concurrency in a delegation-based language is a source of synchronisation problems. The reason is that both variable access and variable assignment in their delegation-based language under study is based on asynchronous message passing. As such, variable access and update messages can become intertwined causing race conditions when two expressions of the form `x:=x+1` are concurrently executed. In our work, we were able to reconcile delegation with synchronisation by blocking the scope between parents and descendants. By introducing explicit scope opening messages `asuper` and `athis` we still obtain the desired scoping. Making the scope opening explicit gives the virtual machine the ability to ensure atomic execution.

Delegation Semantics

The conventional semantics attributed to delegation and inheritance is that of an **is-a** relationship between the descendant and its parent. It is not wrong to think of a ChitChat (smart) proxy in the same way: a proxy for a complex number is-a complex number, albeit a remote one. However, as already explained briefly in section 6.2, the distributed delegation relationship does not necessarily adhere to that semantics. This is the case when distributed delegation is combined with parent sharing which gives rise to a **shares-a** semantics as exemplified by the chat server. In that case, the distributed descendants are literally conceived as local views on a remote shared object.

Conditional Synchronisation

An important issue in concurrent object-oriented programming is how to temporarily enable and disable methods of an object. An example is to disable a 'read' operation on a empty buffer until a 'write' operation has occurred. Although this was not neglected in ChitChat, a complete description of the mechanism is beyond the scope of the paper. It basically consists of a reification of promises such that objects can grab first class promises for future fulfillment. For more information we refer to [10].

External Object Extensions

Because object creation in ChitChat is aligned with message passing (i.e. a view method needs to be invoked), it may seem that object hierarchies require a lexical nesting of all potential extensions in the code of the object. Clearly, from a software engineering point of view, this hampers unanticipated code reuse. This problem has been solved before [9]. Its solution basically consists of a code quoting mechanism that allows one to inject code in an object (as always, by sending it a message).

Implementation Issues

At the time of writing, ChitChat has been provided with a tail-recursive interpreter written in Java. After an unsuccessful attempt to implement the distributed aspects of the language using SOAP technology [35], the current implementation uses plain Java RMI for this purpose. In the current status of the implementation, our entire focus of attention was getting the semantics of the language right. Implementation issues such as efficiency were often neglected. Investigating optimisation techniques for the language is an interesting topic of future research. We expect to reuse a lot of the low-level implementation technology that was developed to provide an efficient implementation of a (non-distributed) predecessor of ChitChat [11]. For a more formal specification of ChitChat, we refer to [10].

Future work encompasses more programming language support for expressing distributed delegation patterns and the incorporation of partial failure handling tailored to open ad-hoc networks.

Us

Many experiments conducted in ChitChat involve registering remote descendants with the parent that has spawned them. This was the case in the caching example of section 6.1 and in the chat example of section 6.2. Managing these references manually obviously has a number of drawbacks. We therefore plan to incorporate them into the language such that a parent has hidden links to the descendants it has spawned. This could form the basis for an `us(code)` scope opening message that generalizes the `this(code)` scope opening message which would broadcast a message to all descendants. It would allow one to notify all descendants of a shared parent in one stroke. However, this requires more research.

Advanced Delegation Schemes

As explained in section 5.2, the single delegation link between two active objects in ChitChat is fixed when creating the descendant. This has its restrictions. Since ChitChat features only single inheritance, an active object cannot extend one object in the **is-a** sense and another in the **shares-a** sense. Another problem of single inheritance in ChitChat is that an object cannot share state with multiple objects. Hence, ChitChat's smart proxies cannot have more than one subject. Also, since parent assignment is not allowed, smart proxies cannot dynamically change their subject. Experimentation already has revealed that adding parent assignment to the language without violating the extreme encapsulation principle is not trivial. A way out might be a grouping mechanism similar to the `us` construction that allows one conceptual parent to embody several existing objects. Parent assignment then boils down to toggling between the members of the group. These advanced delegation schemes are required to enable some of the advanced roles attributed to smart proxies in section 2.2.

Partial Failure Handling

A problem of DOC that our work does not address at all is the problem of partial failure. From this perspective, losing objects from a distributed active object graph in ChitChat is as problematic as in any other DOC system. But notice that the extra structure imposed by a delegation hierarchy might help to ease the problem because more structural information is available for the virtual machine. E.g., the examples of section 6 where a subject explicitly keeps a reference to all its descendants are more vulnerable than if this would be accomplished implicitly with an `us(code)` construction. The latter would simply not broadcast the message to lost descendants. The other way around, lost parent pointers might be dealt with by fully replicating the shared parent's state to the clients. The `us(code)` construction would prove helpful to facilitate programming such advanced replication machinery in the parent. However, the design of language features that give support to deal with partial failures is future work.

8. RELATED WORK

Most related work concerned with inheritance and *concurrency* involves class-based inheritance schemes and studies the inheritance anomaly. In [18], reuse and sharing are clearly distinguished as two different facets of inheritance. It is argued that inheritance is more useful for reuse and maintenance, while delegation emphasizes flexibility in sharing. Class-based languages with active objects like ACT++ [17] and Eiffel// [8] lack parent sharing and are therefore not confronted with race conditions following from hierarchically structured active objects. In actor languages such as ACT1 [22] and ACT3 [15] that *do* enable such hierarchies, parent sharing has never been exploited for managing state shared by different concurrent actors. Instead, these languages encapsulate shared state in special *guardian* or *receptionist* actors.

ChitChat builds upon prior work in the field of *distributed* object-oriented programming languages. ChitChat's distinction between active and passive objects resembles Argus's [23] dichotomy between ordinary and *guardian* objects. The languages Emerald [3, 16] and Obliq [7] both feature a prototype-based object model but do not feature delegation, such that parent sharing is unattainable. Active objects in ChitChat can widen their scope to that of a remote parent. As such, they achieve scoping similar to Obliq's distributed lexical scope for closures. To the best of our knowledge, the language dSelf [33] – a distributed version of Self – is the only language achieving the same kind of distributed delegation as ChitChat. Unfortunately, the language lacks a sound concurrency model. Also, dSelf inherits Self's object model in which object encapsulation is easily breached, impeding the construction of secure distributed programs.

9. CONCLUSIONS

The paper identifies a number of problems exhibited by proxies in contemporary middleware solutions. Some of these are quite technical and directly relate to static typing. Others however, are more fundamental and boil down to the fact that a proxy and a subject conceptually denote the same object but are technically represented by two idiosyncratic objects. The other extreme of the DOC spectrum is formed by distributed programming languages that completely hide the existence of proxies. This approach has the drawback of prohibiting programmers to use their knowledge in order to improve an application's performance by incorporating that knowledge in smart proxies.

The key insight of this paper is to align distributed proxies with delegation-based descendants of active objects. Conventional proxies are nothing but empty extensions that delegate every request to their subject by default. Smart proxies are conceived as proxies that override and/or add behaviour to the descendant. In ChitChat, special scoping rules and scope opening messages manage the visibility and modifiability of state between the parent and its descendants. These generalizations of prototype-based delegation to a concurrent and distributed setting were shown to facilitate the construction of a number of quite complex DOC programming problems such as expressing simple serialized transactions, serving shared state amongst connected applets and the realisation of smart proxies.

10. ACKNOWLEDGEMENTS

We would like to thank Pascal Costanza for his helpful comments on earlier versions of the paper.

11. REFERENCES

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

[2] H. G. Baker Jr. and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of Symposium on AI and Programming Languages*, volume 8 of *ACM Sigplan Notices*, pages 55–59, 1977.

[3] A. Black, N. Hutchinson, E. Jul, and H. Levy. Object structure in the Emerald system. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 78–86. ACM Press, 1986.

[4] S. Brandt and O. L. Madsen. Object-Oriented Distributed Programming in BETA. In *ECOOP '93: Proceedings of the Workshop on Object-Based Distributed Programming*, pages 185–212. Springer-Verlag, 1994.

[5] J.-P. Briot, R. Guerraoui, and K.-P. Lohr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, 1998.

[6] J.-P. Briot and A. Yonezawa. Inheritance and Synchronization in Concurrent OOP. In J. Bézivin, J.-M. Hullot, P. Cointe, and H. Lieberman, editors, *Proceedings of the ECOOP '87 European Conference on Object-oriented Programming*, pages 32–40, Paris, France, 1987. Springer Verlag.

[7] L. Cardelli. A language with distributed scope. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 286–297. ACM Press, 1995.

[8] D. Caromel. Towards a method of object-oriented concurrent programming. *Communications of the ACM*, 36(9):90–102, 1993.

[9] W. De Meuter. Agora: The story of the simplest mop in the world - or - the scheme of object orientation. In J. Noble, I. Moore, and A. Taivalsaari, editors, *Prototype-based Programming*. Springer-Verlag, 1998.

[10] W. De Meuter. *Move Considered Harmful: A Language Design Approach to Mobility and Distribution for Open Networks*. PhD thesis, Vrije Universiteit Brussel, 2004.

[11] W. De Meuter, T. D'Hondt, and J. Dedecker. Pico: Scheme for mere mortals. *1st European Lisp and Scheme Workshop, Ecoop 2004*, 2004.

[12] T. D'Hondt and W. De Meuter. On first-class methods and dynamic scope. *RSTI - Lobjet 9/ 2003. LMO 2003*, pages 137–149, 2003.

[13] R. Guerraoui and M. E. Fayad. OO Distributed Programming is *Not* Distributed OO Programming. *Communications of the ACM*, 42(4):101–104, 1999.

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

[15] C. Hewitt, T. Reinhardt, G. Agha, and G. Attardi. Linguistic support of receptionists for shared resources. In *Proc. of the NSF/SERC Seminar on Concurrency*, pages 330–359. Springer-Verlag, 1984. also MIT AI Memo 781.

[16] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems*, 6(1):109–133, February 1988.

[17] D. Kafura. Act++: building a concurrent C++ with actors. *Journal of Object-Oriented Programming*, 3(1):25–37, 1990.

[18] D. G. Kafura and K. H. Lee. Inheritance in actor based concurrent object-oriented languages. *Comput. J.*, 32(4):297–304, 1989.

[19] D. Lea. Design for Open Systems in Java. In *COORDINATION '97: Proceedings of the Second International Conference on Coordination Languages and Models*, pages 32–45. Springer-Verlag, 1997.

[20] D. Lea. *Concurrent Programming in Java: Design Principles and Patterns*. Addison-Wesley, second edition, November 1999.

[21] H. Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223. ACM Press, 1986.

[22] H. Lieberman. Concurrent object-oriented programming in ACT 1. In A. Yonezawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, pages 9–36. MIT Press, 1987.

[23] B. Liskov. Distributed programming in Argus. *Communications Of The ACM*, 31(3):300–312, 1988.

[24] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988.

[25] S. Maffeis. A flexible system design to support object-groups and object-oriented distributed programming. In *Proceedings of the ECOOP '93 Workshop on Object-Based Distributed Programming*, pages 213–224. Springer-Verlag, 1994.

[26] C. Mascolo, L. Capra, and W. Emmerich. Mobile Computing Middleware. In *Advanced lectures on networking*, pages 20–58. Springer-Verlag New York, Inc., 2002.

[27] Object Management Group. Common Object Request Broker Architecture: Core specification, 2002. `http://www.omg.org`.

[28] A. Snyder. Encapsulation and Inheritance in Object-oriented Programming Languages. In *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*, pages 38–45. ACM Press, 1986.

[29] P. Steyaert, W. Codenie, T. D'hondt, K. De Hondt, C. Lucas, and M. Van Limberghen. Nested mixin-methods in Agora. In *Proceedings of the European Conference on Object Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 197–213, 1993.

[30] P. Steyaert and W. De Meuter. A marriage of class- and object-based inheritance without unwanted children. In *Proceedings of ECOOP '95*, volume 952 of *Lecture Notes in Computer Science*, pages 127–144. Springer, August 1995.

[31] Sun Microsystems. Java RMI specification, 1998. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

[32] T. Thorn. Programming languages for mobile code. *ACM Computing Surveys*, 29(3):213–239, 1997.

[33] R. Tolksdorf and K. Knubben. Programming distributed systems with the delegation-based object-oriented language dSelf. In *Proceedings of the 2002 ACM symposium on Applied computing*, pages 927–931. ACM Press, 2002.

[34] D. Ungar and R. B. Smith. Self: The power of simplicity. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242. ACM Press, 1987.

[35] T. Van Cutsem, S. Mostinckx, W. De Meuter, J. Dedecker, and T. D'Hondt. On the performance of soap in a non-trivial peer-to-peer experiment. In *Proceedings of the 2nd International Working Conference on Component Deployment*, Lecture Notes In Computer Science. Springer Verlag, May 2004.

[36] J. Vitek, M. Serrano, and D. Thanos. Security and Communication in Mobile Object Systems. In *Mobile Object Systems: Towards the Programmable Internet*, pages 177–200. Springer-Verlag: Heidelberg, Germany, 1997.

[37] J. Waldo, G. Wyant, A. Wollrath, and S. C. Kendall. A note on distributed computing. In *MOS '96: Selected Presentations and Invited Papers Second International Workshop on Mobile Object Systems - Towards the Programmable Internet*, pages 49–64. Springer-Verlag, 1996.

[38] N. Wang, K. Parameswaran, and D. Schmidt. The design and performance of metaprogramming mechanism for object request broker middleware. In *Proceedings of the 6th USENIX Conference on Object-Oriented Technologies and Systems (COOTS'01)*, January 2001.

[39] J. Wilson. Get Smart with Proxies and RMI. In *Javaworld*, November 2000. `http://www.javaworld.com`.

[40] World Wide Web Consortium. Simple Object Access Protocol (SOAP) 1.2 W3C Recommendation, 2003. `http://www.w3.org/TR/soap12-part2/`.

[41] A. Yonezawa, J.-P. Briot, and E. Shibayama. Object-oriented concurrent programming in ABCL/1. In *Conference proceedings on Object-oriented programming systems, languages and applications*, pages 258–268. ACM Press, 1986.