Dynamic Layout of Distributed Applications in FarGo

Ophir Holder **Israel Ben-Shaul** **Hovav Gazit**

Department of Electrical Engineering
Technion — Israel Institute of Technology
Technion City, Haifa 32000, Israel
+972-4-829-{4659,4689,4659}
{holder@tx,issy@ee,ghovav@tx}.technion.ac.il

ABSTRACT
The design of efficient and reliable distributed applications that operate in large networks, over links with varying capacities and loads, demands new programming abstractions and mechanisms. The conventional static design-time determination of local-remote relationships between components implies that (dynamic) environmental changes are hard if not impossible to address without reengineering. This paper presents a novel programming model that is centered around the concept of "dynamic application layout", which permits the manipulation of component location at runtime. This leads to a clean separation between the programming of the application's logic and the programming of the layout, which can also be performed externally at runtime. The main abstraction vehicle for layout programming is a reflective inter-component reference, which embodies co- and re-location semantics. We describe an extensible set of reference types that drive and constrain the mapping of components to hosts, and show how this model elevates application's performance and reliability yet requires minimal changes in programming the application's logic. The model was realized in the FarGo system, whose design and implementation in Java are presented, along with an event-based scripting language and corresponding event-monitoring service for managing the layout of FarGo applications.

Keywords
Engineering Distributed Systems, Dynamic Objects, Distributed Components, Mobile Objects, Java

1 INTRODUCTION AND MOTIVATION

The growing adoption of large-scale networking infrastructures is vastly changing the architecture of software systems and applications. Many conventional standalone applications such as office automation and electronic publishing are becoming "network-enabled". Distributed client-server applications that were designed to run on a LAN must also be adapted in order to operate correctly and efficiently in a WAN scope.

Wide-area computing introduces new challenges to architects of scalable distributed applications. The large deployment space — i.e., the large number of available hosts that are connected by networks with varying capacities and loads — implies that the designer is unlikely to know a priori how to structure the application in a way that best leverages the available infrastructure. Furthermore, the constantly changing nature of global environments, such as varying network bandwidth, machine loads, and availability, implies that any assumptions that are made early at *design time* regarding the underlying physical system are unlikely to hold during deployment time, let alone throughout the application's lifetime.

Thus, any static and fixed determination of the local-remote partitioning and the overall mapping of the distributed application (logic) onto a set of (physical) processes/hosts, which we generically term the *layout of the application*, is undesirable and likely to impact its scalability. Moreover, design-time layout implies that layout changes require application reengineering.

At first glance it might look appealing, then, to provide a *dynamic application layout* capability and defer all layout decisions to runtime. However, ignoring altogether distribution layout considerations at design time is very problematic too. As pointed out in [24], this *implicit* approach neglects the fundamental differences between local and distributed computing. If ignored, programmers are likely to encode unreliable applications since high latency and partial failure cannot be treated robustly when the physical local-remote split is unknown. Performance is also likely to be affected, since co-location related optimizations cannot be made. In this sense, global scope even reinforces these arguments. Finally, the lack of shared memory in remote invocations implies that it is highly undesirable to fully mask distribution at design time, since parameters cannot be passed by-reference across address spaces. Providing such transparency (via distributed shared memory, or by making all language objects "remote" and passing always remote references) is not scalable to wide-are deployment. Thus, overlooking the difference between local and remote invocation semantics, while tempting, is misleading and is likely to cause many programming errors. Indeed, most recent distributed frameworks (e.g., CORBA [16] and RMI [21]) follow the (static) *explicit* approach, requiring different interfaces for local and remote invocations, and even require different syntax that forces programmers to be aware of the local-remote split in their applications. For example, remote objects in CORBA must be defined in a special Interface Definition Language, and in RMI they must inherit a mandatory interface and throw/catch remote exceptions.

The implicit approach does have one clear advantage, however. By masking the location of objects, programmers can use the same programming model for local and remote objects. This eases the construction of large distributed applications since programmers can focus on the logic of the application without being concerned with the distributed aspects. In other words, the implicit approach improves *programming scalability.*

A major challenge is then to design a distributed programming model that provides a dynamic layout capability without compromising on explicit programmability of the layout (thereby improving system scalability) and yet retains as much as possible the local programming language model (thereby improving programming scalability). FarGo attempts to reconcile these seemingly conflicting goals.

FarGo is an extension of Java. At the basic system level it provides extensive dynamic layout capability, including arbitrary component mobility, attachment of remote components into the same address space and detachment of co-located components into different address spaces. During these activities, all references between components remain valid. Such dynamic mapping enables to adapt applications to external changes in the environment, including changes in network bandwidth, machine loads, and partial network or node failure (or addition of new nodes). On top of the system level, FarGo provides a programming model layer that supports the specification of various co-location relationships between components, as in the explicit approach, except: (1) the relative physical structure between components (in terms of co-locality) may be automatically retained despite possible relocation of (parts of) the application, and (2) both the structure and re-location patterns may evolve at runtime to address the dynamic variability in the environment. Thus, co-location relationships drive and constrain the mapping process. Relocation, co-location relationships, and evolution, may be either programmed early at design time (as a separate part of the application) or after deployment using an external scripting facility. In order to enable layout programmers to base their re- and co-location decisions on runtime information, FarGo provides a monitoring service that enables applications to register for, and get notified about, system events. Finally, we retain as much as possible the local programming model of Java for encoding the logic of the application, as in the "implicit" approach. This is only possible because of the clear separation between the programming of the application's logic and the programming of the layout. It is this principle that permits to change the layout on-the-fly without changing the code of the application. FarGo is implemented and available for download and experimentation in `http://www.dsg.technion.ac.il/fargo`.

The rest of the paper is organized as follows: Section 2 presents the programming model and its main entities: *complets*, the relocatable application building blocks, and *complet references*, FarGo's main abstraction for dynamic layout programming. Sections 3 overviews the monitoring service along with an event-based scripting language for external programming of the layout. Section 4 discusses some implementation issues. Section 5 compares our work to other systems, and Section 6 summarizes our contributions and points to future work.

2 THE PROGRAMMING MODEL

In general, a FarGo application is comprised of a set of *complets*. Complets are the basic building blocks of the application, somewhat analogous to modules in a conventional programming language, except that they also define the minimal unit of relocation. That is, a complet instance relocates in its entirety.[1] Complets are interconnected via inter-complet references, henceforth termed *complet references*. Unlike remote references in conventional distributed frameworks (e.g., RMI [21]), the same complet reference may be at times local and at times remote, depending on the (dynamic) relocation of its source or target complets during the lifetime of the application. Unlike virtual references in other mobile frameworks which mostly provide (re)location transparency (e.g., Voyager [17]), complet references can be associated with rich semantics that describe various co- and re-location relationships between complets, as described below. Furthermore, these relationships are reified by the reference, and thus can be interrogated and evolve over time, e.g., to adhere to changes in the environment that demand changes in the relationships. Thus, complet references are a major abstraction mechanism for layout programming in FarGo.

Relocation support is facilitated by a light runtime infrastructure, consisting of a set of distributed Core objects, also termed site objects (we will use these

[1] Unless otherwise specified, we will refer to complet instances simply as complets, for brevity.

terms interchangeably throughout the paper). Cores are uniquely-identifiable objects that provide various system support for mobilizing and interconnecting complets across machines. However, except for special services that require direct interaction with the Core (e.g., reflection), most Core services are transparent to the application programmer since they are abstracted via complet references and a high-level monitoring API. Each complet is associated with exactly one Core at any given time, but a complet may relocate to a different Core during its execution while preserving its state, including its external outgoing and incoming (complet) references. We now turn to discuss complets and their references in detail. Core internals are mostly beyond the scope of this paper, see [9].

Complets

A fundamental issue in dynamic layout support is the granularity of the minimal relocatable entity. The fine-grained approach supports relocation for every programming language object. Since mobility affects all objects, this approach is typically implemented by introducing a special programming language, such as Distributed Oz [6]. Although flexible, this approach has several major drawbacks. Firstly, well engineered applications are typically partitioned into modular units, each containing several closely related and highly cohesive objects. But the fine-grained approach requires mobility support to be built into each and every object, thereby introducing unneeded overhead. Secondly, Enforcing *all* programming language objects to adhere to a remote-call semantics that assumes the lack of shared memory, adds unnecessary complexity to programming. Finally, requiring programmers to use a new programming language is also problematic.

The coarse-grained approach, in contrast, divides the application into a set of processes, each of which is assigned its own distinct address space, and can move between hosts during its execution. Examples of this approach include Telescript [25] and AgentTCL [7]. An application designer must partition the application into a number of processes, and populate each process with a (possibly large) number of objects. While this approach clearly separates local and remote invocations, its drawback is that if two or more processes happen to be co-located, they cannot take advantage of the available shared memory. For example, they cannot pass arguments by reference. This is particularly important in cases where processes are intentionally relocated at runtime to a shared host for the purpose of improving their interaction. Thus, placing rigid process boundaries on the unit of relocation seriously restricts the power of dynamic relocation.

We take a mid-grained approach, in which the unit of relocation — the complet — is a collection of local objects,[2] i.e., they share an address space and all references among these objects are regular (local) references. Each complet may be either co-located with, or remote to, other complets, and as a complet moves, it dynamically changes its locality relationship with other complets. Each complet has a single object, termed *anchor*, whose interface is the interface of the complet.

Given the anchor a of a complet α, the set of objects that comprise the complet, denoted as $closure(\alpha)$, is defined recursively as follows:

- $a \in closure(\alpha)$
- If $b \in closure(\alpha)$ then $\forall c$ such that b references c (denoted as $b \rightarrow c$), if $c \neq anchor(\beta)$, $(\beta \neq \alpha)$, then $c \in closure(\alpha)$

That is, a complet closure is defined by the reachability graph of objects and references, starting from the anchor, except for references that point to other anchor objects, which are termed complet references.

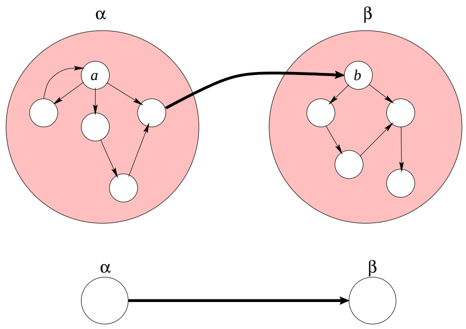

Figure 1: Complet Structure

Figure 1 shows two complets, α and β, whose anchors are a and b respectively. All objects inside the left circle comprise $closure(\alpha)$ while all those inside the right circle comprise $closure(\beta)$. The bold arrowhead line is a complet reference from one of the objects of α to the anchor of β, denoted as $\alpha \Rightarrow \beta$. Relationships between complets are defined by the set of complet-references, as shown in the bottom of the figure.

The use of anchors provides a clean interface to complets, but it also facilitates syntactic transparency between regular and complet references. Both are encoded similarly, but complet references are treated differently at runtime. Syntactic transparency helps to preserve the conceptual separation between programming the logic of the application and programming of its layout.

Semantically, intra- and inter-complet references must differ at least because of relocatability and cross

[2]In principle these could be objects of any object-oriented language but we use Java syntax throughout the paper since FarGo supports Java.

address-space operation. One basic difference is in passing arguments in method invocations along references. Inside a complet all objects are passed by (local) reference. Across complets (i.e., along complet references), the following rules apply:

- regular (non-anchor) objects are passed by value.
- anchor object are passed by (complet) reference.

The first rule allows the caller and the callee to operate across address spaces (since arguments are copied over), and it also ensures that non-anchor objects cannot be referenced from a different complet, consistent with the definition of complet closure. The second rule prevents from automatically copying recursively all other (possibly remote) complets that happen to be pointed to by an object that is passed by value.

Notice that the programmer must be aware of the distinction between intra- and inter-complet interaction, consistent with our general approach to distributed programming. At the same time, it is important to note that when the source and target objects of a complet reference happen to be co-located, arguments are still passed by value for consistency with the model, but the complet reference is turned into a local reference, thereby improving performance and reliability.

Complet References

In addition to the default semantics provided by a complet reference — a pointer that remains valid despite its source or target migration — FarGo provides means to control the layout of the application by associating co- and re-location-related semantics with such references.

Relocation semantics are provided by an extensible hierarchy of reference types, which collectively define the basic layout programming interface. Since references can evolve dynamically, each reference type has additional semantics that describe relocation activity that occurs as a result of a change in type. Finally, we define a set of space and time modifiers that apply to all types, allowing to generalize the notion of spatial and temporal co-locality. Thus, in the general case co-locality is a relative term: it may mean the same host, same LAN, etc. But we begin the discussion with an abstract notion of co-locality and make it concrete later on.

Specifically, we provide five basic types of complet references. This list is by no means exhaustive, however, and we expect to enlarge the repository of basic types as practical situations arise. For each type we define: (1) the basic semantics, consisting of co-location relationship and how it is affected by relocation; (2) the semantics when a type is dynamically changed; and (3) in what situations the type is useful. We conclude the section with an example application that exhibits the various types.

1. **Link**(α,β) (denoted as $\alpha \overset{link}{\Rightarrow} \beta$) — This is the default basic complet reference from α to β, with no constraints (and thus no special) co/re location semantics:

- Co-location: α and β may or may not be co-located.
- Re-location: relocation of α does not affect the location of β, and vice-versa.

a Link reference may be used when the source and the target complets need not have any special relationship with respect to relocation. It does provide, however, location transparency, allowing to move both source and target complets while keeping the validity of references. This reference is used when no layout programming has been performed.

2. **Pull**(α,β) (denoted as $\alpha \overset{pull}{\Rightarrow} \beta$) — This reference is used to ensure co-location between complets; when the source complet relocates, it "pulls" its target along. More precisely:

- Co-location: α and β are co-located.
- Re-location: if α relocates, β moves along to the locality of α.

This definition implies several additional constraints. First, β cannot be relocated on its own (i.e., not as a result of α's relocation), since this would violate the first condition. Second, β can be referenced by at most one Pull reference. Otherwise, if $\gamma \overset{pull}{\Rightarrow} \beta$ is also allowed, then upon movement of α one of the two above conditions is not fulfilled: if β moves with α it is not co-local with γ, and if it does not move it is not co-local with α. Notice however, that α can still be a target of a Pull reference, i.e., $\delta \overset{pull}{\Rightarrow} \alpha$ is possible. Pull is transitive, and relocation of δ will relocate α, and hence β.

Both automation and enforcement of these semantics are performed by the runtime infrastructure (this is true for all reference types). That is, the target complet implicitly moves along with the source, and upon an attempt to create a second Pull reference to a complet, or upon an attempt to move a complet which is a target of a Pull reference, an exception is raised and the operation is canceled. Finally, we have to discuss what happens when an existing (non Pull) reference is evolved into a Pull reference. In order to fulfill the first condition, such a change leads to automatic movement of β to the locality of α, if it is not already local.

A Pull reference is useful when α and β need to interact frequently and/or require heavy data-transfer on each interaction, yet they cannot be programmed inside a single complet (e.g., because β needs to be referenced by other complets, or because their coupling is needed temporarily). In these cases, arbitrary physical dispersion of α and β might lead to poor performance and reduced reliability, or even to halt the application.

3. **Duplicate**(α,β) (denoted as $\alpha \stackrel{dup}{\Rightarrow} \beta$) — This reference is similar to Pull, except it is sufficient to ensure co- and re-location with a copy of β, instead of with β itself, where a copy of a complet is naturally defined as a copy of its closure (including the anchor).

- Co-location: α and a copy of β are co-located.
- Re-location: if α relocates, a copy of β moves along to the locality of α.

Unlike Pull references, there can be multiple Duplicate references to the same complet. This is possible since if $\gamma \stackrel{dup}{\Rightarrow} \beta$ exists and α moves, it takes along a copy of β, and γ continues to be co-local with β. Also, (a copy of) the target complet can be relocated, since it leaves behind the original copy, thus all references still point to it. Thus, a complet may be simultaneously the target of a (single) Pull reference and the target of multiple Duplicate references. This is a desirable property, since it does not over-restricts Pull references by constraining their compatibility with non-Pull references. When a non-Duplicate reference is evolved into a Duplicate reference, then again, a copy of the target complet (not the target complet itself) is created and sent to the source complet. This ensures that semantics of other references to the original target (e.g., Pull) are preserved.

Duplicate references are useful when the target complet represents a read-only entity that can be easily replicated without violating the logical semantics of the application. In this case, the replication may speed-up performance and improve reliability by decreasing the amount of network messages, with no extra effort by the programmer. All s/he has to do is set the reference as Duplicate, and the rest is provided by the runtime.

4. **Stamp**(α,β) (denoted as $\alpha \stackrel{stamp}{\Rightarrow} \beta$) — This reference is similar to Duplicate, but instead of creating a copy at β's locality and passing it along with α, α locates a local instance of β's type and connects to it:

- Co-location: α and some instance of β's type are co-located.
- Re-location: if α relocates, an instance of β's type is located in α's new locality and gets attached to it.

The implications on the target complet with respect to relocation and compatibility with other references, are similar to those of Duplicate references. An instance of β's type is free to move regardless of α (which remains connected to the original β), and multiple Stamp and Duplicate references can co-exist along with a single Pull.

By "a local instance of β's type" we refer to type equivalence, i.e., a local complet instance that implements the interface (type) of β, although not necessarily an instance of β's implementation (class). In addition to explicit type equivalence test, FarGo allows to define name equivalence (each complet can bind itself to a name at a given core, and the equivalent target can be defined as bound to the same name as the original), or any user-defined equivalence procedure. Finally, upon evolution of a reference into a Stamp type, an instance of β's type is located in α's locality and gets reconnected to α.

A useful application of a Stamp reference is for facilitating a constant connection from a mobile complet to a non-mobile complet. For example, if the target complet encapsulates a hardware device (e.g., a printer), a Stamp reference could be used to reconnect the source complet to a local printer after it arrives to a new location.

5. **Bi-directional Pull**(α,β) (denoted as $\alpha \stackrel{bpull}{\Rightarrow} \beta$) — This is the most powerful (but most expensive) relocation reference. It is similar to Pull, with the additional rule that the target can Pull the source too.

- Co-location: α and β are co-located.
- Re-location: if α relocates, β moves along to the locality of α, *and vice versa*.

This implies a restriction similar to that of the Pull reference — β cannot be referenced by any Pull reference. By definition, this reference enables the target to move not only as a result of source relocation, so it is less restrictive than the (uni-directional) Pull. This in turn allows to remove the singleness constraint of Pull too, i.e., a complet can be the target of multiple BPull references. But the most interesting property of BPull is that it effectively defines "group re-location" semantics. The group is defined as the sub-graph of all nodes that are connected via BPull references, and relocation of *any* member of the group leads to relocation of *all* members of the group. In case of evolution into a BPull reference, we arbitrarily determine that the target relocates to the locality of the source.

One issue with BPull references is that they ignore the natural direction of the reference, from the source to the target. Also, realization of this reference requires a (hidden) back-reference from the target to each of its sources, in order to track relocation of the source complets. In certain applications, this overhead may be intolerable. For example, a client-server application with a very large number of BPull clients would require the server to know who holds a reference to it, which is not scalable. It would also mean that each client relocation forces the server and all other clients to relocate, which is typically not reasonable. This in fact suggests yet another useful reference type — Inverted Pull — in which only the target can relocate, and when it does, all source references follow. Although we haven't implemented Inverted Pull references, one can see how this type hierarchy can be enriched incrementally as the need arises.

Space, Time and other Reference Attributes

So far, we have left out precise definition of co-locality and the timing of relocation (i.e., when relocation occurs). The default values, as may be expected, are same address-space for co-locality, and "immediately" for time (i.e., with no delay). However, in many cases it is worthwhile to generalize these notions. For example, an Internet-based video application might require certain minimum bandwidth that can be provided by a LAN connection, but not by a WAN or a dial-up connection. Or, complets may be allowed to operate within an Intranet but not across a firewall, and so forth. In such cases, it is useful to extend the notion of co-locality by defining a distance metric, and add it as an optional parameter to a reference. For example, $\mathrm{Pull}(\alpha, \beta, d)$ means that α and β should operate within distance d, where d is interpreted according to the implemented metric.

A second useful attribute which is also orthogonal to reference types is a delay factor. Consider the network-computing model, for example, by which a client downloads a "thin" version of an application that has only the basic features, and the advanced (and rarely used) features are stayed in the server. However, if the client does want to use any of the advanced features, then upon the first reference to this feature, it gets lazily downloaded. This may be termed *Delayed Pull*. Alternatively, the user may be able to execute this feature remotely (although more expensively), and only when s/he invokes it more than a certain number of times, it gets downloaded. In general, a delay metric can be formed with a delay factor attached to every reference.

Finally, as with relocation types, these reference modifiers are two of many other possible attributes. For example, another useful attribute is time-expiration, after which a reference becomes invalid (e.g., to be used in electronic-commerce applications).

The TODO Application

Let us illustrate the programming model by presenting a simple multi-user tool for management of a project's todo-list. The todo-list can be examined and updated concurrently by a number of users from different locations. Shown in Figure 2, TODO consists of five kinds of complets. The User Interface complet is used for viewing and updating the todo-list. The Engine complet receives updates from users and multicasts each update to all currently connected users. The Engine can move among a number of hosts to be as close as possible to most of its clients (e.g., reside in different continents at different times of the day). The todo-list is persistently saved by the File complet, which receives a location on the local file system (a pathname) from the stable Disk complet. The Startup complet is used as the point of contact for joining users. Upon a request to connect, it instantiates a User Interface complet and moves it to the requester's site. Below is a (simplified) portion of the user interface's code:

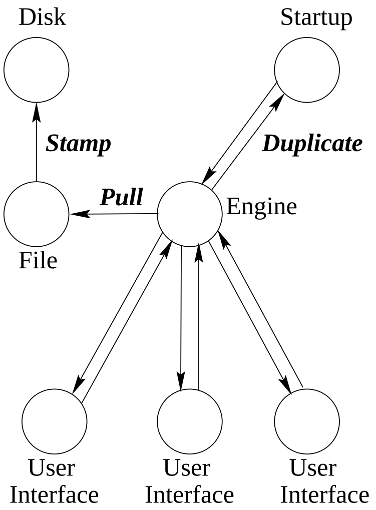

Figure 2: The TODO Application

```
public class UserInterface_
                        implements Complet {
 private Engine engine;
 UserInterface_(Engine engine) {
  this.engine = engine;
 }
 void addTask(String newTask) {
  engine.addTask(newTask);
 }
 // ...
}
```

Notice how this code is similar to regular Java. For example, the `Engine` complet instance is held as a regular data field of the `UserInterface` complet and is invoked with regular Java syntax, eventhough both complets may dynamically relocate. The only noticeable difference is an extra underscore character, explained later in Section 4.

The complet references in Figure 2 that are not labeled are Link references. Since the Engine interacts with the File complet on every update of the todo-list, pulling it along on each movement is desirable. Thus, it points to the File complet with a Pull reference. Upon arrival to a new site, the File must contact the local Disk complet to receive a new location on the local file system, thus it points at the (stationary) Disk complet with a Stamp reference. Finally, to offer many points of contact to the application, each new location that the Engine visits should become a possible target for a request to join the group of users. Thus, the Engine points at the Startup complet with a Duplicate reference. The Engine sets its reference to the Startup complet in its constructor as follows:

```
public Engine_(Startup startup) {
 this.startup = startup;
 CompletRef cr = Core.getCompletRef(startup);
 cr.setRelocator(new Duplicate());
 // ...
}
```

Notice again the separation of concerns. The use of this reference for method invocation in the rest of Engine's code (the application logic) is not affected by the manipulation of its relocation attribute (the layout semantics).

3 LAYOUT PROGRAMMING USING MONITORING INFORMATION

Layout programming in FarGo consists of three layers: a layout API that enables embedding of dynamic layout algorithms within applications, a high-level script language for attaching layout scripts to applications at runtime, and a graphical tool for layout management.

Layout policy is specified in an event-based style, which involves registration for event notifications that are generated by the Core, and specification of callback procedures that should be executed upon event notifications. The Core continuously performs a set of performance and resource utilization measurements which are examined both by the Core itself, to determine when to fire events, and by the callback procedures, to determine what action to take upon the occurrence of an event.

Both events and measurements are divided into three groups: those that act on a whole Core (e.g., *Core Shutdown* and *Complet Count Limit*), on a single complet (e.g., *Complet Departure* and *Complet Arrival*), and on a single complet reference (e.g., bandwidth, and average number of invocations per time unit).

The layout API provides means to register to events and to perform the various measurements. This API is based on a simple distributed extension of the standard Java event model [22]. Unlike the distributed event model in Sun's Jini Technology [23], our extension is tailored especially for layout programming, thus more simple and efficient.

Scripting Language

In addition to the API, FarGo provides a high-level scripting language external to the application. Scripts are written in an Event-Action style. A script consists of a set of rules of the form:

`on` *event* [`at` *core*] `do` *actions*

An *event* is specified by its name and a set of attributes. For example, `completArrival(complet, sourceCore)` designates the arrival of a certain complet from a certain location. *core* specifies the Core on which the event is expected to occur (the default is the Core on which the script is running), and *actions* are one or more statements, each is either a built-in primitive (e.g. `move`) or means for interfacing with Java methods in which more sophisticated policies can be specified.

Let us revisit the TODO example presented earlier and show how a script that relocates the Engine complet to its optimal location may be encoded:

```
Script EngineScript {
 on completDeparture(complet, target) do {
  engine = thisComplet();
  userInterfaces += complet;
  best = Locator.findBest(userInterfaces);
  move(engine, best);
 }
 on coreShutdown(coreName) do {
  moveAll("zeus.technion.ac.il");
 }
}
```

The first rule catches the `completDeparture` event on the local core. This event is fired when a new User Interface complet is about to move from the local core to a new user. The action block of the rule adds the complet to the user interfaces list, then invokes a Java method that finds the best location for the En-

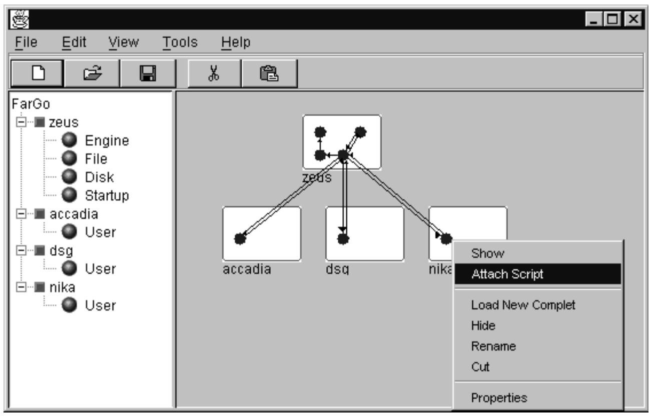

Figure 3: The Graphical Monitor

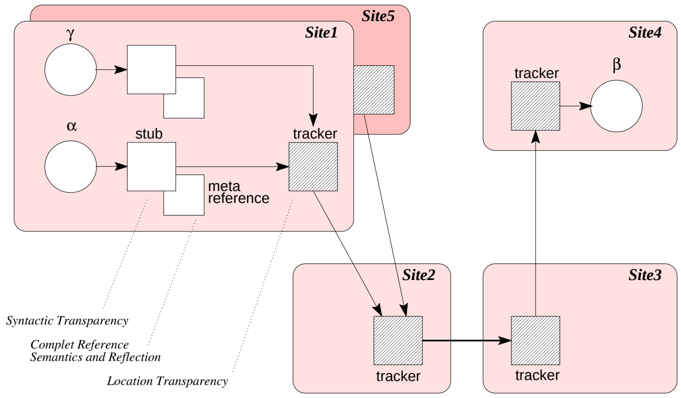

Figure 4: Complet Reference's Implementation

gine complet, and finally moves the engine to that core. The second rule implies that upon shutdown of the local core all complets will relocate to a machine named `zeus.technion.ac.il`.

The FarGo graphical monitor visualizes the state of a FarGo application. Figure 3 shows a visualization of TODO. This tool is used by an administrator to control and manipulate both running applications and Core environments. Complet references can be examined and changed by simple point-and-click operations, and complets can be moved between cores using drag-and-drop operations.

Monitoring support in FarGo is implemented using FarGo's own facilities. The entity to which scripts register is a special stationary event-notifier complet that resides in each Core that is willing to be managed. This complet uses the basic layout API to contact its local Core. Each script is held and executed by a special script complet. Passing events to a script is implemented by using complet references for invoking event notification methods, which enable the scripts to relocate with the complets they manage and still listen to events that occur on remote cores.

4 IMPLEMENTATION

In this section we briefly overview the implementation of FarGo. A detailed discussion is given in [9].

Complet References

The internal structure of a complet reference $\alpha \Rightarrow \beta$ is shown in Figure 4. The source, α, holds a Java reference to a *stub* object, that has the same method and constructor interface as that of β's anchor. The stub provides syntactic transparency in that, syntactically, invocations of its methods are identical to direct invocations of β's anchor.

The stub contains an object, termed *meta reference* that reifies the complet reference and allows to dynamically change it. The meta reference of a given complete reference can be fetched using the Core's method `getCompletRef`. The meta reference can be used to examine the semantics of the complet reference by invoking its `getRelocator` method, and to change the semantics by invoking the `setRelocator` method. Other properties that can be examined include complet reference equivalence (two references are equivalent if they are both pointing to the same anchor), and co-locality with another complet.

The stub holds a *tracker* object that functions as the means for maintaining the complet reference's validity despite β's location changes. If β is local, the tracker points to its anchor directly with a regular Java reference. Complet movement results in creation of a chain of trackers (as in [5, 19]), where each tracker points at its successor on a different site (inter-site pointing is implemented using RMI). For example, In Figure 4, the complet reference $\alpha \Rightarrow \beta$ embodies a chain of four trackers. Possible creation of redundant cyclic chains, due to complet's cyclic migration, is detected and prevented by the Core. Chains are shortened automatically whenever the source's Core interacts with the target's Core (e.g., on every method invocation), by setting each tracker along the chain to directly point to the chain's tail. After shortening, each tracker that is no longer pointed to (Site3 in the Figure) becomes available for distributed garbage collection. Each site holds at most one tracker per referenced complet, which is shared among all the complet references of that site. In Figure 4, for example, the two complet references $\alpha \Rightarrow \beta$ and $\gamma \Rightarrow \beta$ share the same tracker on Site1.

Invocation of complet methods is being relayed in the following way. The source invokes the stub, which invokes the tracker, which in turn either locally invokes the target's anchor, or forwards the invocation through a chain, whose tail invokes the (remote) anchor. The meta reference is a system object that does not include any application-specific code. The stub and tracker classes are generated by a FarGo stub compiler, with methods and constructor signatures that are identical to those of the anchor.

Complet Design Issues

As seen in code samples given in Section 2, an anchor class must end with a special marker (an underscore character) and must implement the empty `Complet` interface. The marker is needed simply to distinguish between the name of the stub class and the name of the anchor class. By implementing a special interface, all Core mechanisms that should distinguish between anchors and other objects (e.g., the parameter passing and movement mechanisms) can identify an anchor efficiently using Java's `instanceof` operator.

Our approach in that matter differs from that of Voyager [17], where the mobile object's class is free from interface inheritance requirement in order to support seamless conversion of any existing class to a mobile class. Following the approach of explicit distributed semantics advocated in Section 1, we claim that such automatic conversion might break the (local) semantics by which the existing class was designed, thus lead to programming errors. For cases where the semantics of the existing class is known and not problematic, FarGo (non-seamlessly) supports such conversion by wrapping the non-mobile class with a delegating anchor, which can be automatically built by the compiler upon request.

Unlike most other mobile frameworks, FarGo poses almost no restrictions on the usage of the anchor from within its closure. For example, it can pass itself (using Java's `this`) as a parameter to a method of a different complet (or a Core service). The parameter-passing mechanism automatically detects such cases and replaces the Java reference to the anchor with a complet reference (further implementation details are beyond the scope of this paper, see [9]).

Another syntactic transparency is provided with complet instantiation. Although it is very different from instantiating a regular Java object, it is identical from the programmer's viewpoint, using the ordinary `new` operator (unlike instantiation in Voyager [17], or Aglets [13]). Internally, the system objects that comprise the complet reference are constructed from the stub's constructor. In addition to local complet instantiation, FarGo supports remote complet instantiation using the Core's `remoteNew` method, which returns a reference to a complet after instantiating it on the remote site.

Status
Most of the FarGo system as described in this paper has been implemented, including a full implementation of the TODO application (which is intended to aid in our own development work). Monitor support is currently available only through the monitoring API; the scripting language and graphical monitor are currently under development. Several additional utilities have been implemented, including a Core command-line shell (itself a FarGo application) for remote management and debugging of Cores and a stub compiler compiler. The system's codebase consists of approximately 40,000 code lines and the Core's binary footprint is about 260KB.

5 RELATED WORK

The three most widely available environments for distributed computing are CORBA [16], DCOM [4], and Java RMI [21]. The programming model presented by all these environments is based on an object oriented extension of the traditional RPC [3]. FarGo's parameter passing semantics is closest to that of RMI (in fact FarGo uses RMI as part of its implementation). The anchor resembles RMI's implementation of a remote interface and the complet's stub resembles RMI's stub. However, neither RMI nor CORBA or DCOM currently support dynamic relocation of distributed objects, let alone means to program the layout or monitor an application's behavior.(Effort in this direction has started lately, see CORBA MAF [15].)

Much research has been conducted in the field of mobile objects and agents in recent years. The language-based approach taken, for example, in Telescript [25], in the mobile agent extension of (the ML-based) Facile [12], and in Distributed Oz [6] suggests a new programming language that features object mobility. Other environments take a system-based approach where an existing language is used as is, along with a set of libraries and runtime support. Representatives are the Java based systems Aglets [13], Sumatra [1], and Voyager [17], and the multi-lingual systems D'Agents (formerly AgentTCL) [7] and Tacoma [10]. These environments also differ in their type of mobility. Strong mobility (as in Telescript) involves movement of a full program's runtime context, including the stack and program counter. Weak mobility (as in Aglets) involves only movement of object's code and state. FarGo belongs to the system-based Java-only environments. Being Java-based implies weak mobility due to the use of a standard virtual machine that does not expose a program's full runtime context (strong mobility is achieved in Sumatra at the cost of a non-portable implementation).

The most essential and unique characteristic of FarGo is its extensive support for *programming the dynamic layout* separately from the application's logic. Like FarGo, the above systems do support mobility, but in a model that tightly couples movement operations to the application's logic. This reflects a major difference between most of these systems and FarGo that stems from their focus on agents — *autonomous* entities that move themselves as part of their computation, versus FarGo's focus on general widely-distributed applications, not necessarily autonomous-agents-based. As a result, unlike most of the above environments, FarGo's programming model is very close to Java's own model, which facilitates programming scalability. In Aglets, for example, inter-agent communication is done using a special event model, not by regular Java invocations, which is not natural for a Java programmer. An agent class must be a subclass of a standard `Aglet` class, which limits the designer since Java permits only single inheritance. Agent instantiation both in Aglets and in Voyager is done using a special procedure, not by invoking a regular object constructor as in FarGo.

Another unique aspect of FarGo is *how* dynamic layout is integrated with the overall architecture of the application. All the above environments only provide movement primitives and leave all the rest to the programmer. FarGo, on the other hand, introduces the notion of references that may occupy sophisticated relocation semantics and are manipulated using a reflective mechanism. From a software architecture perspective, these references function as architectural connectors (as proposed in UniCon [20]), which explicitly specify the architectural glue between components with respect to their (re)location.

An additional dimension of support for dynamic layout programming is enabled with the monitoring information supplied by the Core. Of all the above systems, only Sumatra employs such support, but using a drastically different programming model and API, which tightly couples relocation into the application's logic. External attachment of layout policies to a live application, which is not supported in Sumatra, further promotes decoupling between the two. Using a high-level scripting language as means for monitoring-based layout programming, adds another dimension of dynamicity.

Playground [11], Darwin [14], Polylith [18] and Hadas [2, 8] are component-coordination environments for development of distributed applications. They are all centered on separating component interconnections (composition) from their individual behavior, and some (e.g., Darwin, Hadas) allow dynamic-reconfiguration of the connections between components. This architectural principle is also incorporated in FarGo's complet references and layout scripts. However, none of these systems supports full mobility of deployed components, and thus the layout programming layer is not even applicable to them. Hadas, FarGo's predecessor, provides a limited form of mobility via Ambassadors, which are dynamically adaptable and deployable component stubs, but general relocation is not programmable.

6 CONCLUSIONS AND FUTURE WORK

This paper proposes a new dimension of flexibility for the architects of large-scale distributed systems — the ability to program dynamic layout policies separately from the application's logic. We have developed a programming model that carefully balances between programming scalability and system scalability, and which uses the inter-component reference as its main abstraction vehicle. We have also presented a monitoring facility to assist in making relocation decisions, and an event-based scripting language to encode layout policies.

Future directions include the design of a global, yet scalable, complet naming and location service, and also persistence and security mechanisms. With respect to the programming model, we intend to use the mechanisms described in this paper to develop "runtime patterns" (the runtime complement of design patterns), which will offer a taxonomy and optimized system support for commonly used layout policies that could be easily assembled by applications to achieve sophisticated functionality and a high level of scalability and adaptability.

REFERENCES

[1] A. Acharya, M. Ranganathan, and J. Saltz. Sumatra: A language for resource-aware mobile programs. In *Mobile Object Systems: Towards the Programmable Internet*, pages 111–130. Springer-Verlag, April 1997. Lecture Notes in Computer Science No. 1222.

[2] I. Ben-Shaul, A. Cohen, O. Holder, and B. Lavva. HADAS: A network-centric system for interoperability programming. *International Journal of Cooperative Information Systems (IJCIS)*, 6(3&4):293–314, 1997.

[3] A. Birell and B. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984.

[4] N. Brown and C. Kindel. *Distributed Component Object Model Protocol — DCOM/1.0. Internet Draft*, January 1998. Available at `http://www.microsoft.com/oledev/olecom/draft-brown-dcom-v1-spec-02.txt`.

[5] S. J. Caughey, G. D. Parrington, and S. K. Shrivastava. Shadows — A flexible support system for objects in distributed systems. In *Proceedings of the Third International Workshop on Object Orientation and Operating Systems*, pages 73–82, Asheville, NC (USA), December 1993.

[6] P. V. R. et al. Mobile objects in Distributed Oz. *ACM Transactions on Programming Languages and Systems*, 19(5):804–851, September 1997.

[7] R. S. Gray. Agent TCL: A transportable agent system. In *Proceedings of the CIKM Workshop on Intelligent Information Agents, Fourth International Conference on Information and Knowledge Management (CIKM 95)*, Baltimore, Maryland, December 1995.

[8] O. Holder and I. Ben-Shaul. A reflective model for mobile software objects. In *Proceedings of the 17th International Conference on Distributed Computing Systems (ICDCS'97)*, pages 339–346, Baltimore, Maryland, May 1997. IEEE Computer Society Press.

[9] O. Holder, I. Ben-Shaul, and H. Gazit. System support for dynamic layout of distributed applications. Technical Report EE Pub No. 1191, Technion — Israel Institute of Technology, October 1998.

[10] D. Johansen, R. van Renesse, and F. B. Schneider. Operating system support for mobile agents. In *Proceedings of the 5th IEEE Workshop on Hot Topics in Operating Systems*, Orcas Island, WA, USA, May 1995. IEEE Computer Society Press.

[11] K. Goldman et al. The programmer's playground: I/O abstraction for user-configurable distributed applications. *IEEE Transactions on Software Engineering*, 21(9):735–746, September 1995.

[12] F. C. Knabe. *Language Support for Mobile Agents*. PhD thesis, Carnegie Mellon University, December 1995. Technical Report CMU-CS-95-223.

[13] D. B. Lange and D. T. Chang. IBM Aglets Workbench: Programming mobile agents in Java. A white paper. Technical report, IBM, Tokyo Research Lab, September 1996. Available at `http://www.trl.ibm.co.jp/aglets/whitepaper.htm`.

[14] J. Magee, N. Dulay, and J. Kramer. Structuring parallel and distributed programs. *Software Engineering Journal*, 8(2):73–82, March 1993.

[15] Object Management Group. *Mobile Agent Facility Specification*, June 1997.

[16] Object Management Group. *The Common Object Request Broker: Architecture and Specification. Revision 2.2*, February 1998. Available at: `http://www.omg.org/corba/corbaiiop.htm`.

[17] ObjectSpace Voyager core package: Technical overview, December 1997. Available at: `http://www.objectspace.com/voyager/whitepapers/VoyagerTechOview.pdf`.

[18] J. Purtilo. The POLYLITH software bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[19] M. Shapiro, P. Dickman, and D. Plainfosse. SSP chains: Robust, distributed references supporting acyclic garbage collection. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, Vancouver, August 1992. ACM.

[20] M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and implementations for architectural connections. In *Proceedings of the Third International Conference on Configurable Distributed Systems*, May 1996.

[21] Sun Microsystems, Inc. *Java Remote Method Invocation (RMI) Specification*, December 1997. Available at: `http://java.sun.com/products/jdk/1.2/docs/guide/rmi/spec/rmiTOC.doc.html`.

[22] Sun Microsystems, Inc. *JavaBeans*, July 1997. Available at : `http://java.sun.com/beans/docs/spec.html`.

[23] J. Waldo. *Distributed Event Specification*, July 1998. Available at : `http://java.sun.com/products/jini/specs/`.

[24] J. Waldo, G. Wyant, A. Wollrath, and S. Kendall. A note on distributed computing. Technical Report TR-94-29, Sun Microsystems Laboratories, Inc., November 1994.

[25] J. E. White. Mobile agents make a network an open platform for third-party developers. *Computer*, 27(11):89–90, November 1994.