System Support for Dynamic Layout of Distributed Applications

Ophir Holder Israel Ben-Shaul Hovav Gazit

Department of Electrical Engineering
Technion — Israel Institute of Technology
Technion City, Haifa 32000, Israel
{holder@tx,issy@ee,ghovav@tx}.technion.ac.il

Abstract

Dynamic application layout is the capability to move the components of a distributed program among different hosts during the execution of the application. This capability is essential for large-scale applications since it allows to adapt them to changes in resource availability, which are common in wide-area environments. The FarGo system introduces a model for programming the layout of distributed applications separately from their basic logic, by attaching relocation semantics to inter-component references, and by using a built-in monitoring support for making relocation decisions. Dynamic layout policies are encoded within the application using a special API or externally using a high-level scripting language. This paper presents the design of the runtime environment that realizes the model.

Keywords

Distributed Components, Distributed Programming Models, Mobile Objects, Java, Engineering Distributed Systems.

1. Introduction

Wide area computing allows distributed applications to be deployed over a large number of interconnected nodes. The global deployment space is characterized by consisting of many nodes with different computing power and dynamically changing resources, and many links with widely different and dynamically changing transfer rates, reliability, and qualities of service.

These characteristics imply that it is highly undesirable to fix the local-remote relationships between components at design time, and equally undesirable to fix the location of these components at deployment time. Since the operating conditions are dynamically changing, and since the interaction between the application’s components may vary at different invocations of the application, static component layout might lead to low resource utilization, high network-latency and low reliability. In other words, it is impossible to set a priori the structure of the application in a way that best leverages the dynamically changing computing and networking resources.

Thus, the natural approach to address these issues is to provide a “dynamic layout” capability, which allows the location of components to be changed at runtime. Indeed, numerous solutions have been proposed in recent years along this direction, including AgentTCL [8], Aglets [12], Telescript [22] and Voyager [14]. Two necessary runtime mechanisms must exist in any framework that supports dynamic layout. The first is code mobility, which enables to move an active program fragment (along with its state) from one site to another during execution. The second is a tracking facility, which allows components to find and interact with each other despite their (possibly uncoordinated) movement, throughout the lifetime of the application. Both capabilities can be realized in different ways. For example, mobility is generally classified into weak mobility (as in Aglets), where only the object’s code and state are transferred, and strong mobility (as in Telescript), where the full program’s runtime context including the stack and program counter are moved. Tracking can also be realized in two major ways. One way is to continuously maintain a valid (virtual) reference to the moving target object (as in Voyager). The other way is to reconnect a reference to a moved object on-demand, using an external location and naming facility (as described, for example, in [20]). In either case, it is desirable to hide the implementation inside the runtime and provide location transparency.

On top of these basic enabling mechanisms, we argue that there are three additional requirements that are essential for realizing effective support for dynamic layout. First, there must be mechanisms that allow to make judicious decisions regarding component relocation. These include monitoring facilities that track the quality and availability of resources (e.g., the bandwidth between two remote co-
ponents) and the usage of these resources by the application (e.g., the invocation rate along a reference), along with means to asynchronously notify applications when certain threshold values are monitored.

Second, there should be means to specify co- and relocation constraints, as well as mechanisms to enforce and automate relocation of related components in order to comply with the specified constraints. For example, an application may specify that components \( \alpha \) and \( \beta \) are always colocated, e.g., because they interact frequently and need to share a large data source, or because partial failure (of \( \alpha \) or \( \beta \), but not both) cannot be tolerated. This means that when \( \alpha \) moves, \( \beta \) moves along to \( \alpha \)'s new site, and vice versa.

Third, the “local” programming language model should be preserved as much as possible. That is, except for aspects that are inherently different such as parameter passing along remote invocations (due to lack of shared memory), or aspects in which distribution should be explicit, such as explicit specification of co-location constraints and relocation policies, as noted above, everything else should resemble local programming. This includes the syntax, semantics, object model, event model, etc. There has been an on-going debate over the desired level of transparency that should be given to developers of distributed applications (see [21]), and the same arguments apply when extending the discussion to distribution with dynamic layout. The conflict is between programming scalability (easing the task of programming large and complex applications), which is achieved by such transparency, and system scalability (providing better reliability, performance, etc.), which is achieved by non-transparency.

The FarGo system reconciles this conflict by separating these concerns and providing a programming model with distinct programming interfaces for “local” component programming and for dynamic layout programming. The details of the programming model itself are given elsewhere [10] and are presented here only in a summary form, in Section 2. The main focus of this paper is on system support for realizing the programming model. Specifically, Section 3 addresses runtime support for layout programming and Section 4 addresses support for monitoring and relocation programming. Section 5 gives some implementation notes, Section 6 compares FarGo to other systems, and Section 7 summarizes our contributions and points to future work.

2. Programming Model Overview

The design and implementation of a FarGo application involves three facets (which correspond to the three requirements mentioned above, in reverse order). The first facet involves the design of the application logic by defining the components of the applications, which are termed complets, and their interactions. A complet is similar to a component in other component frameworks (e.g., Java Beans [18]). It is a collection of objects that perform a certain task, and is accessed through a well-defined interface object, termed the complet’s anchor. Thus, all external references into a complet point to the anchor, and the closure of the complet is defined as the directed graph of objects and references, starting from the anchor, except for references to other anchors (i.e., other complets).

In addition to their role as ordinary components, complets define the smallest unit of relocation. Thus, all objects within the same complet instance always share an address space and all intra-complet references are local, but complets may migrate during their lifetime. Therefore, inter-complet references (henceforth complet references) may at times be local and at times remote. However, in order to provide clean semantics for method invocation, complets are always considered remote to each other with respect to parameter passing. Thus, parameters are always passed by value along a complet reference, except for complet parameters, which are passed by (complet) reference. The complet programming model is very similar to plain Java. In particular, complets are instantiated, referenced, and relocated using regular Java syntax and semantics, and more importantly, following closely the Java object model. Realizing such transparency for relocatable complets introduces several technical challenges, which are addressed in Section 3.

The second facet involves the encoding of co- and relocation semantics, which we term dynamic layout programming. These semantics are specified on complet references. They allow one to state how a complet reacts to the movement of a related complet. FarGo provides a range of primitive reference types, as well as means to extend them with user-defined types. The basic (and default) reference type is the link reference, which acts as a remote reference but in addition keeps track of the (possibly moving) target complet. A pull reference means that whenever the source complet moves, the target complet automatically moves along (somewhat similar to attachment of one mobile object to another in Emerald [5] and in Dowl [2]). A duplicate reference means that when the source complet moves, a copy of the target complet is moved along (instead of the original complet, as in the case of pull). This is useful when replication can be used (e.g., for readonly data sources), without violating the logical semantics of the application. A stamp reference means that when the source complet relocates, a complet with an equivalent type of the original target complet should be looked-up and connected at the new location of the source complet. For example, if the target complet encapsulates a hardware device...

---

1For brevity we refer to complet instances simply as complets, unless otherwise specified.
such as a printer, a source complet (e.g., a mobile desktop complet) could use a stamp reference in order to reconnect to a local printer (complet) after it arrives at a new location. Finally, FarGo provides the capability to manipulate the type of references at runtime through a reflective mechanism that reifies the reference and allows to change its type.

Layout programming defines co- and re-location constraints given that (at least) one complet has decided to move. The third facet, termed relocation programming, allows to explicitly specify relocation policies, i.e., under what circumstances should complets be relocated in the first place, and where they should be relocated to. These decisions are typically driven by monitoring resource availability and use. FarGo provides three relocation programming interfaces: a monitoring API that can be used directly from within applications, a scripting language, and a graphical monitor. FarGo scripts are defined externally, possibly after the application has been deployed. Thus, they may be used by administrators who can control the layout of components separately from the encoding of individual components or from co-location-related constraints.

For more details on the programming model per-se, see [10]. We now turn to the realization of this model in the runtime environment.

3. The Runtime Environment

From the programmer’s point of view, the distributed runtime environment is transparent while programming the logic of the application. The basic structure and behavior of the application, including complet instantiation and references that connect them, does not explicitly involve any interaction with the runtime. Such explicit interaction is only needed when programming the layout, or for activating general runtime facilities (e.g., a naming service).

FarGo’s runtime infrastructure is composed of a collection of distributed stationary components called Cores. Each Core runs within an instance of a single Java Virtual Machine (JVM), which in turn runs inside a single operating system process. Notice that this means that the process boundaries of the application are changing while the application is executing; the application might occupy one process per complet at one time, and a single process for all complet at another time.

The architecture of a single Core is shown in Figure 1. The Reference Handler is responsible for realizing complet references, including their semantics, dynamic evolution, and support for location transparency. The Movement unit actually migrates complets. In addition to “ordinary” movement, this unit must also ensure that the various relocation semantics of complet references are preserved. The Invocation unit implements the parameter passing scheme over complet references. These three entities, together with the

![Figure 1. Core’s Architecture](image)

Monitoring facility (discussed separately in the next section) realize most of the runtime support for complet and layout programming, and are discussed in depth in this section. The Complet Repository stores complet references and is used by the Naming service to map logical names to complet, and the Peer Interface layer performs low-level Core-to-Core communication. They are not discussed any further.

The Core API provides various services to applications, including initial activation of the Core, manipulation of complet references, movement, naming, remote complet instantiation, and monitoring API. Programmers use both the Core API and the standard Java API to implement their application’s complets.

Finally, there are several system complets, which are outside the Core either because they need to be able to move (recall that the Core is stationary), or because they are directly pointed by complet and are somewhat external to the kernel. An example is the FarGo shell complet, which provides a user-interface to administrators.

3.1. Inside Complet References

Achieving syntactic and (partial) semantic transparency for remote referencing requires the use of some “proxy” element that has the same interface as the remote object. Typically, then, remote referencing is implemented by a regular local reference from the source to the proxy object, which in turn points to the remote object (through some system-generated code). However, in addition to basic remote referencing capability, a complet reference also needs to keep track of its possibly moving target.

Although it is still possible to implement this additional functionality within the proxy, our implementation divides the proxy into two separate entities: a stub and a tracker.
The stub object is pointed by the source object using a regular local reference and its interface is identical to the interface of the target anchor. The tracker is responsible for tracking the target complet and for actually performing invocations. There are two advantages to this separation. First, the stub always refers only to a local object (the tracker). In particular, whether the target complet is local or arbitrarily remote, this has no effect on the stub. Thus, the stub’s interface can be nearly identical to that of the target’s anchor, because no remote (or local-remote detection) code need to exist in it. This design facilitates syntactic and semantic transparency, since the programmer can access the stub exactly as if it were the target anchor. Second, this addition of a level of indirection allows to keep only one tracker per target complet in a single Core, although the number of complet references that point to this target complet can be large. This design enhances scalability, with a small price of an extra local method invocation.

Figure 2 illustrates the internal structure of complet references. The left side shows two complets, α and γ, both pointing with complet references to a third remote complet, β. The right side of the figure shows a closer look at these references. The tiny diagrams inside α and β represent intra-complet objects that are part of their closure. β’s stub has an outgoing regular (local) reference to the (local) tracker, for passing all invocations, and a reference to a meta-reference object whose purpose is explained later. Other than that, the stub has identical signatures of methods and constructors as those of β’s anchor. In order to distinguish between the name of the stub’s class and the anchor’s class name, the latter should be augmented with an underscore character (done by the programmer), as seen below.

Notice that there is only one tracker for β in Core1, even though it is referenced by two complets, α and γ.

Figure 3 shows the definition of a complet anchor, Message, followed by code that instantiates that complet and assigns it to the msg variable, moves it to another Core, acadia, and invokes its print method. Notice how msg is treated as the anchor (complet) by the program, both in terms of instantiation and use. In effect, it is an instance of the stub class Message, and its constructor generates the entire complet reference up to the (instantiated) anchor. The stub, as well as the tracker and the rest of the complet reference, are generated automatically by the FarGo Compiler, which accepts as input the anchor class.

The tracker(s) are responsible for achieving location transparency (as in [15, 6]). Upon the arrival of a complet to a new site, a new tracker is generated there and is set to directly point to that complet. Then, the tracker at the old site is set to point to the tracker at the new site. After several hops, a chain of trackers is being formed, and each tracker forwards invocations to the next one until the target’s anchor is reached and invoked. Figure 2 shows a chain of trackers that has been created while β has moved from Core1 to Core2, then to Core3, and finally to Core4. In addition, the figure shows (part of) a complet in Core5 that has started pointing to β when it was in Core2. Chains are automatically shortened by the runtime. For example, while returning from each invocation, all the trackers in the chain are set to point directly to the target’s location, and all trackers that are not pointed at all after shortening become available for garbage collection.

The Core’s invocation mechanism implements the parameter passing semantics (regular objects by value, an-
allow an object to dynamically and autonomously change
cocations. In Hadas, we used a reflective object model \[9\] to
flexibility of reflection for programming distributed appli-
Hadas2 framework \[4\] was with respect to the utility and
ing the invocation syntax, or the object model in general.
One of our important conclusions from working on the

dynamic evolution of the reference semantics, without chang-
ing a regular (non-anchor) object (by value) that may contain out-
put references. First, when passing an anchor (complet) as pa-
parameter (thus by reference), the question is what reference
type should be assigned to it at the receiving complet. One
alternative is to retain its current type. However, since the
complet reference is conceptually part of the source com-
plet, when it is passed to another complet there is no rea-
son to enforce the old type on the new containing complet.
Thus, before passing it, its reference is “degraded” to the
default link type. The second issue involves passing a
regular (non-anchor) object (by value) that may contain out-
going complet references in its containment graph. This is
handled by observing the referenced objects in the graph
and not copying referenced anchors (which are never passed
by value). Thus, the object graph is copied along with all
the outgoing complet references (again, degraded to type
link) but without the complets themselves.

3.2. Reflection

One of the challenges in supporting the special reloca-
tions by complet reference), and gets activated by the
tracker before forwarding an invocation towards the target.
Two new issues arise here due to the semantics of complet
references. First, when passing an anchor (complet) as pa-
parameter (thus by reference), the question is what reference
type should be assigned to it at the receiving complet. One
alternative is to retain its current type. However, since the
complet reference is conceptually part of the source com-
plet, when it is passed to another complet there is no rea-
son to enforce the old type on the new containing complet.
Thus, before passing it, its reference is “degraded” to the
default link type. The second issue involves passing a
regular (non-anchor) object (by value) that may contain out-
going complet references in its containment graph. This is
handled by observing the referenced objects in the graph
and not copying referenced anchors (which are never passed
by value). Thus, the object graph is copied along with all
the outgoing complet references (again, degraded to type
link) but without the complets themselves.

Figure 3. Complet definition and invocation

3.2. Reflection

One of the challenges in supporting the special reloca-
tion semantics of complet references is how to allow dy-
amic evolution of the reference semantics, without chang-
ing the invocation syntax, or the object model in general.
One of our important conclusions from working on the
Hadas\(^2\) framework \[4\] was with respect to the utility and
flexibility of reflection for programming distributed appli-
cations. In Hadas, we used a reflective object model \[9\] to
allow an object to dynamically and autonomously change
its functionality while deployed and used by other objects.
We have applied this principle in FarGo, but while in Hadas
reflection imposes a unique message passing (and object)
model, in FarGo our goal was not to change the regular Java
model for inter-object communication.

The general idea is to reflect upon the (complet) refer-
ence, instead of the object. Specifically, each complet ref-
ence has a meta reference object that reifies its relocation
semantics and allows to change it. As shown in Figure 2,
this object is contained by the stub, and can be fetched in
the program by calling the Core’s getMetaRef method.
For example, the following code fragment retrieves the
meta reference of the msg reference, and then checks
whether the type of that reference is link, in which case it
changes the type to pull.

```java
MetaRef metaRef = Core.getMetaRef(msg);
if (metaRef.getRelocator() instanceof Link)
    metaRef.setRelocator(new Pull());
```

The meta reference’s methods getRelocator
(setRelocator) get (set) an object that reifies the
reference relocation type. In addition to these two methods,
the meta reference has other reifying methods, e.g., for
going the current location of the reference’s target. The
key point is that the rest of the program preserves the
complet reference’s transparency, and no changes are
needed to the referencing mechanism (as seen in the use of
msg in Figure 3).

3.3. Movement under Layout Constraints

Recall that a movement of one complet may result in
movement or construction of other complets that are pointed
by it(e.g., by pull or duplicate), where the kind of
movement, if any, depends on the type of the complet ref-
rence. Contemporary movement mechanisms are usually
based on a marshaling mechanism (e.g., Java Serializa-
tion \[17\]) that traverses the moved object’s graph and copies
all its objects into a byte stream. This stream is sent to
the new location, where a complementary procedure unmar-
shals the graph into memory.

FarGo uses a similar approach. During the graph traver-
sal, the mobility protocol detects all the complet references
that are pointing out of the moved complet, and for each
such reference it applies a special routine to handle its mar-
shaling. For example, if the detected reference is of type
pull (namely it should move along with the pointing com-
plet) the traversal continues to recurse into this pointed
complet, which is then marshaled into the byte stream. In
the case of a stamp reference, only the type of the pointed
object is marshaled, but the traversal does not recurse into
it. A complementary procedure is performed during unmar-
shaling at the new site, where again, a special routine is
applied for each unmarshaled reference type. For example,

\(^2\)http://www.dsg.technion.ac.il/hadas
in the case of a stamp reference, a local instance of the target’s type is located and is set as the reference’s target. Notice that all complets that should move as a result of the same movement request are part of the same stream, thus only a single inter-Core message is involved.

With respect to link references, the mobility protocol must ensure that all outgoing complet references (from any object inside the complet) and all incoming complet references (which all point to the anchor) keep pointing to their targets after movement. Incoming complet references are taken care of by simply making the local tracker point to the new tracker at the new location. Since all incoming references point to the tracker, any reference to it are necessarily forwarded to the new location. Handling outgoing references is more complicated, because multiple complets may move along with the original one (e.g., in case of pull or duplicate references). To handle this case, during the traversal of the object graph, all the trackers that locally point to any complet that is about to move are first recorded. When the graph is unmarshaled, the receiving Core returns to the sending Core a set of remote references to the trackers of all the complets that have arrived, which are used to update the recorded trackers.

Another important design issue is how to integrate the predefined set of reference types into the protocol in a manner that allows the extension of the reference type hierarchy. This is done as follows. The behavior imposed by the type of each complet reference is implemented by a special Relocator object, which is contained in the meta reference. The Relocator implements the special routines that govern (un)marshaling of its containing complet reference. A new reference type can be implemented as a new Relocator object, possibly by extending one of the predefined Relocators.

Finally, FarGo supports weak mobility, which means that the stack and program counter do not move, only object state. Thus, in order to allow self movement of components, FarGo provides means for continuation of the computation after movement. Two types of continuation are supported. The first one is similar to the “call with continuation” programming style of Lisp, whereby an invoked method receives a continuation method as a parameter, which is invoked by the receiving Core after unmarshaling. A continuation method can be given to the Core’s movement primitive as follows:

```java
Carrier.move(msg, // the moved complet
    "accadia", // destination
    "start", // continuation method
    new Object[] {a1, a2}); // arguments
```

The second type of continuation is based on callback methods which should be supplied as methods of the anchor class by the programmer and are invoked by the movement protocol in different phases of the movement. There are four such methods. preDeparture is invoked before the movement at the sending Core, preArrival is invoked before finishing unmarshaling at the receiving Core, postArrival after unmarshaling, and postDeparture right before releasing the old copy of the complet (that was left behind in memory) for reclamation by the garbage collector.

A complet can actually move itself simply by passing its anchor to the above move method.

4. Monitoring Support for Relocation

As mentioned earlier, layout programming focuses on defining relocation constraints between complets, whereas relocation programming allows the definition of relocation policies. Monitoring support consists of four elements: profiling services, asynchronous event handling, script interpretation and visual representation.

4.1. Profiling Services

FarGo provides two kinds of profiling services: system profiling and application profiling. The former kind is more common (e.g., Sumatra [1] provides similar system services). Examples of measurable system services include completLoad, which counts the number of complets that reside in a given Core, and bandwidth, which measures the bit rate along the network connection between two sites. Application profiling is less common and is harder to achieve since it involves interaction with an application that is unknown to the system. Thus, the runtime can provide application-level profiling only when the application accesses the Core. Fortunately, many interesting measures can be provided due to the fact that complet references are accessible by the Core. An example of a useful application-level profiling is the invocation rate along a complet reference. Using this measure, one relocation policy in an application may be to move two disparate complets to the same site only if the bandwidth between the sites is below some threshold value and the invocationRate is above some threshold value. Otherwise it keeps them apart to spread the load.

Another property of profiling services is whether they are continuous or instant. The latter is used when current values are needed, whereas the former is used for measurements over time, where the returned value is some average value (typically an exponential average). Some services (e.g., complet size) are expensive to monitor and more useful when measured instantly, while other services (e.g., bandwidth) are more useful when measured over time, but the profiler provides both interfaces for each service and leaves these considerations for the administrator to make. The instant interface is done through regular methods of the
Core. The continuous interface consists of three Core methods per service. A **start** method notifies the Core to begin profiling of the resource, along with an interval parameter that specifies the length of the period between measurements. A **get** method calculates the current average value, and the **stop** method terminates the profiling if no other application has requested it. Thus, the Core monitors only resources that some application has interest in, minimizing system overhead. Finally, to further decrease the monitoring overhead, the monitor caches recent results so successive instant requests can be served without re-evaluation.

### 4.2. Monitor Events

The profiling mechanism enables applications to examine the state of resources in a synchronous fashion. However, quite often applications need to be notified asynchronously when certain resource levels change beyond some threshold, instead of having to continuously poll the resources. This is particularly important for complets that manage resources, which are the prime target of profiling. Thus, FarGo provides an event based mechanism that allows complets to register for Core events and get notified when the events occur.

Specifically, every profiling service has a corresponding event to which complets can register, along with a threshold value. When that value is obtained, the event is generated and the listening complets gets notified. Internally, the event registration mechanism invokes the proper **start** method, and the threshold value is kept separately with the listener, in order to filter the results. This design allows many listeners (threads) without overloading the measurement unit.

In addition to profile-based events, each FarGo Core always fires certain non-measurable events that reflect changes in the state of the environment, including changes to the layout. For example, every complet relocation fires a **completDepartured** event at the source Core and a **completArrived** event at the destination Core. Another example is the **CoreShutdown** event, which can be used by applications to migrate their complets to another Core in order to keep their applications alive.

Consistent with the “quasi-Java” approach, programming FarGo events is similar to Java’s event model (part of the JavaBeans spec [18]), and is extended with support for distributed events. This capability is essential for enabling (distributed) layout control. In addition, it enables complets to migrate to different sites and still be able to catch the (remote) events to which they are registered as listeners.

### 4.3. Scripting and Visual Representation

The monitoring API that was described thus far requires the administrator to program the layout from within the application. FarGo provides an alternative high-level scripting interface that eases the administration task and further decouples the relocation task from the application programming. The scripting language is event-driven, consisting of a set of event-action pairs, or rules. The event part of the rule corresponds to the monitoring events described above, and the action part involves basic movement commands. As in other scripting languages, the action part is not limited to the built-in commands, and can be extended with any user-defined (Java) class, which is automatically loaded upon its invocation.

The following example script has two rules. The first “reliability” rule (lines 4-7) listens to shutdown events on certain Cores, and moves all complets from the Core that fired this event into a “safe” Core (bound to the $targetCore variable). The second “performance” rule (lines 8-11) listens to the profiled event that occurs when the method invocation rate between two complets is greater than 3 invocations per second. Upon the occurrence of this event, the source complet is moved to the Core of the target complet.

```
1 $coreList = %1
2 $targetCore = %2
3 $comps = %3
4 on shutdown firedby $core
5  listenAt $coreList do
6   move completsIn $core to $targetCore
7 end
8 on methodInvokeRate(3)
9   from $comps[0] to $comps[1] do
10  move $comps[0] to coreOf $comps[1]
11 end
```

Finally, in addition to the programmable scripting interface, FarGo provides a visual layout and monitoring tool, whose snapshot is shown in Figure 4. The graphical monitor can connect to multiple cores, and show in real-time which complets reside in which cores. Thus, a movement of a complet is tracked by the viewer (who listens for such events at the inspected cores) and is automatically shown to the user. In addition to viewing complets and complet references, this monitor allows the administrator to perform various layout manipulation tasks, including moving complets between Cores (by drag-and-drop), viewing properties of complet references, including profiling information (average number of invocations per time unit, average network bandwidth, etc.), and examining and changing the type of complet references (e.g., from link to pull).

### 5. Implementation Notes

FarGo is a pure Java system, implemented with Java 1.1, and the communication layer is implemented on top
Figure 4. The Graphical Monitor

of Java RMI [19]. The Core, including all the functionality described in this paper, is fully implemented, along with a set of programming tools, including the compiler that generates complet stubs and trackers and a command-line shell for administering remote Cores. The Core is multi-threaded, to enable concurrency within its (single) process. For example, each invocation of an anchor’s method starts a new thread that executes the invocation, and each monitoring event is asynchronously notified to the application by starting a new thread that invokes a method of the listener complet. The monitor’s graphical viewer and the low-level profiling and events are implemented, allowing programmers to access the monitor from within the applications. The scripting language is still under development. FarGo’s current codebase includes about 40,000 lines of code, and the Core’s binary footprint is of about 260KB. The system is available for download and experimentation at http://www.dsg.technion.ac.il/fargo.

6. Related Work

Programming distributed systems with mobile code frameworks has become an increasingly active research area in recent years (for a survey see [7]). Many frameworks have been developed, including Java-based systems such as Aglets [12], Sumatra [1], Concordia [13], Mole [16], Voyager [14], and JumpingBeans [3]. Frameworks based on other platforms are AgentTCL [8], and Tacoma [11], which is an extension of Unix.

There is a major difference between FarGo’s programming model and that of systems that are based on the autonomous mobile agent paradigm, which greatly affects the required system support. Most agent models are very different from the underlying language’s object model. Communication with the agent, its instantiation and the usage of exceptions cannot be done by means of the language’s method invocations, object instantiation and exception handling models, as in FarGo. In Sumatra, for example, the mobile entity is not modeled as a regular object, it is rather a special group object to which aggregates join by explicitly checking in/out, which requires special invocations. In Aglets, two remote agents cannot communicate since no movement-aware remote referencing mechanism is supported. Message dispatching and processing for inter-agent communication should be implemented in Aglets by the programmer and cannot be done by regular method invocation. The programming model of Voyager is closer to FarGo, but it does not support regular instantiation of mobile objects (using new) or direct use the self (this) reference of a mobile object. Most importantly, none of these systems support the concept of layout programming via inter-component reference, using a reflective approach that minimizes the impact on the regular semantics of a reference.

With respect to monitoring, there are few systems that address this issue for mobility purposes. One such system is Sumatra. But in Sumatra, policies can be encoded only using an API. FarGo includes, in addition to the API, the event model that maps monitoring results and a high-level scripting language with the capability to attach layout scripts to active applications. Although Sumatra is also Java-based, unlike FarGo most of its monitoring support is implemented either by changes to the JVM, or by external non-Java processes that intimately interact with the operating system using Unix-specific mechanisms (e.g., signals). While this approach has several advantages, e.g., supporting strong mobility from the interior of the JVM, it implies that Sumatra is a non-portable and Unix-specific system, which makes it unsuitable for large-scale distribution. In contrast, the entire Core of FarGo including its monitoring layer, is implemented in Java, which makes its large-scale deployment much easier.

7. Conclusions and Future Work

Programming dynamic layout policies separately from the application’s logic enables a large-scale distributed application to adjust to environmental changes during its execution, thereby enhancing its efficiency, stability, responsiveness and fault tolerance. Supporting such capability requires special system support for applying the semantics of relocation policies and for supplying monitoring services to the application, on which relocation decisions could be based. The basic mobility and location-transparent remote referencing mechanisms that are required, should be tailored to support high-level layout abstractions (such as ref-
ferences that embody relocation semantics), and low-level profiling of resource consumption during execution. FarGo provides such support and implements a unique programming model that is very close to the programming model of the application’s implementation language, while allowing sophisticated layout programming and monitoring-based relocation with orthogonal interfaces.

There are several future directions. We intend to design a global location-independent naming scheme, which will present an alternative to tracking complet objects using chains. In addition, we plan to develop persistence and mobility-aware transactional models, and we are working on a security and resource negotiation models.

Acknowledgements

This research is supported by the Israeli Ministry of Science, Basic Infrastructure Fund, Project 9762. We would like to thank Boris Lavva and George Heineman for invaluable discussions and advice throughout the project. Idan Zach and Udi Shitrit have implemented the graphical monitor, and Ofer Tal and Opher Klingman have worked on the FarGo Compiler. Tamir Ronen is working on a location-independent naming scheme, and Yoaid Gidron is working with us on adaptive resource negotiation and allocation schemes.

References