Dynamic Layout of Distributed Applications

Ophir Holder and Israel Ben-Shaul

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

Abstract

We propose a novel capability for the architects of modern large-scale distributed applications — manipulating the location of their components at runtime. We argue that such “dynamic application layout” greatly elevates system scalability. Around this capability, we provide a model for dynamic layout programming that is separate from the programming of the application’s logic. We show that this model improves both programming and system scalability. The FatGO system overviewed here realizes this model by providing a compiler and runtime support for layout programming, including component mobility and inter-component references, automation and enforcement of relative co-location invariants, and monitoring facilities that allow to program the layout based on runtime information.

1 Introduction

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 systems 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 potentially large deployment space — i.e., the possibly large number of participating hosts with various capabilities, connected over links with various capacities — implies that the designer is unlikely to know a priori how to structure the application in a way that best leverages the available infrastructure. Geographical distribution between sites that are connected via a public network (Internet) introduces variable and possibly long latencies and higher chances of partial failure compared with the fixed-bandwidth and reliability of a private LAN. Often, such global applications have long duration, further demanding flexibility in placement and activation of functionality since hosts may come up, go down, or move sideways (i.e. mobile hosts) while the application is active.

We argue that current distributed programming paradigms are not adequate for this new generation of global applications.

2 Implicit vs. Explicit Distribution

The implicit approach to distributed programming advocates applying the local programming model to the distributed case. In the context of distributed objects, this means that there exist a unified object model, and the goal is to make remote method invocation syntactically and semantically equivalent to local method invocation. By hiding the location of the object as part of its implementation, programmers can use the same interface, or more generally use the same programming model, for local and remote objects. An advantage of this approach is that it 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 is geared towards providing programming scalability.

From the system-scalability perspective, the implicit approach might seem appealing too, since by avoiding hard-wired specification of the local-remote relationships such decisions may be deferred to runtime, and possibly even changed dynamically. However, this approach has some major drawbacks. First, as clearly pointed out in [9], the approach neglects the fundamental differences between local and distributed computing. If ignored, programmers are likely to encode applications that are unreliable since high latency and (partial) failure cannot be treated robustly. Performance is also likely to be poor, since location-related optimizations cannot be made. In short, the implicit approach does not support system scalability.

Second, semantic equivalence between local and remote invocation is impossible to achieve, simply because the local model inherently assumes shared memory between objects to enable by-reference argument passing. Two possible solutions are: (1) to implement distributed shared memory (DSM) (e.g., TreadMarks [1] and Millipede [4]); or (2) to eliminate by-reference argument passing in local programming. Neither alternative is too attractive, because they both require changes to the local programming model. DSM requires the use of “system” references (as opposed to regular programming-language pointers), and furthermore, it is in itself not scalable to wide area networks. On the other hand, eliminating call by reference takes away a very common and effective facility in local programming models (pure functional languages with no side-effects have not been very popular among programmers). It makes no sense to elim-
inate an important local programming mechanism just because it cannot be used in the distributed model, for the sake of semantic equivalence. Consequently, programming systems that attempt to follow the implicit approach tend to compromise on absolute semantic equivalence, which contradicts their initial goal. This, in turn, leads to confusion among programmers and ends-up degrading programming scalability too.

The explicit approach to distribution, led by Java RMI [8] and CORBA [6], argues that it is necessary to distinguish between local and remote objects in order to provide system scalability. Thus, it requires different interfaces (not only different implementations) for local and remote invocations, and even require different syntax and/or notations that force 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 the (empty, but mandatory) Remote interface and each remote method must throw a RemoteException. By not masking the distribution, this approach is more scalable systemwise (i.e., addressing fault-tolerance, performance, etc.) compared to the “naive” implicit approach.

A clear drawback of the explicit approach, however, is the tight-coupling between the programming of the distribution and programming of the application’s logic. This coupling hurts programming scalability. Table 1(a) summarizes this analysis. (In practice, the implicit-explicit range is continuous with many points in between, but for simplicity we show only the extreme points.)

Although better than the implicit approach with respect to system scalability, the explicit approach has system scalability problems of its own. The most significant of them is the early design-time determination of the local-remote partitioning and overall mapping of the application (logic) onto a set of (physical) processes/hosts, which we generally term the layout of the application. Design time layout means that the layout is fixed and cannot be altered afterwards unless the application is reengineered.

Fixing the layout in an early design phase might lead to high costs at run time. In order to be able to correctly determine the layout of an application, the design phase must involve a-priori structural and behavioral assumptions regarding the underlying physical system, assumptions that might not be known to developers, or not hold at deployment time or at runtime. For example, the designer must assume that certain hosts perform better than others in order to utilize the resources wisely. Other assumptions include host existence and network connectivity.

3 Dynamic Layout Programming

Recalling the implications of global scope — large number of widely distributed and long-lived components — it is our position that both the implicit and the explicit approaches are inherently limited in their scalability because they lack dynamic layout capability.

Basically, dynamic layout means that the local-remote partitioning and overall mapping of the application (logic) onto a set of (physical) hosts can be manipulated at runtime, without making any changes to the source code of the application. Such capability introduces high benefits for both system and programming scalability (table 1(b)). With dynamic layout, applications can be adapted to external changes in the environment, including deterioration (or improvement) in network bandwidth, partial network or node failure (or addition of new nodes/networks), and changes in processor loads. Dynamic layout facilitates also better layout-related decisions even during design since the otherwise a-priori assumptions could be validated before deployment and alternative layouts could easily be experimented.

Given the clear benefits of dynamic layout, the key issue is how to design a distributed programming model that provides such dynamic layout capabilities 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 support, 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 are sustained. On top of the system level, FarGo provides a programming model layer that allows to specify various co-location relationships between components, as in the explicit approach, except: (1) these relationships are relative, i.e., the physical structure is retained although (parts of) the application may relocate; and (2) both the structure and re-location patterns may evolve at runtime to address the dynamic variability in the environment. The evolution may either be programmed early in design time from within the application, or after deployment using an external scripting facility. In order to enable layout programmers to base their re- and co-location actions on runtime information, FarGo provides a monitoring service that allows applications to register and react to 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, including shared memory, event-model, multi-threading, and synchronization. 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 allows to change the layout on-the-fly without changing the code of the application. Fargo 1.0 has been implemented and will be soon available for download and experimentation in http://www.dsg.technion.ac.il/fargo. A full account of FarGo is beyond the scope of this position paper, and we present a brief overview here. For more details, see [5].

4 FarGo Design Overview

A typical FarGo application is comprised of a set of completets. 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 (we will refer to complet instances simply as complets, for brevity). In order to preserve the local semantics of the programming language (in our case Java), regular objects, references and method invocations are used inside a complet.

Thus, compared with the granularity of the mobile entity in other mobile object systems, we take the midway between the fine-grained approach that allows every programming language object to be mobile (e.g., Distributed Oz [3]) and hence increases the overhead for local interaction, and between the coarse-grained approach that suggests to structure an application from communicating mobile processes
(e.g., TeleScript [10]) and therefore does not allow to “join” components into the same address space when needed.

Complets are interconnected via complet references. Unlike remote references in conventional distributed frameworks (e.g., RMI [8]), 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 [7]), complet references can be associated with rich semantics that describe various co- and re-location relationships between complets, as described below. These relationships are guaranteed to hold despite movement of parts of the application. 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. Thus, complet references are the major abstraction mechanism for layout programming in FARGO.

Syntactically, the use of complet references for method invocations is similar to local references. Thus, in the trade-off between forcing the programmer to use different syntax and object models, and between allowing same language/syntax but different semantics, we vote for the latter. Our experience in using Hadas [2] has shown that using a different object model (including means to access objects) is problematic for application developers, even if the model has powerful features that are targeted at dynamic distribution (including reflection, dynamic mutability and security).

The base semantics of inter-complet references are different from intra-complet (i.e., Java) references, however, to enable physical remoteness and dynamic relocation. In particular, parameters are always passed by value, and a complet reference is kept valid despite possible movement of its source or target complets. But perhaps the most interesting aspect of complet references is that they can be augmented with semantics that characterize the re- and co-location relationships between the source and target complets. Various built-in reference types are supported (automated and enforced) by FARGO, but arbitrary extensions can be added by layout programmers. For example, a pull reference means that the target complet follows the location of the source complet, thereby maintaining a co-location invariant between the two complets. It is useful when the complets 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 one complet needs to be referenced by other complets, or because their coupling is needed only temporarily). A stamp reference means that when the source complet relocates, a complet with an equivalent type of the original target complet (although not necessarily with an equivalent implementation) should be looked-up and connected at the new location of the source complet. This reference is useful for enabling 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 automatically reconnect the source complet to a local printer after it arrives at a new location. For a complete list of reference types, see [5].

System support is facilitated by a light runtime infrastructure, consisting of a set of distributed Core complets. Core complets provide system support for naming, mobility and for 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. Notice, however, that as a Java-compatible system, mobility is restricted to object state (data fields values), not to the internal program context (i.e., call-stack and program counter) which is currently not accessible in a standard Java virtual machine.

Finally, FARGO provides extensive monitoring facilities that enable to make layout decisions based on runtime information. For example, a layout programmer may specify that if the bandwidth between two complets is below a certain threshold, then the target complet should get co-located with the source complet. Relocation 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. Event-action specifications can be programmed inside the application using an API (although still mostly isolated from the rest of the code), or they may be encoded externally using a rule-based scripting language. For example, the first rule below states that when a new client is deployed, the server should reconsider its best location (in this case by invoking the user-defined findBest Java method) and move there; the second rule simply states that when the core shuts down, all local complets should move elsewhere. Needless to say, all existing incoming and outgoing complet references continue to be valid despite such movements.

```
Script CloseServer {
  on completDepartured(client, target) do {
    server = thisComplet();
    clients.add(client);
    bestSite = Locators.findBest(client);
    move(client, bestSite);
  }
  on coreShutdown(coreName) do {
    moveAll("zeus.technion.ac.il");
  }
}
```
5 Conclusions

The FatGo system proposes a new dimension of flexibility for the architecture of large-scale distributed applications — the ability to dynamically change the location of components. Around this capability, FatGo provides a programming model that reconciles the implicit and explicit approaches in a manner that ameliorates their limitations. No static commitment to absolute physical binding, yet providing facilities for specifying relative physical bindings for ensuring robustness and reliability, along with reflective facilities for dynamically evolving these bindings as needed, based on runtime information that is made available by the infrastructure. No tight coupling between application and layout programming models by separating the two programming tasks and distinguishing local from inter-component interaction, yet providing a uniform (and familiar) syntax and programming language for all tasks.

References


