FarGo Programming Guide

Ophir Holder
Hovav Gazit

TECHNICAL REPORT EE Pub 1194

Version 0.2, January 31, 1999

This document is a detailed guide to distributed application development with FarGo. It begins with an introductory chapter that describes an example application in detail. The following chapters cover the various capabilities of FarGo and show how to use them in applications.

Electrical Engineering Department

Technion — Israel Institute of Technology

Ophir Holder & Hovav Gazit
Department of Electrical Engineering
Technion — Israel Institute of Technology
Technion City, Haifa 32000
Israel

Phone: +972-4-829-4659
Fax: +972-4-832-3041
Email: `fargo@dsg.technion.ac.il`
URL: `http://www.dsg.technion.ac.il/fargo`

Preface

FarGo is a lightweight middleware and a programming environment for development and deployment of distributed applications that should operate in wide-area settings. The main facilities provided by FarGo to programmers implementing such applications are (1) component mobility and tracking, (2) the ability to encode relocation relationships between components and (3) an integrated monitoring service that enables to design relocation policies based on measurements of performance and resource usage.

Great emphasis was given to introducing an easy-to-use programming model. The result is a programming model that is very close in its syntax and semantics to the well known programming model of the Java language. In addition to being easy to use and master, it allows to straightforwardly employ Java's advanced APIs within FarGo applications.

This preface includes valuable information for anyone that wishes to work with FarGo or just to learn about it. Please read it through before you move on the the first chapter.

About This Document

This guide is intended to be the primary source for learning how to program with FarGo. It gradually teaches the basic concepts and techniques that are involved in developing and deploying applications, and includes many practical illustrations and examples. It is targeted to the experienced programmer, although generally, no special experience in programming distributed systems is required. This guide assumes that the reader has some experience in Java, and is able to read and understand basic Java programs.

For more information about FarGo the reader is directed to our publications [2, 3, 4, 5] and to FarGo's website at:

`http://www.dsg.technion.ac.il/fargo`

Download and Installation

The FarGo Toolkit can be downloaded from FarGo's website `http://www.dsg.technion.ac.il/fargo`. Assuming Java is installed on your machine, this distribution includes everything you need in order to develop and deploy FarGo applications. In particular, it includes an API class library, a set of tools, and documentation. Additional documentation, such as this programming guide, is available separately at the website. In case you don't have Java, the FarGo download page includes instructions on how to get it.

After downloading FarGo, the first place to look for information is the file `readme.html` in the installation directory. In that page, click "Installing FarGo" to see the installation instructions.

Examples

There are several examples which are included in the distribution, including their full sources and accompanying documentation in HTML. To see this documentation, use your browser to open the file `readme.html` that resides in the installation directory, and click "FarGo Examples".

The most comprehensive example, a groupware application called TODO, is extensively described in the first chapter of this guide. All the other examples are much more trivial and small, and are documented only in the HTML pages.

Things that are not included in version 0.2

Version 0.2 is an early access release. It does not include several important elements which are not yet ready for release. The most important ones are:

- Class loader — No special class loader is currently provided. Thus, all the classes of an application must reside at every host that might need to load them.
- Script language — The script language is currently not available (we expect it to be ready very soon).

In addition, there are several features that are not yet available and whose absence is clearly marked inside the text. Finally, we are sure that there are bugs in this pre-release (we hope not many), beyond the ones that we know of and that are listed on the website. We will be more than happy to receive your bug reports, and also to hear your general feedback (see the next section).

Your Feedback is Welcome

FarGo is a research project conducted by the Distributed Systems Group (DSG) at the Electrical Engineering Department of the Technion — Israel Institute of Technology. Your feedback is important for our research, thus, we will be happy to receive your comments, suggestions, bug reports and general feedback. To contact us, please send email to:

`fargo@dsg.technion.ac.il`

You are also welcome to subscribe to the FarGo mailing list (see the homepage for instructions).

Acknowledgments

First, we would like to thank our advisor, Dr. Issy Ben-Shaul. We thank Boris Lavva, our group's research engineer, for his great help through numerous technical discussions, and for his artistic work in designing and building the website. We thank Ofer Tal and Opher Klingman who have implemented the FarGo compiler. We also thank Udi Shitrit and Idan Zach who have implemented the graphical monitor.

Contents

List of Figures

Chapter 1

Getting Started

This chapter introduces FarGo by walking through the implementation and deployment of an example application. It includes a brief presentation of the system's fundamentals whenever necessary, but does not go into the details, which are deferred to the following chapters.

Before you begin reading this chapter, please make sure you have read the preface. It includes pointers to valuable information, in particular the examples that are included in the distribution, and the download and installation instructions.

The TODO application, which we are about to present, is included in the distribution, so the reader can open the source files, compile them and then run the application along with the explanations herein.

We assume hereafter that FarGo was installed in `/FarGo`.[1] The source code of TODO resides at `/FarGo/examples/src/todo` and the compiled class files are at `/FarGo/examples/classes/todo`. Before you start, please make sure that your PATH and CLASSPATH environment variables are set correctly. Consult the installation instructions in `/FarGo/docs/notes/install.html`.

[1] If you haven't yet downloaded and installed FarGo, please do so now. Consult the Preface of this guide for instructions.

1.1 The TODO Application

TODO is a simple application for collaborative management of a list of tasks by a group of people, which are working on distant machines. One possible use of TODO is for managing the todo-list of a software project by the developers. In fact, we use it to maintain the todo-list of the FarGo project itself.[2]

Basically, TODO enables many users to view and update the task list in parallel. Each user interacts with the application using a simple Graphical User Interface(GUI). The GUI enables the user to view and update the task list, such that every update issued by one user is instantly displayed on the screens of all the other users. The task list is persistently saved by the application on the disk, so that if the application is shut down and then reactivated, the task list does not get lost.

TODO is composed of the following five components (see Figure 1.1):

- User Interface — is the user's agent with which the user sees and updates the task list.
- Engine — is responsible for receiving updates from users, distributing (multi-casting) each update to all other users, and updating task list file.
- Startup — is used to add new users to the application. A new user first contacts this component to receive a User Interface, which is then used to interact with the application. This is similar to contacting a Web server for receiving an applet that can then be used to access some remote data source.
- File — this component encapsulates the logical file in which the task list is saved, and
- Disk — represents the physical disk which is used by File to get a filename.

Figure 1.1 shows the placement and usage of these components. The figure shows three users and a central server. Suppose the rightmost user wishes to

[2] We are actually presenting a simplified version of the application we use, for simplicity.

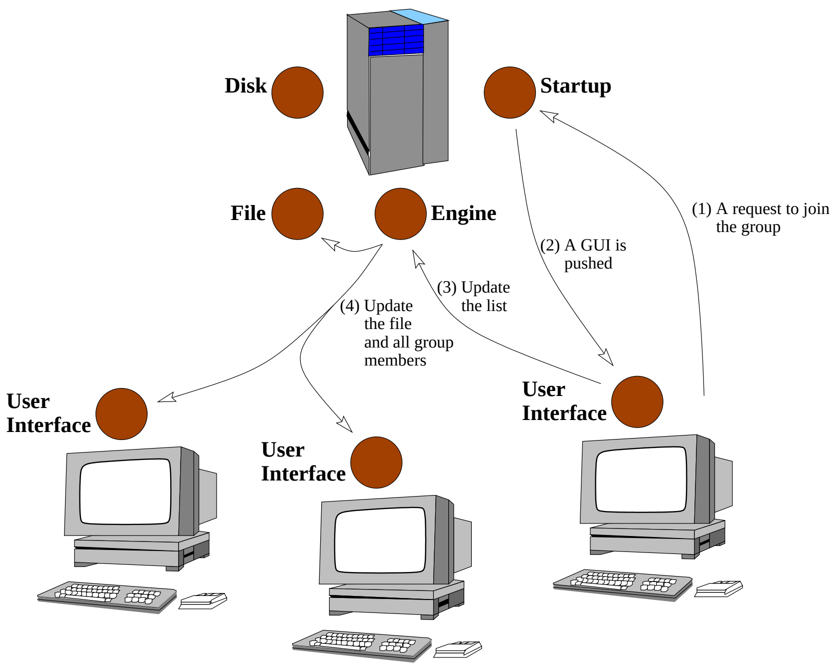

Figure 1.1: The TODO Application

join the group that currently manage the task list. Initially that user does not have any component of the application pre-installed, not even the User Interface. Thus, in order to join, the user must first issue a request (step 1), which is handled by the Startup component that resides on the central server. The Startup creates a new User Interface component and moves this interface to the user's machine (step 2). As a result, the user sees a new window on the screen with the task list shown in it.

Now, suppose our rightmost user wishes to change something in the task list, for example to add a new task. She first interacts with the User Interface to enter a description of the new task. When she finishes, the User Interface component that resides on her machine tells the Engine component that resides on the server that an update should be done (step 3). As a consequence, the Engine first updates the local File component, which writes the new task to the disk, and then updates the User Interfaces of the other two group members (step 4).

So far, TODO looks like a regular groupware application whose implementation does not require any special technology in addition to that of the standard distributed programming environments (e.g., Java RMI, CORBA or DCOM). However, let us now assume that TODO is about to be deployed in a very wide distributed setting. That is, group members are expected to be very far from each other and connected via the Internet. For example, the group can be a collection of developers working on a single product, but located in different branches of the company, each branch resides in a different continent. Taking into account the different time zones, we would expect the branches at one side of the globe to be active while their counterparts on the other side are idle, and vice versa. Thus, we will require that the server components, mainly the Engine that might introduce a performance bottleneck, will *move dynamically to get as close as possible to most of the users.*

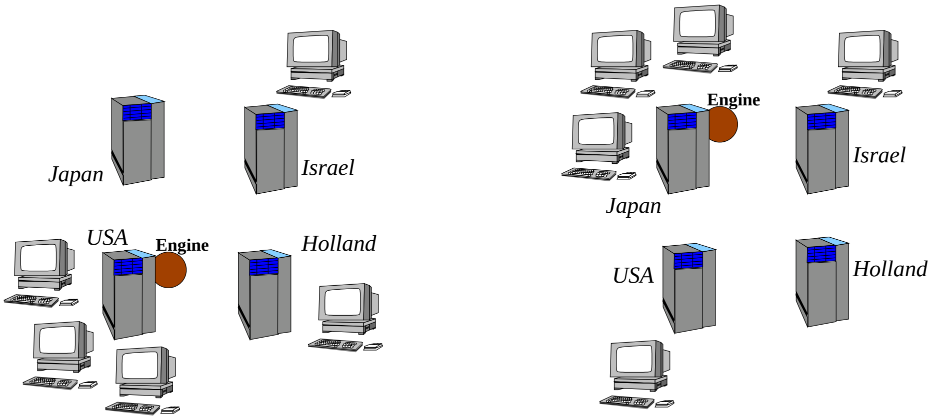

Figure 1.2: Wide-Scale Deployment

Figure 1.2 shows such scenario. There are four servers in which the Engine can operate, each server is located in a different country. The left side of the figure shows a time of the day when most of the clients are located in the USA, thus the Engine resides at the server at that country. The right side of the figure shows the situation several hours later, where most of the users are in Japan. As a consequence of the different spread of users, the Engine has migrated to the server in Japan so as to keep being close to most of its clients.

1.2 Implementing TODO

We proceed by gradually presenting TODO's design and implementation. The recommended design methodology is to start with the basic logic of the application which, in our case, involves receiving messages from users, updating the file, and multi-casting updates. The second phase involves augmenting mobility to the application. This step includes programming the "application layout", namely the interrelations among components with respect to their relocation, or in other words, how should components react to relocation of other components. Finally, that last step includes the encoding of the actual relocation of components.

1.2.1 The Basic Logic

One thing that differentiates a FarGo application from other standard distributed applications is that the components of the application might be required to *move during their execution*. Movable FarGo components are called *complets*. Basically, complets are very similar to regular Java objects or JavaBeans. However, in order to support mobility, they have several characteristics that are different from plain Java. We shall implement each one of the five components of TODO as complets.

Complets run in the *core* — FarGo's runtime environment. A machine in which complets are supposed to run must execute an instance of the core, which can be viewed as a (non-movable) server that can execute, send, and receive complets, as well as provide various services to FarGo applications (e.g., naming). A complet can move from core to core during its lifetime, however it is associated only with a single core at any given time.

We continue by presenting the code of each one of the five complets. If you wish to look at the files that contain that code, please notice that we firstly present a simplified version of the classes, and gradually add more code along this chapter.

Startup — The Application's Starting Point

Startup's definition is shown in Figure 1.3 and resides in file `/FarGo/examples/src/todo/Startup_.java`. It starts with the Startup's class declaration. Notice that the class name ends with an underscore character and also that the declared class implements the `Complet` interface. These two attributes distinguishes this declaration from that of a regular Java class and must be incorporated in the declaration of every complet. Startup has two data fields.

```
public class Startup_ implements Complet {
    private String name;
    private Engine engine = null;

    public Startup_(String[] args)
        throws CompletInstantiationException {
        name = args[0];
        Core.rebind(this, name);
        engine = new Engine();
    }

    public void connect(CoreURL dest) throws MovementException {
        UserInterface ui = new UserInterface(engine);
        engine.addGUI(ui);

        Carrier.move(ui, dest);
    }

    public void postArrival() {
        Core.rebind(this, name);
    }
}
```

Figure 1.3: The Definition of the Startup Complet

The first one, `name`, holds a string that is used to identify the TODO application by name (see below). The second data field will hold the Engine complet.

Startup's constructor starts by binding a name to Startup, so that users will be able to specify a URL when they issue a request to start using the application. Let us assume that the binding operation is done in a core whose name is `core` that listens on port 7000, on host `pooh.technion.ac.il`. After this operation finishes, Startup's URL becomes:

```
pooh.technion.ac.il:7000/core#todo
```

The next thing the constructor does after binding, is instantiating the Engine and depositing this complet into the `engine` data field.

The `connect` method is used to connect a new user to the application. The user specifies the above URL, which causes the core that runs Startup to automatically invoke `connect`. This method instantiates a new User Interface and adds it to the list of User Interfaces known to the Engine, which needs to refer to the new User Interface in order to update it about changes to the task list. Finally, the User Interface is moved to the core of the requester using the movement primitive `Carrier.move`. Notice how this method is invoked. Its first parameter is the complet to be moved and second one is the destination, in our case the location of the requester.

Engine — A Message Distribution Server

The definition of the Engine complet is shown in Figure 1.4 and resides in file `/FarGo/examples/src/todo/Engine_.java`. Like any complet, the class name ends with an underscore and this class implements the `Complet` interface.

Our Engine complet includes two data fields: `file` that holds the File complet (not `java.io.File`), and `GUIs` that holds a vector of User Interface complets. Whenever a user updates the task list, Engine invokes each User Interface in this vector in order to update it.

Engine has four methods. Its constructor simply initializes its data fields. The `addGUI` method adds a new User Interface complet to the `GUIs` vector.

```
public class Engine_ implements Complet {
    private File    file;
    private Vector  GUIs;

    public Engine_() {
        file = new File();
        GUIs = new Vector();
    }

    public void addGUI(UserInterface ui) {
        GUIs.addElement(ui);
    }

    public synchronized void addTask(String task) {
        file.addTask(task);
        for (int i=0; i<GUIs.size(); i++) {
            ((UserInterface)GUIs.elementAt(i)).addTask(task);
        }
    }

    public Vector getTasks() {
        return file.getTasks();
    }
}
```

Figure 1.4: The Definition of the Engine Complet

The addTask method receives a new task, writes it to the file and updates all the remote User Interfaces. We have intentionally omitted code that handles concurrency and failure in this method, for simplicity. Lastly, the getTasks method return all the tasks currently saved File as a single string.

User Interface — The User's Agent

The definition of the User Interface complet is shown in Figure 1.5 and resides in file /FarGo/examples/src/todo/UserInterface_.java. It includes two data fields: mainFrame that holds the AWT Frame object, which encapsulates the actual window on the screen, and engine that points to the remote Engine complet.

User Interface's constructor sets the engine data field, which is used to inform the Engine of changes to the task list that are issued using this User Interface.

The next method, addTask, is invoked by the remote Engine to inform this User Interface of a task that was added by another user. This method updates the display by appending the new task's description to the bottom of the tasks list.

The two last methods, preDeparture and postArrival are special call-back methods that are invoked by the core upon movement of this complet. preDeparture is invoked right before this complet leaves to a new destination, while postArrival is invoked once it arrives. A complet informs the core that these two methods should be invoked by implementing the MovementAware interface, as done by this complet.

In our case, preDepature simply close the tasks window if it is open and postArrival (re)opens that window at the new location. More specifically, postArrival begins by instantiating an AWT Frame, passing it a reference to the Engine so that it could ask the Engine to update the task list when required by the user. It continues by printing the tasks inside the window, where each task's description is followed by a short separating line. Finally, it sets the window's title to be the URL of the core in which this complet runs.

We do not present here the definition of the MainFrame class, which does the actual windowing work. This code has almost nothing that is interesting from

```
public class UserInterface_ implements Complet, MovementAware {
    private MainFrame mainFrame = null;
    Engine engine;

    public UserInterface_(Engine engine) {
        this.engine = engine;
    }

    public void addTask(String task) {
        mainFrame.tasksTextArea.append(task + "\n--\n");
        mainFrame.repaint();
    }

    public void preDeparture() {
        if (mainFrame != null)
            mainFrame.setVisible(false);
        mainFrame = null;
    }

    public void postArrival() {
        mainFrame = new MainFrame();
        mainFrame.engine = engine;

        Vector tasks = engine.getTasks();
        StringBuffer tasksList = new StringBuffer();
        for (int i=0; i<tasks.size(); i++)
            tasksList.append((String)tasks.elementAt(i) + "\n--\n");


        mainFrame.tasksTextArea.setText(tasksList.toString());
        mainFrame.setTitle("TODO: "+Core.getURL().toString());
        mainFrame.setVisible(true);
        mainFrame.repaint();
    }
}
```

Figure 1.5: The Definition of the User Interface Complet

```
public class File_ implements Complet {
    private DiskComplet disk;
    private Vector tasks;
    private java.io.File theFile;

    File_() throws CompletInstantiationException {
        DiskComplet disk = (DiskComplet)Core.localLookup("disk");
        if (disk == null) {
            String msg = "named complet not found: disk";
            throw new CompletInstantiationException(msg, null);
        }

        theFile = new java.io.File(disk.getFilename());
        try {
            if (theFile.exists()) {
                ObjectInputStream objectStream =
                    new ObjectInputStream(new FileInputStream(theFile));
                tasks = (Vector)objectStream.readObject();
            } else {
                tasks = new Vector();
                ObjectOutputStream objectStream =
                    new ObjectOutputStream(new FileOutputStream(theFile));
                objectStream.writeObject(tasks);
            }
        } catch (Exception e) {
            // ...
        }
    }

    public void addTask(String task) {
        tasks.addElement(task);
        try {
            ObjectOutputStream objectStream =
                new ObjectOutputStream(new FileOutputStream(theFile));
            objectStream.writeObject(tasks);
        } catch (Exception e) {
            // ...
        }
    }

    public Vector getTasks() {
        return tasks;
    }
}
```

Figure 1.6: The Definition of the File Complet

```
public class DiskComplet_ implements Complet {
    private String filename;

    public DiskComplet_(String[] args) {
        Core.rebind(this, "disk");

        // Expect args[0] to be a full path filename
        if (args.length >= 1)
            filename = args[0];
    }

    public DiskComplet_() {
        this(new String[]"tasklist");
    }

    public String getFilename() {
        return filename;
    }
}
```

Figure 1.7: The Definition of the Disk Complet

our perspective; it is a standard and simple AWT-based class (see `/FarGo/examples/src/todo/MainFrame.java`).

File — A Persistent Repository

File's definition is shown in Figure 1.6 and resides in file `/FarGo/examples/src/todo/File_.java`. It contains three data fields: `disk` is the Disk complet from which File receives a filename, `tasks` is the task list in the form of a string vector, and `theFile` is Java's `File` object that is used to actually contact the file system.

The constructor starts by asking the Disk to give it a filename. Then, if the file exists, it opens this file and reads the task list from it. If it doesn't exist, the file is created and initialized to contain an empty task list. Notice how the task list, which is represented as a string vector in memory, is saved into the file and read from it using Java Serialization [7].

The `addTask` method adds a new task to the list by first adding it to the vector and then writing the vector to the file on the disk. `getTask` simply returns the current vector of tasks.

Disk — A Local Provider of Disk Space

For now, the mere purpose of the Disk complet is to supply a filename to File. The reason we encapsulate this functionality in a distinct complet will become clear later, when we will add mobility to the the application. Disk's definition is shown in Figure 1.7 and resides in file `/FarGo/examples/src/todo/DiskComplet_.java`. The filename is stored in the `filename` data field and can be retrieved by the `getFilename` method.

1.2.2 Programming The Layout

After implementing the basic logic of TODO, we are ready to deal with movement and dynamic layout. As shown in Section 1.1, our final goal is that the Engine will move among several servers in order to be as close as possible to most of the users. We hope that such behavior will result in high responsiveness from the viewpoint of most of the users at all times.

However, if we move the Engine complet alone and leave behind the File complet, our efforts will most probably be in vain since the Engine must invoke File upon every change made to the task list. This observation suggests that the *File will follow the Engine to its new location* whenever Engine moves, so as to always be co-located with it to fasten these invocations. But this does not seem to be enough. When File reaches a new core, it needs to save the task list on some new local file that it must somehow receive. We will require that each one of the cores to which the Engine can move will run a Disk complet from which the arriving File could ask for a filename. Finally, we will also want that while the Engine is located in a core, new users will be able to contact that particular core in order to start using the application. For achieving this, we will set Startup to also follow Engine when it moves. But since the application's address that was known prior to the movement should remain valid, a copy of Startup should be left behind when it moves in order to connect new users that still specify the old address.

The above requirements define a set of relocation relationships between the complets. FarGo allows to specify such relationships using *reflective complet* *reference*. The idea is that the programmer can set attributes of the references between complets, attributes that implement these relationships. For example, returning to the relationship between Engine and File, the complet reference to File that is contained in Engine's data field `file` (see Figure 1.4) can be set to "drag along" its target complet, File, when its source complet, Engine, moves. To do so we should set this reference's *relocator* attribute. This attribute determines how the target of the reference is affected by a movement of the source. When the value of this attribute is *pull*, we say that the reference functions as a *pull relocator* and the target will be dragged along with the source.

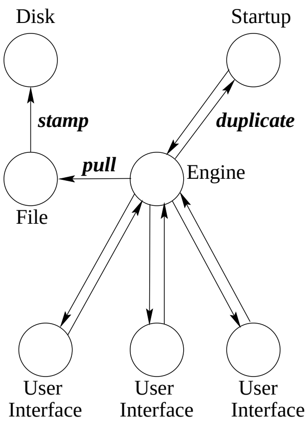

Figure 1.8: TODO's Relocators

Figure 1.8 shows all the complets of the application when three users are active, including all complet references and their desired relocator attributes. The reference from Engine to File is a *pull relocator*. The reference from File to Disk, which is held in File's `disk` data field (see Figure 1.6), is a *stamp relocator*. That is, whenever File moves, this reference gets disconnected from its target Disk before File departures, and reconnected to a local instance of Disk at the new core after File arrives there. Finally, the reference from Engine to Startup is a *duplicate relocator*, which means that when Engine moves, a copy of Startup moves along with it, while the original pointed Startup is left behind.

All other complet references with no explicit relocator type are *link relocators*, meaning that the location of their target is not affected by movement of the source. Nevertheless, these references keep being valid in spite of movement of their source or target. For example, all references from the User Interfaces to the Engine, held in User Interface's `engine` data field (see Figure 1.5), keep being valid while Engine moves among the servers.

An interesting thing about the way we set relocators is that it doesn't affect the usage of the references as we programmed so far. Thus, we will now add the code that sets the right relocator attributes to the classes of the previous section, while not changing any existing code.

```
public Engine_(Complet startup) throws CompletInstantiationException {
    this.startup = (Startup)startup;
    MetaRef mr = Core.getMetaRef(this.startup);
    mr.setRelocator(new Duplicate());

    file = new File();
    mr = Core.getMetaRef(file);
    mr.setRelocator(new Pull());

    GUIs = new Vector();
}
```

Figure 1.9: Setting Relocators in Engine's Constructor

Figure 1.9 shows Engine's constructor after adding code that sets reference relocators. Observe that now the constructor receives a reference to Startup in order to be able to "drag" its copy when moving to a new core. The third and fourth statements of the constructor sets this reference to function as a duplicate relocator. First, the core's `getMetaRef` method is invoked, given this reference as a parameter. This method returns a meta-reference object that *represents the internals of the reference*. This object is then used to set the relocator type of the reference by an invocation of `setRelocator`. The same technique is used to set the reference to File to become a pull relocator.

Figure 1.10 shows the code we add to File in order to get a new filename each time File moves. The constructor starts with looking up the Disk by name. As noted before, the deployment phase of TODO involves running an instance of Disk on each server the Engine can move to, now we see that Disk binds itself to the name `disk`. Once Disk was found, the reference to it is set to function as a stamp relocator using the same technique we saw above. The stamp relocator is set to find its new target after arrival to a new core with the `setEquivalence` method, which in our case, tells the relocator to point to a local complet to which the name `disk` is bound.

Pointing to Disk with a stamp relocator ensures that after each movement

```
public File_() throws CompletInstantiationException {
    DiskComplet disk = (DiskComplet)Core.localLookup("disk");
    if (disk == null) {
        String msg = "named complet not found: disk";
        throw new CompletInstantiationException(msg, null);
    }
    MetaRef mr = Core.getMetaRef(disk);
    mr.setRelocator(new Stamp().setEquivalence("disk"));

    // ...
}

//...

public void postArrival() {
    theFile = new java.io.File(disk.getFilename());
}
```

Figure 1.10: Implementing File-Disk Relocation Relationship

of File, it will be pointing to an instance of Disk that resides locally. Once arrived, File should contact that local Disk and get a new filename. To do that, File implements a special callback method, `postArrival`, which is invoked automatically by the core right after File arrives. This method simply invokes Disk to get a new filename, while the target of the invocation is always the local Disk.

This concludes the tasks we need to do in order to program TODO's layout, as depicted in Figure 1.8. After specifying how complets should respond to relocation of other complets, we are ready to specify which complets should move and under which circumstances, which is the subject of the following section.

1.2.3 Specifying Movement Policy

The last stage in programming a FarGo applications is to apply a movement policy to the complets of the applications. In the TODO application we want to `engine` to be as closed as posible to the clients. By saying `'closed to'` we mean that the sum of `network delays` between the engine and all the `GUI's` should be minimized.

FarGo has a special system complet – `LocationOptimizer` which implements such a policy by monitoring the network delay to a list of complets and whenever there is a dramatic change in these delays it recalculates the optimal location of its managed complet by finding a location from a list of posibble locations that minmized the `network delay` to the complets. In TODO the managed complpet is the `engine` and the the list of complets is the list of `GUIs`.

Figure 1.11 shows the code we add to `Engine` in order to bind it with the `LocationOptimizer`. Note that the reference to the reloc complet must be changed to pull connection in order to insure that the `LocationOptimizer` will be colocated with the `Engine`.

The `addLocation` method is used for specifing posiblle locations for the `engine`. The `Engine` will be located in one of these cores depending on the network delay to the `GUIs`.

After adding the GUI to the GUIs list The `addGUI` method adds the meta reference of the GUI to the reloc complet. The reloc complets starts to monitor the network delay of this reference and may relocate the `Engine` to one of the locations that were specfied before.

1.3 Compilation with the FarGo Compiler

FarGo applications are compiled by the FarGo compiler — fargoc. Fargoc is essentially a source-to-source converter that generates a special stub for each complet. After source generation, it invokes standard compilers (javac and rmic) to generate bytecode.

1.3.1 Compiling TODO

As noted in the beginning of this chapter, the sources of TODO resides in `/FarGo/examples/src/todo` (assuming FarGo was installed in directory `/FarGo`). We shall now compile these sources and build the application. We will order fargoc to put all the class files in `/FarGo/examples/classes/todo`. Notice that since the distribution already includes all the class files, you can skip this section and move ahead to Section 1.4.

```
Private LocationOptimizer reloc

public Engine_()
{

    // ...

    reloc = new LocationOptimizer(this);
    mr = Core.getMetaRef (reloc);
    mr.setRelocator (new Pull());

    reloc.addLocation (new CoreURL ("pooh.technion.ac.il/core");
    reloc.addLocation (new CoreURL ("piglet.technion.ac.il/core");
    reloc.addLocation (new CoreURL ("tiger.technion.ac.il/core");
    reloc.addLocation (new CoreURL ("kenga.technion.ac.il/core");

    // ...
}

public void addGUI (UserInterface ui)
{

    GUIs.addElement(ui);
    reloc.addReference(Core.getMetaRef(ui));

}


//...
```

Figure 1.11: Specifing movement policy of the Engine

To compile the application issue the `fargoc` command at the operating system's shell as follows:

```
/> cd FarGo/examples/src/todo
/FarGo/examples/src/todo> fargoc -d /FarGo/examples/classes *.java
```

If the prompt returns with no error message, then compilation has ended successfully. You should notice that for each one of the complet files, five class files were generated (in `/FarGo/examples/classes`). For example, the following class files are created for the Engine complet:

```
Engine.class
Engine_Tracker.class
Engine_TrackerImpl.class
Engine_TrackerImpl_Stub.class
Engine_TrackerImpl_Skel.class
```

The Tracker and TrackerImpl files implement FarGo's internal referencing mechanism. The Stub and Skel files are Java RMI stub and skeleton, which are needed since the referencing mechanism is implemented using Java RMI [8].

1.4 Deployment and Usage

Running TODO can be divided into two phases. First, the application should be set up so that users will be able to connect to it. After setup is complete, users first connect to the application, and then use it.

Notice that all the classes of TODO must reside on each host that executes a core, since the 0.2 distribution does not include any special class loader that can load application classes through the network (such class loader will be included in future releases).

1.4.1 Setup

Each FarGo application operates in the environment of the FarGo runtime, further referred to as the *core*. The core can be viewed as a server that operates on each machine that should receive, send and run complets.

Thus, the first step in deploying TODO is to start several cores at several machines among which the Engine will migrate. Starting a core on a machine is done by activating the `fcore` command from the operating system's shell. This command echoes some copyright information and then starts the core daemon:

```
/> fcore
FarGo Core 0.2  Distributed Software Systems Group (DSG)  EE Dept.  Technion-
Israel Institute of Technology  (C) Copyright 1996-1998  All Rights Reserved

Core is ready!
```

Let us assume that four cores have been started on hosts: `pooh`, `piglet`, `tiger`, and `kenga`. Each one of the cores will later be referred by its URL. In this example the URL for each core is identical to its host name. If you wish to work only on a single machine, you will have to locally start four cores, and give each one of them a different *local-name*. To do that, you should use the `-name` option of the `fcore` command. For example, the following command starts a core with the local-name `core1` on the local host.

```
/> fcore -name core1
FarGo Core 0.2  Distributed Software Systems Group (DSG)  EE Dept.  Technion-
Israel Institute of Technology  (C) Copyright 1996-1998  All Rights Reserved

Core is ready!
```

Assuming the local host's name is `pooh`, the URL of the core that has just been started will be `pooh/core1`.

Recall that each core to which Engine could migrate must run an instance of the Disk complet. Thus, we shall now instantiate this complet on each one of the four cores we have started. We shall do that with fshell — the FarGo shell, a tool for remote management of running cores (see 7.4). This shell is activated by invoking the `fshell` command on the operating system's shell. For the moment, we shall not use the shell interactively, we shall rather use it as a single command from the operating system's shell.

This is how we instantiate Disk at `pooh` and then at `piglet`. Please remember to instantiate Disk also on `piglet`, `tiger`, and `kenga` (use the same command on the same machine, only replace `pooh` with the other host names). We specify `disk` as the name that should be bound to each Disk complet that we load.

```
/>fshell -url pooh load todo.DiskComplet disk
FarGo Shell 0.2  Distributed Software Systems Group (DSG)  EE Dept  Technion-
Israel Institute of Technology  (C) Copyright 1996-1999  All Rights Reserved

establishing connection... done
done

/>
```

The next step is to instantiate the Startup complet. We shall use fshell to instantiate that complet at `pooh` while specifying the `todo` as a name to bind to this complet.

```
/>fshell -url pooh load todo.Startup todo
FarGo Shell 0.2  Distributed Software Systems Group (DSG)  EE Dept  Technion-
Israel Institute of Technology  (C) Copyright 1996-1999  All Rights Reserved

establishing connection... done
done

/>
```

This finished the setup and now TODO is ready to serve users.

1.4.2 Using the Application

To start using TODO, a user should activate `connect` command in fshell. This command receives the application's name (the name which is bound to Startup, `pooh/#todo`) as a parameter:[3]

```
/>fshell -noquit connect pooh/#todo
FarGo Shell 0.2  Distributed Software Systems Group (DSG)  EE Dept  Technion-
Israel Institute of Technology  (C) Copyright 1996-1999  All Rights Reserved

done

fshell@fshell>
```

As a response, TODO's window pops up on the screen (notice that on some systems it might be covered by currently active windows, on others it might

[3] by specifying the `-noquit` option to fshell, we tell it not to terminate after executing the command thus, its prompt appears. Use the command `quit` to close TODO's GUI and leave the shell.

be iconized). This window shows the current list of tasks and, by pressing a button, it allows to add a new task to the list. When several users are connected, every such update is shown in the windows of all users, right after it is issued.

Let us now use fshell interactively to move the Engine among cores and track the behavior of other complets. First, we open a new console window and enter the interactive shell with the command `fshell`. We then log into `pooh` and use the `ls` command to list all the complets in it:

```
prompt> fshell
FarGo Shell 0.2  Distributed Software Systems Group (DSG)  EE Dept.  Technion-
Israel Institute of Technology  (C) Copyright 1996-1998  All Rights Reserved

fshell@fshell> rlogin pooh
establishing connection... done

fshell@pooh> ls
5395031792347982758  DelegateComplet
5395031792348231453  File
5395031792348231442  Engine
5395031792348231441  Startup
5395031792348223208  DiskComplet
```

Now, let us move the Engine to `piglet`. We shall do that with the shell's `move` command as follows:

```
fshell@pooh> move -id 5395031792348231442 piglet
done

fshell@pooh> ls
5395031792347982758  DelegateComplet
5395031792347995598  Startup
5395031792348223208  DiskComplet
```

Notice that after the movement, only (a copy of) the Startup complet was left behind at `pooh`. You might now want to log into `piglet` and see which complets resides there. You will find the Engine, which was explicitly moved, but also File and Startup which were dragged by the pull and duplicate relocators.

To actually see the result of movement more impressively, you might want to move one of the user interfaces. First, use fshell to open a new user interface (by specifying the `connect` command), then, from that shell's prompt use `ls` to get the ID of the UserInterface complet, and finally move that complet with the `move` command:

```
/>fshell -noquit connect accadia/#todo
FarGo Shell 0.2  Distributed Software Systems Group (DSG)  EE Dept  Technion-
Israel Institute of Technology  (C) Copyright 1996-1999  All Rights Reserved

done

fshell@fshell> ls
6342758038935168411  DelegateComplet
8989467244945487344  UserInterface

fshell@fshell> move -id 8989467244945487344 piglet
done
```

As a result, the local window will disappear, and then reappear on `piglet`'s screen. Notice that the shell is actually a FarGo program that runs inside a core of its own. Thus, it is possible to see and move the active complets inside the shell's own core.

1.4.3 Working with the Graphical Monitor

****** not available ******

Chapter 2

Fundamentals

This chapter introduces the basic concepts of FarGo, as well as their concrete usage while developing applications. It should be regarded as a prerequisite for understanding the rest of this guide.

2.1 The Core

Generally, a FarGo application is a distributed program composed of several components, each running on a different machine. Unlike conventional distributed frameworks, these components can move among different machines while the application is executing.

To be able to run a component, a machine must continuously execute a special process named *core*. The core is a server that can contain and activate components, send components to other cores, and receive components from other cores. Each core has a name and can be identified with a URL (see Section 6.1.1).

The core is a single instance of the Java Virtual Machine (extended with some services), namely it is a single operating system process. It may contain, however, a number of components of different applications that may execute in parallel due to Java multi-threading. Every component, however, is associated with exactly one core at any given time.

Figure 2.1 illustrates a distributed application that is composed of five com-

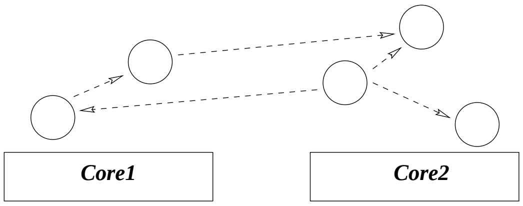

Figure 2.1: The Core

ponents, two of which are running in one core in one machine, and the rest in another core in another machine. The interactions between these components, which are depicted as dashed arrows, occur both *locally* between components that reside on the same core, and *remotely* between components that reside on different cores.

From the programmer's perspective, most of the functionality of the core is carried out behind the scenes. The idea is that the programmer writes components in regular Java, but compiles the application with a special compiler. The special generated code works in conjunction with the core to allow movement of components among cores while the application is executing. Nevertheless, some of the services of the core are exposed in the form of an API, to allow the programmer to control the *policy* of component relocation.

2.1.1 The Core API

The API of the core is available in the form of a Java class library. Most of the API has the form of static methods of the API classes. For example, to invoke the core's `getURL` method that returns the URL of the current core,[1] one should write:

```
CoreURL url = Core.getURL();
```

The classes of the core API are organized in the following packages:

- `fargo.core` The classes in this package provide the basic services of the core including component movement and naming.

[1] The usage of Core URLs is described in Chapter 6

- `fargo.complet` This package includes classes that help to define the interface of components.
- `fargo.prof` Classes of the API support for the profiling facilities.
- `fargo.event` This package supplies the event mechanisim of the monitor.
- `fargo.reloc` This package includes a set of relocation policies .

2.1.2 Starting and Stopping the Core

The core can be started from the operating system's shell using the command *fcore*. This command starts a local core process that occupies port number 3099 of the machine. This is how the shell looks when fcore is activated:

```
prompt> fcore
FarGo Core 0.2  Distributed Software Systems Group (DSG)  EE Dept.  Technion-
Israel Institute of Technology  (C) Copyright 1996-1998  All Rights Reserved

Core is ready!
```

Notice that `fcore` can receive several arguments. Like most FarGo utilities, you should call it with the `-help` parameter to get help on all these arguments. The `fcore` utility is described later in Section 7.3.

fcore starts the core as a daemon on your machine. A different way to start a core is from within a Java application. When a Java program that includes invocations of the FarGo API runs, a core automatically starts to be active inside that program's JVM when the first API method is called. Likewise, when that JVM's process is killed, the core that runs inside it also stops.[2]

Sometime, however, it is required to explicitly start and stop the core from within a Java program. For example, a program that wishes, at some point in its execution, to receive a component, change it, and then move it to another core, but never interact with it afterwards. It can start a core when it wishes to be able to receive the component and stop it after it has moved that component away. Another reason to explicitly start/stop the core is

[2]For example, the FarGo Shell, which is included in the distribution (see 7.4), is implemented as a FarGo program that does not perform any explicit action to start or stop the core.

to allow the system to respond to these actions, for example, to move all the components from the current core to another core right before a planned shutdown.

Explicit activation and shutdown of the core can be done from within a Java program by invoking the `startup` and `shutdown` services as follows:

```
Core.startup();
Core.shutdown();
```

Obviously, `startup` checks first whether a core is already active in the current JVM, in which case it does nothing and returns.

A FarGo application *must* call the `startup` methods at its very beginning in order to start FarGo's services inside its own process. It is highly recommended that such program calls `shutdown` before it terminates.

2.2 Complets

FarGo components are named *complets*. Complets are the basic building blocks of the application, somewhat analogous to modules in a conventional programming language, or to components in component architectures, such as JavaBeans [9] and CORBA [6]. As already mentioned earlier, unlike other component architectures, complets are *mobile*, namely they can move from one core to another during their execution.

Practically, a complet is a Java class that must implement the `fargo.complet.Complet` interface and whose class name must end with an underscore character, as in the following example.

```
public class Message_ implements fargo.complet.Complet {
  // ...
}
```

Like any Java class, `Message_` might include data fields that are themselves classes, which can, in turn, include other classes and so on, a structure that is called the *object graph* of `Message_`. When we say "the Message complet", we mean the entire object graph of `Message_`, and we refer to the `Message_` class as the *anchor* of the Message complet. As an example suppose the rest of Message's definition looks like this:

```
public class Message_ implements fargo.complet.Complet {
  private Viewer viewer;
  private String text;

  public Message_(String text) {
    viewer = new Viewer();
    this.text = text;
  }
  public String getText() {
    return text;
  }
}

public class Viewer {
  // ...
}
```

The Message complet (Message's object graph) includes two classes, `Message_` and `Viewer`, where `Message_` is the anchor. The anchor has three special characteristics:

- The anchor functions as the interface for invocations of the complet. In other words, "invoking a complet" means invoking one of the methods of its anchor class.
- The anchor defines the set of objects that moves together, that is, when a complet moves, the entire object graph of the anchor moves.
- The anchor as well as all the objects of its object graph, except for objects that are the anchors of other complets, are always co-located in the same core.

Figure 2.2 illustrates two complets α and β, whose anchors are a and b, respectively. α's object graph is in the left oval, while β's object graph is in the right one. One of the objects of α points to β, namely to β's anchor b, which functions as β's interface. When β moves, its entire graph moves as a whole, yet the reference that points to its anchor from α remains valid.

When we want our program to refer to a complet, and also to instantiate it, we use the type of its anchor, but *without the underscore character*. For example, suppose we want to hold an instance of Message in variable `message`

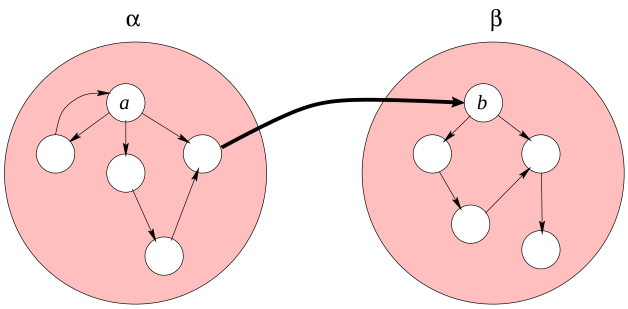

Figure 2.2: Complets, Anchors, and Object Graphs

of a class that is included in the object graph of some other complet. The declaration and initialization of this variable might look like this:

```
Message message = new Message("Hello!");
```

Except for the above difference in the class name, the syntax of invocations, including that of the constructor, is exactly as in regular Java. For example, after initializing the `message` variable we can invoke Message's `getText` method as follows:

```
String s = message.getText();
```

The class we use to refer to a complet, the one without the underscore, is called a *stub*, and is automatically generated during compilation (see Section 2.4 below). This class includes all the public and package-access methods of the anchor with exactly the same signatures and it also implements all the interfaces that are implemented by the anchor.

2.2.1 Complet Invocation and Parameter Passing

Although the syntax of complet method invocations is similar to that of regular Java, there are several important differences in their semantics:

- Only `public` and default-access methods of the anchor can be invoked by another complet.

- The parameters and return values in complet invocations are passed by (complet) reference only if the parameter or return value is an anchor (namely it implements `fargo.complet.Complet`). Otherwise, it is passed by copy.

Notice that when an object is passed by copy, complets that are pointed from inside its object graph are not copied. Rather, the copy of the object graph keeps pointing to these complets with complet references.

2.2.2 Remote Instantiation

As seen above, complets are instantiated like every regular Java object with the `new` operator. `new` creates a fresh instance in the current core, an instance that can later be moved to a remote core.

Sometimes, however, we want a complet to be instantiated *remotely*. One reason could be saving the overhead of movement, while a more obvious one is when local instantiation is impossible, for example, since the instantiated complet needs to natively interact with local special-purpose hardware during instantiation.

Remote instantiation can be done with the static `remoteNew` method of the API class `Core`. `remoteNew` receives an instantiator — an object that insatiates the complet by calling `new` — and a remote location. The remote location can be given either as a core URL (see 6.1.1), or as a complet. Specifying a complet as the location means that the instantiation should take place at that complet's current core. The instantiator's class should implement the `fargo.core.Instantiator` interface, which includes a single method, `instantiate`, that should regularly instantiate the complet by calling `new`.

A very convenient practice is to define the instantiator as an anonymous class (see [1]) at the point where it is needed. Follows an example of instantiating the Message complet at a remote core whose URL is `pooh.technion.ac.il`:

```
CoreURL url = new CoreURL("pooh.technion.ac.il");

Message mssage = Core.remoteNew(new Instatiator() {
  public Stub instantiate() throws CompletInstantiationException {
```

```
    return new Message("Hello!");
  }
}, url);
```

After the call to `remoteNew` returns, `message` contains the new instance that actually resides in the remote core.

2.3 Complet References

A reference that points to a complet is called a *complet reference*. It is important to observe that as a consequence of the special semantics of parameter passing described earlier, *a complet reference always points to an anchor of some complet*. For example, returning to our Message complet, no complet can hold a reference that points to its internal `Viewer`. Each complet reference that points to it actually points to an instance of `Message_`.

Since complets are mobile, the target of a complet reference can at one time be local to the source, and at another time be remote, it can also move from one machine to another while the source is pointing to it. FarGo automatically ensures that the complet reference remains valid in spite of such movements of both its source an its target. In Figure 2.2 the solid arrow (that is contained in one of α's objects and points to β's anchor, b) is a complet reference. When implementing α and programming its interactions with β, these interactions are programmed as regular method invocations through Java reference. In runtime, however, both α and β can move, a fact that does not affect the way invocations are programmed.

2.3.1 Reflection of Complet References

Complet references are *reflective*. That is, a program can ask the system for a meta-reference object that can supply *information about* the reference and allows to *change its behavior*.

For example, let us return to the `message` variable that contains an instance of the Message complet. `message`, in fact, contains a complet reference that points to that instance. Follows an example that uses the meta-reference of `message` for introspection:

```
MetaRef mr = Core.getMetaRef(message);
if (!mr.isTergetLocal())
  CoreURL url = mr.getTargetLocation();
```

The first statement deposits the meta-reference into variable `mr`. The methods of `MetaRef` can now be invoked to ask the meta-reference object questions regarding the complet reference it represents, and also to change the represented reference. The `if` statement that follows asks the meta-reference whether the reference target (the Message complet) is not local, in which case it gets the target's location.

This was an example of using the meta-reference for introspection (namely, asking questions about a complet reference). Chapters 3 and 5 include many examples of using the meta reference also to change the behavior of the reference.

2.3.2 Using the Anchor's `this` Reference

A very convenient possibility in FarGo is to use the anchor's `this` reference like done in regular Java objects. More precisely, if the anchor wants to pass a reference to itself as a parameter to a method, it can simply pass `this`. At first glance, this ability might seem obvious, since every object in Java can pass `this` as a parameter. However, notice that in our case the passed reference will remain valid even when the passed object (the anchor) moves to another core.

Here is an example. Suppose our Message complet needs to be looked up by a meaningful name, say "my_message". It can invoke the core's `bind` method to bind a name to itself. Notice how it specifies `this` to that method:

```
public class Message_ implements Complet {
  public Message_() {
    Core.bind(this, "my_message");
  }
  // ...
}
```

Likewise, a complet may need to pass `this` in an invocation of another complet. The trouble is that the type specified by the invoked method would be that of the stub, while `this` has the type of the anchor (the class name of the stub appended by an underscore character). To resolve this difficulty, the core supplies the `getStub` method that returns a newly created stub for a given complet, a stub that can then be passed in such invocation.

For example, suppose our Message complet receives a Registrar complet in its constructor and should register by passing itself as a parameter:

```
public class Message_ implements Complet {
  public Message_(Registrar reg) {
    reg.register(Core.getStub(this));
  }
  // ...
}

public class Registrar_ implements Complet {
  public void register(Complet c) {
    // ...
  }
  // ...
}
```

After Message's constructor has finished, it can move to another core, yet the complet reference which is contained in Registrar and points to it remains valid.

2.4 Compiling FarGo Programs

FarGo programs are compiled using the special *fargoc* compiler. Fargoc is very similar in its command-line interface to the standard javac compiler, thus, should be easy to use to any Java programmer.

Fargoc can be used either to compile all the classes of the application, or only the anchor classes. If you choose the latter, make sure you first use fargoc, and then javac to compile the rest of the classes.

Before you start using fargoc, see that your environment variables are set correctly (consult FarGo's installation instructions, see the preface of this document for the details). In addition, you should make sure that both the directory in which you want your class files to be placed as well as the directory in which your source file reside are in you CLASSPATH.

Fargoc is basically a source generator, which creates several class files for each one of your complets. After creating these sources, it invokes a standard Java compiler (e.g., javac) and the RMI compiler, rmic, in order to compile these sources and to generate RMI classes (these are needed since the complet referencing mechanism uses Java RMI [8] as its basic RPC mechanism). For each anchor class, fargoc generates the stub (that has the same name as that of the anchor, but without the underscore), and also several other classes. After compilation, there are six class files per each complet:

1. *complet-name*`.class`
2. *complet-name*`_.class`
3. *complet-name*`_Tracker.class`
4. *complet-name*`_TrackerImpl.class`
5. *complet-name*`_TrackerImpl_Stub.class`
6. *complet-name*`_TrackerImpl_Skel.class`

Classes no.1 and 2 are the stub and anchor, respectively. The rest are classes that implement the special characteristics of the complet reference. Classes no. 5 and 6 are the Java RMI classes.

For a full description of how to work with fargoc, please refer to Section 7.2.

Chapter 3

Layout Programming with Relocators

As mentioned before, a basic capability of FarGo is to move complets between cores while the application is running. However, an application is composed from many interacting complets, thus the movement of one complet might affect other complets. For example, assuming that complet α contains a complet reference to complet β and that these two complets were located in the same core since the frequency of invocations of β by α is high. Now, assume that at some point in the application's lifetime, most of the complets that use α happen to reside in core C, thus α is ordered to move to C in order to improve the responsiveness of the application. But if β will be left at its current core, α will start working slower since each invocation of β that was previously local will now become remote. Thus, the gain in responsiveness that was expected in the first place might be harmed. In this example, we could use a rule that would say "keep β always co-located with α", or in other words, whenever α moves, β moves along to the same new location.

This is an example to a design rule that we would like to set in design time, and we would like the system to enforce at runtime. The set of such rules that define the relations between complets with respect to their relocations is named the *layout* of the application. By specifying these rules we abstractly say how should the complets be spread out on the network, without committing where exactly (on which machine) should each complet reside. Notice that we do not specify here which complet should move and to which new location, an issue that is differed to Chapters 4 and 5. We only say that *if* one complet moves, other complets should also move.

We use the complet reference as the entity that holds layout information. That is, suppose α includes a complet reference that points to β, we tell this reference that whenever α moves to a new core, the target, β, should follow. This example in which the relation is that the reference's source and the target should be co-located is only one of a set of possible relations, which are presented in this chapter.

A complet reference has several attributes that can then be set by the program. One of these attributes, named *relocator*, determines the layout relation between the source and target of the reference. For example, we set the above relation between α and β by specifying that the relocator attribute of the reference from α to β should have a certain value. The different types of relations are implemented as different values that we set to the relocator attribute of references.

The rest of this chapter describes the possible layout relations between complets, or in other words, the possible values of the relocator attribute and their meanings.

3.1 Manipulating Relocators

Before we start presenting the various relocators, we shall first show *how* to manipulate them. To manipulate a complet reference, the program should first ask the core for this reference's meta-representation using the core's `getMetaRef` service. A meta-representation is an object that represents the internals of the reference at hand. The methods of this object can be used to manipulate the attributes of the complet reference it represents.

Let us give an example. Follows two complets Alpha and Beta, where Alpha has a data field, `beta` that points to Beta:

```
public class Alpha_ implements Complet {
  private Beta beta;
  // ...
}
public class Beta_ implements Complet {
  // ...
}
```

Now, suppose we want to add a method to Alpha, say `setRelation`, that sets a value to the relocator attribute of the reference in `beta`. Suppose that the value we want to assign is *pull*, which means that whenever Alpha moves, Beta also moves along with it. This is how we do it:

```
public void setRelation() {
  MetaRef metaRef = Core.getMetaRef(beta);
  metaRef.setRelocator(new fargo.core.Pull());
}
```

The first statement of `setRelation` retrieves a meta reference object that represents the reference contained in variable `beta`. The next statement calls this object's `setRelocator` method to set a value to the relocator attribute of the reference. The actual value that we set is a new instance of the `fargo.core.Pull` class.

Notice that the value of the relocator attribute can change while the program is running. For example, we could pass to `setRelation` some parameter that could be used by this method to decide what value to assign to the relocator attribute.

****** partially implemented ******

Each of the relocator classes (e.g., `fargo.core.Pull`) includes various methods that can be invoked to fine-tune its semantics. However, only the `setRelocator` method can actually cause a complet reference to change its behavior, based on the parameter it receives. Consider the following example:

```
MetaRef metaRef = Core.getMetaRef(beta);
Pull pull = new Pull();
metaRef.setRelocator(pull);
pull.setDelay(100);         // Fine-tuning.
metaRef.setRelocator(pull); // Affects the reference.
```

****** setDelay: not available ******

The `setDelay` method causes the movement of Beta to Alpha's new locality to occur after some delay from the time Beta moves. The above invocation of `setDelay` does not affect the complet reference, only the `Pull` instance on which it is invoked. To actually affect the reference, `setRelocator` should be invoked again with the updated `pull` passed as a parameter.

3.2 Introduction to Relocator Types

As presented above, each value of a relocator defines a relation between the source, α, and target, β, of its complet reference. To precisely specify this relation, for each relocator value we must explain how α and β behave regarding two major aspects:

- Co-location — Whether and when should α and β be co-located in the same core, and
- Re-location — How does the movement of one of these two complets affects the location of the other one.

In addition to specifying these two aspects, for each relocator value we shall give a class whose instance should be the parameter to the `setRelocator` method. In the code examples that follow, we assume that we have a complet reference whose source complet is Alpha, whose target complet is Beta, and that it is contained in Alpha's `beta` data fields (as in Section 3.1 above).

Finally, since relocator values can be changed dynamically by the program, we will have to specify what happens when a certain relocator evolves to become a different one.

3.3 Link Relocator

The default value of the relocator attribute is *link*, namely when a complet references is created, its relocator attribute has this value (we say that this reference is a *link relocator*).

The link relocator does not specify any relation between the source and the target of the reference. In other words, movement of one of these complets does not affect the location of the other one.

Thus, if α points to β we get the following relations:

- 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 relocator 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 needs to be performed.

This relocator value is an instance of the class `fargo.core.Link`. Setting our complet reference from Alpha to Beta to function as a link relocator can be done as follows:

```
MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new fargo.core.Link());
```

3.4 Pull Relocator

This relocator is used to ensure co-location between complets; when the source complet relocates, it "pulls" its target along. The relations between α and β are:

- Co-location: α and β are always co-located.
- Re-location: if α relocates, β moves along to the new 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. Thus, If an attempt to move β is made, a `MovementException` is thrown.

Second, β can be referenced by at most one reference that is a pull relocator. Otherwise, if another complet γ points to β with a pull relocator, 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. Thus, if a reference to a complet is changed to a pull relocator when this complet is already pointed with such relocator, the `RelocatorAssignmentException` is thrown.

Notice however, that α can still be a target of a pull relocator, i.e., some other complet δ might point to α with a pull relocator. Pull is transitive, and relocation of δ will relocate α, and hence β.

When a non-pull relocator is dynamically changed to a pull relocator, then if the reference's target is remote, it automatically moves to the same core of the source. Notice that since a movement operation is done here, an exception might be thrown if this movement does not succeed. In such case the `setRelocator` method will throw the `RelocatorAssignmentException`, possibly with a more descriptive exception nested inside it.

A pull relocator 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.

The pull relocator value is an instance of the class `fargo.core.Pull`. Setting our complet reference from Alpha to Beta to function as a pull relocator can be done as follows:

```
MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new fargo.core.Pull());
```

3.5 Duplicate Relocator

The duplicate relocator is similar to the pull relocator, except it is sufficient to ensure co- and re-location with a *copy* (a clone) of β, instead of with β itself.

The relations in this case are:

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

Unlike the pull relocator, there can be multiple duplicate relocator references that point to the same complet. This is possible since if γ points to β with a duplicate relocator 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 relocator and the target of (multiple) duplicate relocators. This is a desirable property, since it does not over-restricts pull relocators by constraining their compatibility with non-pull relocators.

When a relocator is dynamically changed from a non-duplicate into a duplicate relocator, 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. Like with the pull relocator, this movement operation might fail and the `RelocatorAssignmentException` might be thrown.

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.

The duplicate relocator value is an instance of the class `fargo.core.Duplicate`. Setting our complet reference from Alpha to Beta to function as a pull relocator can be done as follows:

```
MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new fargo.core.Duplicate());
```

3.6 Stamp Relocator

In this case, when the source of the reference moves, the reference first gets disconnected from the target before the movement, and then, once the source arrives to its new core, the reference gets connected to a new target that is *equivalent* to the original source.

The relations are, then:

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

The implications on the target complet with respect to relocation and compatibility with other references, are similar to those of duplicate relocator. 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 relocators can co-exist along with a single pull relocator.

Generally, when we say "a new target that is equivalent to the original source" we refer to 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). The protocol by which an equivalent complet is selected at the destination can be determined by the application. This issue is covered below in Section 3.6.1.

Finally, upon dynamic change of a relocator into a stamp relocator, an instance of β's type is located in α's locality and gets reconnected to α.

A useful application of a stamp relocator is for facilitating 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.

The stamp relocator value is an instance of the class `fargo.core.Stamp`. Setting our complet reference from Alpha to Beta to function as a stamp relocator can be done as follows:

```
MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new fargo.core.Stamp());
```

3.6.1 Types of Equivalence

As mentioned earlier, the application can dictate the exact protocol for selecting a new target to be pointed by the stamp relocator after movement. There are four alternative protocols:

- The default protocol arbitrarily chooses a local complet that is an instance of exactly the same class as the source (not a super-class or super-interface). A `Stamp` that currently uses a different protocol can be set to use this one by an invocation of its `setEquivalence` method without any parameters.

- The second protocol arbitrarily chooses a local complet whose anchor is an instance of a given class or interface (or an instance of some sub-classes/sub-interfaces of the given one). A `Stamp` is set to use this protocol by an invocation of its `setEquivalence` method while passing a `Class` object as a parameter.

- The third protocol chooses a local complet that is locally bound to a given name (see `localLookup` in Section 6.2.1). In this case, a `Stamp` is set to use the protocol by an invocation of its `setEquivalence` method while passing the name (a `String`) as a parameter.

- The most flexible protocol allows the application to supply an operator of its own by which a new target is chosen. To use this protocol, first implement the `fargo.core.StampOperator` interface by implementing its `op` method to return the complet that will become the new target. Then, to set a `Stamp` to use it, invoke its `setEquivalence` method while passing an instance of your implementation as a parameter.

We proceed with several examples. To set the complet reference contained by variable `beta` to act as stamp relocator, one should do the following:

```
MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new Stamp());
```

Here, the default protocol will be used. Suppose that variable `beta` is of type `Printer`, then after movement it will get connected to some local instance of `Printer` at the destination core. To tell a stamp relocator to use the protocol that chooses a new target that is bound to the name "laser_printer" we can write:

```
MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new Stamp().setEquivalence("laser_printer"));
```

Finally, this is how we provide our own operator that for choosing a new target at the destination site. We first define the `FreePrinter` class that includes our desired operator and then pass its instance to `setEquivalence`:

```
public class FreePrinter implements StampEquivalence {
  public Complet equivalence() {
    Pool pool = (Pool)localLookup("printer_pool");
    return = pool.getFreePrinter();
  }
}

MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new Stamp.setEquivalence(new FreePrinter()));
```

3.7 Bi-Directional Pull Relocator

****** not available ******

This relocator (referred hereafter as b-pull) is the most powerful, but also most expensive relocator. It is similar to pull relocator with the additional rule that the target can "pull" the source too.

The two relations are:

- 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 relocator — β cannot be pointed by any pull relocator. By definition, this relocator allows the target to move not only as a result of source relocation, so it is less restrictive than the (uni-directional) pull relocator. This in turn allows to remove the singleness constraint of the regular pull relocator as well, i.e., a complet can be the target of multiple b-pull relocators.

The most interesting property of the b-pull relocator is that it effectively defines "group relocation" semantics. The group is defined as the sub-graph of all nodes that are connected via a b-pull relocator, and relocation of *any* member in the group leads to relocation of *all* members of the group. Like with the regular pull relocator, in case of evolution into a b-pull relocator, the target relocates to the locality of the source.

One issue with b-pull relocators 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 clients which are pointed with b-pull relocators 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.

The b-pull relocator value is an instance of the class `fargo.core.BPull`. Setting our complet reference from Alpha to Beta to function as a b-pull relocator can be done as follows:

```
MetaRef metaRef = Core.getMetaRef(beta);
metaRef.setRelocator(new fargo.core.BPull());
```

Chapter 4

Programming Complet Movement

The previous chapter discusses how to program the layout of the application, but intentionally ignores the issue of actually programming the movement of complets. This chapter introduces the movement primitives of the core and shows how can the movement mechanism be specialized by the program for its own specific purposes.

4.1 Mobile vs. Non-Mobile Complets

By default, every complet is mobile. However, some applications might need to include a complet that cannot be moved under any circumstance. For example, a complet that encapsulates a hardware device might include some native device-specific code, thus must be stable.

To declare a complet as stable, its anchor should implement the `core.complet.Stable` interface. An attempt to move such complet will result in a `MovementException`.

4.2 The Movement Primitives

FarGo's movement primitive is the overloaded static method `move` of class `fargo.core.Carrier`. This class has as many as nine `move` methods. The most powerful `move` has the following signature:

```
public static void move(Complet complet,
                        CoreURL url,
                        String method,
                        Object[] params) throws MovementException
```

It receives the complet that should be moved, the URL of the destination, a continuation method that should be invoked automatically after arrival to the destination core, and the parameters for that invocation. All the other methods are variants of this one, where for example, the URL can be given as a `String`, or the `method` and `params` arguments do not appear.

The core moves a complet in the following steps. It begins by serializing [7] the entire object graph of the complet into a stream of bytes. Then, the current core contacts the destination core and passes it the serialized stream. The destination core de-serializes the stream into an object graph and then invokes the continuation method while passing it the parameters in `params`.

There are several delicate issues concerning movement that must be taken into account when designing a FarGo applications:

- After the current core serializes the object graph, the original copy of the moved complet is still lives in memory. Only after the serialized stream reaches the destination core and gets de-serialized, does the original complet gets freed for garbage collection. This means that a (standard) thread that is active inside the moved complet will continue to run while movement is in progress, which in some cases, might result with inconsistency in the application. Moreover, if such thread does not stop before the complet is freed for garbage collection, the garbage collector will not reclaim it as long as this thread is active, although no reference is pointing to that complet. This might result in a "ghost" complet that exists and executes code at the original core while its copy also executes at the destination core.

- The call to `move` returns right after the destination core receives the moved complet, but before the continuation method finishes at the destination core. That is, the return of `move` does not give any indication regarding the status of the call to the continuation method.

- The continuation method must not be overloaded and must not throw any checked exception. Failure to comply with these rules results in a `MovementException`.

- The parameters you pass in the `params` vector behave according to the regular inter-complet parameter passing semantics. That is, if a parameter is an anchor it is passed by complet reference, otherwise it is passed by copy.

- Lastly, a program can move only local complets. Namely, the caller to `move` and the moved complet must reside in the same core. An attempt to move a remote complet will result in a `MovementException`.

Like with every invocation of a core service, a call to `move` can be done from a method of an anchor, passing `this` as a parameter. This enable an autonomous complet to actually move itself, as in the following example.

```
public class Agent_ implements Complet {
  Itinerary i;
  User u;

  public Agent_(Itinerary i, User u) {
    this.itinerary = i; this.u = u;
    doWork();
  }
  public void doWork() {
    doSomething();
    if (i.hasMorePlaces())
      Core.move(this, i.nextPlace(), "doWork");
    else
      u.finished(this);
  }
  public void doSomething() {
    // ...
  }
}
```

4.3 Movement Awareness

FarGo lets the complet programmer to extend and specialize the default behavior of the movement mechanism, in order to adjust it to the specific needs of the application. This can be done with the movement awareness mechanism.

The movement awareness mechanism makes the complet "aware" of its own movement. For example, a visible complet that shows a window on the screen might need to close the window whenever it is moved and reopen it when it arrives to its new location. For that purpose, there are several callback methods that, if implemented by the complet's anchor, are invokes by the core at different points during the progress of the movement procedure.

To implement these callback methods, the anchor should implement the `fargo.core.MovementAware` interface. This interface includes four methods, which are invoked during the movement process in the following order:

- `preDeparture` invoked at the core from which the complet departures at the beginning of the movement procedure before serialization of the complet begins. This method can throw the `MovementStopException` in order to stop the movement (In such case, `Carrier.move` will throw a `MovementException`).
- `preArrival` invoked at the core to which the complet arrives after deserialization, but before the complet is operational at the new site. Again, this method can throw the `MovementStopException` in order to stop the movement (In such case, `Carrier.move` will throw a `MovementException`). Once this method has completed successfully, the movement is considered finished and cannot be stopped.
- `postArrival` invoked at the destination core right after the complet (more precisely, the entire complet graph) was made ready for use. This method is invoked in a separate thread, thus does not block the movement process. Notice that this method is not guaranteed to be invoked. For example, complet α that points to complet β with a pull relocator can change that relocator and move β in its `postArrival`, before β's `postArrival` has started to execute.
- `postDeparture` invoked at the source core right before the obsolete copy of the moved complet is freed for garbage collection.

Notice that since code that is written in these methods actually executes within the movement procedure, it should be written with care. For example, the `preDeparture` method executes when the complet was not yet connected to its new host (since movement has not been finished yet), thus methods that need to interact with the internal inter-complet referencing mechanism of the core (e.g, `Core.getStub` might not be able to perform their tasks.

Another class `fargo.complet.MovementAwareComplet` is included for convenience in case not all these methods needs to be implemented. This class implements the `MovementAware` interface with empty methods, such that to implement only some of the above methods, one should inherit from it and override only those methods that needs implementation.

Let us return to the visible complet example and see how it implements these methods:

```
public class Visible_
         extends MovementAwareComplet
         implements Complet
{
  java.awt.Frame frame;

  public Visible_() {
    frame = new FancyWindow();
    frame.setVisible(true);  // Show on screen.
  }
  public preDeparture() {
    frame.setVisible(false); // Close the window.
  }
  public postArrival() {
    frame.setVisible(true);  // Show again.
  }
}
```

Chapter 5

Specifying Relocation Policies using the Internal Monitor

So far, we have seen both how to move complets and how to specify the overall layout that should hold when movement occurs. A point that is still unclear is under which circumstances should we actually order a complet to move, and to which destinations. In other words, we need to see how to specify the *policy* of relocation.

One of the major reasons for supporting relocation is to gain from local, or "close" interaction between complets by moving them around the network. But how can we determine whether there are good chances that a movement will indeed result in the expected gain? This problem gets harder as the scale of our deployment setting gets larger and includes link of variable bandwidth and of variable reliability, non-stable machines, etc.

FarGo provides an internal monitoring mechanism to aid us in solving this problem. The monitor does two main tasks. It performs performance and resource utilization measurements, for example, the bandwidth of a remote complet reference and the free memory in the current core's process. In addition, it can notify the application of the occurrence of various events, such as the movement of a certain complet, or a (planned) shutdown of the core. The application can use these facilities by basing its movement policy on an examination of the measured values and on responses to the reported events.

There are two programming interfaces to the monitor service. The first interface is an API for interaction with the monitor within the application. The second interface is a high-level scripting language that is specialized for implementing relocation policies. An important characteristic of these scripts is that they be attached to an application or to a core dynamically.

5.1 Profiling

Profiling support is the lowest layer in which (re)location descicsion take place. This layer gives applications the ability to measure resource use like cpu load and free memory and environment parameters like network delay. FarGo application can access the profiling data in a direct or continuous way. In the direct way, the system measures or calculates the desired information on demand. The request for information is the trigger for starting the measurment or calculations. By using this way the application gets the current state ot the current value of the profiled resource. This type of profiling has no lasting overhead on the system. The profiling operation take place on demand and the result is return after it is done. no background operations take place. By using the continuous way of profiling, applications can gets the average value of a profiled resource. The profiling layer holds a list of resources which are profiled priodicaly. A geometrical average is calculated for each profiled resource. Applications can gets the average values to get less accurate results but more steady ones. In order the avoid unneeded overhead applications can add and remove resources from the profiling list, So only needed measurements and calculation will take place.

5.1.1 Complet profiling

Complets as the basic movement entities, has several attributes that change over time. Some of this attributes like the complet size and the rate in which complets interact with other complets are crucial in finding the optimal layout of applications.

The profiling of complets attributes is supported in the synchronous and asynchronous ways, for most of the attributes (see below). The asynchronous profiling is done in a separate thread for each of the attributes. This thread is activated only if there is at-least one application that monitors this type of attribute. When more than two attributes from the same type are been monitored, (e.g. complet size of two different complets) the thread is started and perform the measurement in a cyclic way, so each complet is measured ones in a cycle.

The access to the profiling data of a complet is done throw a `CompletProfiler` which can be retrieved from the `MetaRef` of a complet in the following way:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
```

The following subsections will describe each of the supported attributes.

Complet Size

The size of a complet is significant in the decision of relocation the complet. Relocation of small complets is likely to be more efficient than perform a series of remote invocations while big complet relocation is not worse while even if the complet interacts a lot with other complets.

The current size of a complet (in bytes) can be measured by in the following way:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
long size = profiler.getCompletSize();
```

It is also possible to measure the average size of a complet over a period of time. The measurement is started in the following way:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
profiler.startCompletSize();
```

The measurements will continue to take place on a periodical basis and the current average result can be examined in the following way:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
double size = profiler.getAverageCompletSize();
```

The stopping of the background measurement is done by using the the `stopCompletSize` method:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
profiler.stopCompletSize();
```

Network Delay

Network delay between complets is one of the important parameter to relocation policies. The internal monitor of FarGo gives the ability to measure the network delay between any two connected complets. This measurement can be done directly or continuously in the same way as the complet size measurement.

Measure the current network delay:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
long delay = profiler.getNetworkDelay();
```

Start measuring the average network delay:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
profiler.startNetworkDelay();
```

Obtaining the average network delay:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
double delay = profiler.getAverageNetworkDelay();
```

Stop measuring the average network delay:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
profiler.stopNetworkDelay();
```

Invocation rate

Another important parameter is the volume of communication between complets. Two complets that perform a massive amount of invocation on each other should be located in the same core. The profiling subsystem gives the facility to measure the number of invocation per second over a specific remote reference. This measure can be operate only on the continuous mode since there is no meaning for measuring `rate` at a single point of time.

The start getAverage and stop methods of the invocation rate measurement are used in the following way:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);

// starting the measurement
profiler.startInvocationRate();

// getting the current average result
double delay = profiler.getAverageInvocationRate();


// stop the measurement
profiler.stopInvocationRate();
```

Outgoing references

Each complet can hold references to other complets in the local core or in remote ones. FarGo supplies an introspection method for obtain the list of remote references originate at a specific complet. The list returned by this method is a list of meta references that can be used for further introspection. see Chapter 3 and this Chapter. The outgoing references inquiry is done as follows:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletProfiler profiler = MetaRef.getProfiler(metaRef);
MetaRef[] metaRef = profiler.getOutgoingRefs();
```

5.1.2 Core profiling

Profiling of the core is important for complets and application which want to balance the load and resource utilization between the available cores. Like complets, cores can be profiled from the local core or from a remote one. The profiling operation is based on a `CoreProfiler` — an object that reifies the core profiling data. A reference to the `CoreProfiler` of a core can be obtained in the following way:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler (coreURL);
```

Complet count

The number of complets in a core can be used to determinate the memory load on a core. This attribute should be use carefully since it doesn't take into account the size of the complets, hence a large number of complets can has a smaller memory need as compare to a smaller group of complets. But applications with complets of similar memory demands can use this attribute to figure out which cores are less loaded.

This attribute an be accessed in the direct way:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler (coreURL);
long count = profiler.getCompletCount();
```

Or it can be used in the continuous way:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler (coreURL);
```

```
// starting the measurement
profiler.startCompletCount();

// getting the current average result
double delay = profiler.getAverageCompletCount();


// stop the measurement
profiler.stopCompletCount();
```

Thread count

The thread count attribute like the complet count gives a sense of the load on a core. This attribute measures the load of the cpu. A core with a large number of threads running in it is expected to have low performance characteristic. This attribute combined with the complet count attribute can give a good sense of which cores are the most busy and which one are more relax and can handle more complets more efficiently. The use of this measurement is very similar to the previous one.

The direct way:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler (coreURL);
long count = profiler.getThreadCount();
```

and the continuous way:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler (coreURL);

// starting the measurement
profiler.startThreadCount();

// getting the current average result
double delay = profiler.getAverageThreadCount();


// stop the measurement
profiler.stopThreadCount();
```

Total and Free memory

The total size of the memory allocated to the virtual machine and the amount of free memory out of it can be a good indication to the ability of the core to host complets. Note that the garbage collector is not called before the checking of the free memory, causing the amount reporter to be possibly smaller than the real amount of free memory in the virtual machine.

The total and free memory attributes can be monitored in the direct and in the continuous ways using these methods:

```
long totalMem = profiler.getFreeMemory();
long freeMem = profiler.getTotalMemory();

profiler.startFreeMemory();
profiler.startTotalMemory();

long totalMem = profiler.getAverageFreeMemory();
long freeMem = profiler.getAverageTotalMemory();

profiler.stopFreeMemory();
profiler.stopTotalMemory();
```

Complets IDs and names

Each complet in FarGo has a unique ID and can have an attached name (see chapter 6). Application can introspect a core (actually the `CoreProfiler`) to get a list of its known complets.

The application can get the list of names binded to complets in the profiled core. The names in the list are bounded to complets that are known to the local core. These complets can be in the profiled core or in any other core, But in any case the profiled core has a (remote) reference to the bounded complet. The list of names can be obtained in the following way:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler(coreURL);
String[] names = profiler.getCompletsNames();
```

The application can also get the list of complet IDs known to the profiled core. This list contains the IDs of *all* complets known to the core. The application can also request a list of complet IDs of the local complets only. These requests are done as follows:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler(coreURL);

// all complets IDs known to the core
long[] IDs = profiler.getCompletsIDs();

// local complets IDs known to the core
long[] IDs = profiler.getLocalCompletsIDs();
```

Network delay

Applications might want to measure the network delay between two cores before relocation complets in order to estimate the effect of the relocation on the application performance. FarGo gives the application the ability the measure the network delay between the profiled core and any other core in the system. This measure is available only in the direct form in the following way:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreProfiler profiler = Core.getProfiler(coreURL);
long delay = profiler.getNetworkDelay ("piglet.technion.ac.il");
```

5.2 Events

In the previous subsection we show how to use FarGo to profile the system and the application in a synchronous way. The application trigger a measurement of an attribute or requests a continuous measurements and pools the average result from time to time to track changes. In the section we will present a distributed event model that allows application the ability to get notification in an asynchronous way about changes in the attributes of the system or of the application.

The distributed event model used by FarGo is very similar to the event model of Java. The registration to the events is done using methods in the form `addXXXListener`. However there are two difference from the regular model.

First, the `addXXXListsner` methods returns a unique listener ID which is used by the corresponding `removeXXXListener` instead of the listener in the traditional event model. The second difference is the ability to supply a predicate object together with the listener to the `addXXXListsner` method. The event will be presented to the predicate (if one was specified) prior to firing the event to the listener. The predicate has the chance to abort the event and spare unneeded events from the listener.

This mechanism is important since the predicate is executed in the core where the event is constructed which may be different from the core where the listener is located, so the network bandwidth is not wasted on unneeded events. Listeners of events can be regular Java objects or complets, this will impact on the location where the listener call-back method will be invoked. In the case of regular Java object, the listener will be copied and invoked in the core were the event happened. In the other case, where the listener is a complet, The call-back method will be invoked at the core where this complet is located at the time the event is fired.

5.2.1 Profiling events

Profiling events are events that reflects changes in a profiled attribute. These events are divided into two groups. The first one are events that reflects changes in a core attribute like the number of complets or the amount of free memory. and the second type are more application specific like the network delay changes between two complets or the rate of invocation over a specific remote reference.

Registration to core events is done using the `CoreEventReg` interface. A reference to an object that implements this interface can be obtained as follows:

```
CoreURL coreURL = new CoreURL ("pooh.technion.ac.il");
CoreEventReg eventReg = Core.getEventReg(coreURL);
```

This interface contains the following methods:

```
public Interface CoreEventReg
{
    long addCompletCountListener (CompletCountListener l);
    long addCompletCountListener (CompletCountListener l, Predicate p);
    void removeCompletCountListener (long ID);

    long addThreadCountListener (ThreadCountListener l);
    long addThreadCountListener (ThreadCountListener l, Predicate p);
    void removeThreadCountListener (long ID);

    long addFreeMemoryListener (FreeMemoryListener l);
    long addFreeMemoryListener (FreeMemoryListener l, Predicate p);
    void removeFreeMemoryListener (long ID);

    ...
}
```

Note that each `addXXXListsner` method has two forms. One with the listener as the only parameter, one that has two parameters the listener and a predicate. The predicate will be described in details at the end of this section.

Registration to complets events is done throw the meta reference of complets since this events reflects changes in the attribute of the remote reference or the target complet. An interface called `CompletEventReg` is obtained from the meta reference in the following way:

```
MetaRef metaRef = Core.getMetaRef(complet);
CompletEventReg  eventReg = MetaRef.getEventReg(metaRef);
```

This interface (`CompletEventReg`) contains the needed methods for registration to the events:

```
public Interface CompletEventReg
{
    long addInvocationRateListener (InvocationRateListener l);
    long addInvocationRateListener (InvocationRateListener l, Predicate p);
    void removeInvocationRateListener (long ID);

    long addNetworkDelayListener (NetworkDelayListener l);
```

```
    long addNetworkDelayListener (NetworkDelayListener l, Predicate p);
    void removeNetworkDelayListener (long ID);
}
```

5.2.2 System events

System events are events that are used to inform the application about changes in the state of the core. The registration to these events is done throw the `CoreEventReg` mentioned before.

```
public Interface CoreEventReg
{
    long addShutdownListener (ShutdownListener l);
    long addShutdownListener (ShutdownListener l, Predicate p);
    void removeShutdownListener (long ID);

    long addCompletsArrivedListener (CompletsArrivedListener l);
    long addCompletsArrivedListener (CompletsArrivedListener l, Predicate p);
    void removeCompletsArrivedListener (long ID);

    long addCompletsDepartureListener (CompletsDepartureListener l);
    long addCompletsDepartureListener (CompletsDepartureListener l, Predicate p);
    void removeCompletsDepartureListener (long ID);

    long addCompletFreedListener (CompletFreedListener l);
    long addCompletFreedListener (CompletFreedListener l, Predicate p);
    void removeCompletFreedListener (long ID);

    long addCompletConstructedListener (CompletConstructedListener l);
    long addCompletConstructedListener (CompletConstructedListener l, Predicate p);
    void removeCompletConstructedListener (long ID);

    long addCompletCountListener (CompletCountListener l);
    long addCompletCountListener (CompletCountListener l, Predicate p);
    void removeCompletCountListener (long ID);

    long addThreadCountListener (ThreadCountListener l);
    long addThreadCountListener (ThreadCountListener l, Predicate p);
    void removeThreadCountListener (long ID);

    long addFreeMemoryListener (FreeMemoryListener l);
```

```
    long addFreeMemoryListener (FreeMemoryListener l, Predicate p);
    void removeCompletConstructedListener (long ID);
}
```

5.2.3 Event objects

Each of the listeners interface (`XXXListener`) is an interface with one method called XXX with parameter of the type XXXEvent. For example, the `ShutdownListener` interface look like this:

```
public interface ShutdownListener extends EventListener
{
    public void shutdown (ShutdownEvent e);
}
```

The event objects carries the information of the event. There are three major types of events objects each one of them carries a different type of information. Table (??) shows the partitioning of the different event object to the three events types.

All the event object implements the method `getSourceCore` which returns the `CoreURL` of the core which fired the event. The specific information can be obtained from the event objects according to theirs basic type:

- `ValueEvent` : This events objects are carries numerical values about the last and average value of the associated attribute. The values can be access using the following methods `getValue()` and `getAverage()`.
- `CompletEvent` : Events that can involve one complet only. Reference to the complet that caused the event can be obtained using the method `getComplet`.
- `completsEvent` : Events that can involve more than one complet. Array of references to the complets can be obtained using the method `getComplets`.

In addition to these methods the `CompletsArrivedEvent` implements to method `getOriginCore` which returns the `CoreURL` of the origin core of the complets that just arrived to the core that fired the event. And the `CompletsDeparturedEvent` implements to method `geDestinationCore` which returns the `CoreURL` of the target destination of the complets that just leaved the core that fired the event.

5.2.4 Predicates

A predicate can be used by listeners to filter unneeded events. Each listener in the FarGo system has an predicate that get the chance of blocking the event from the listener. The predicate is implemented by the interface `fargo.event.Predicate` that contains one method : `predicate` which get an event object as a parameter and returns a boolean that indicates if the event should be pass to the listener.

FarGo provides the following predicates and the application programmer can implements the `Predicate` interface to add new ones.

- `AlwaysPredicate` : This is the default predicate. This predicate returns true always, in other words all the events will be forwarded to the listener.
- `OverPredicate` : Used with events that extends `ValueEvent`. Events with value *over* given threshold will be forward to the listener. After the first event is forwarded all events will be blocked until the value of the event will go under the threshold and over again. The threshold is given to the predicate in the constructor : `OverPredicate(int threshold)`.
- `UnderPredicate` : Like the `OverPredicate` but the event is fired when the value of the event goes *under* the given threshold.
- `DeltaPredicate` : This predicate will forward events to the listener if the value in the `eventObject` is out of given range around the value of the last event forwarded to the listener. The range is given by two numbers. The first one should be in the range [0..1]. a value of 0.6 for examples means that the event will be forwarded if the value falls under 60% of the last forwarded value. The second number should be greater than 1. a value of 1.6 for example means that the event will be forwarded if the value climbs over 60% of its last forwarded value. This numbers are passed to the predicate using its constructor : `DelatePredicate(int min, int max)`.

5.3 The Scripting Language

***** not available in the current release *****

Chapter 6

The Naming Service

When programming distributed system it is sometimes convenient, and many times required to be able to bind a meaningful name to a component. For example, public Web servers have a meaningful name, for example `http://www.dsg.technion.ac.il`, that the user should specify in the browser in order to connect to that server.

Likewise, FarGo allows to bind names to cores and complets and then to locate them when their name is given. This chapter describes how to use this mechanism.

6.1 Core Naming

Each core in FarGo can have associated with it a logical name in the form of a URL. A core URL has the following structure:

host-name **:** *port-number* **/** *local-name*

Where:

- *host-name* is the name or IP number of the host on which this core is running.

- *port-number* is the port number to which this core is listening. The default port number is 3099, and
- *local-name* is a name that distinguishes between different cores that run on the same host. The default value of this field is `core`.

The following two sections show how to bind a URL to a core and how to refer to a core once its URL is known.

6.1.1 Binding a Name to a Core

When activating a core using the `fcore` command from the system's shell while not specifying any parameters, all the fields of the URL will be the default ones. For example, if we execute the command on host `pooh.technion.ac.il` the URL will be:

```
pooh.technion.ac.il:3099/core
```

Each one of the default values in a core URL need not be specified. Thus, the above URL is equivalents to the following three URLs:

```
pooh.technion.ac.il/core
pooh.technion.ac.il:3099
pooh.technion.ac.il
```

When starting a core with the `fcore` command, a *port-number* and *local-name* that are different from the default can be set with `fcore`'s command line parameters. For more information, see Section 7.3.

As shown in 2.1.2, the core can be started from within a Java program either implicitly or explicitly with the `start` method. Once it starts within the program, it does not have any name associated with it. Associating name to the current core from within the program, can be done with the two `daemon` methods of the `Core` class. One of these methods receives a port number and a local name, while the other one uses the defaults. Notice that if the core is not active within the current JVM when `daemon` is invoked, this method also activates the core.

As an example, let us show how the `fcore` command is implemented. It is a simple Java program that receives command line parameters, parses them and then starts a core by calling `daemon`:

```
public static void main(String[] args) {
  CoreArgs ar = parseArgs(args);

  // Check arguments validity
  // ...
  try {
    Core.daemon(ar.port, ar.name, ar.force);
  } catch (Exception e) {
    System.out.println(progName+": exiting due to exception:\n");
    e.printStackTrace();
    System.exit(1);
  }
  // ...
}
```

6.1.2 Referring to a Core by Name

A core name is represented with an object of class `fargo.core.CoreURL`. When constructed, this object can either accept the three elements of the URL separately (the *host-name*, *port-number* and *local-name*), or the whole URL as a `String`, in which case the string is parsed and divided into the three elements. This object can then be given as an argument to API methods that expect a core URL. Notice that some of these methods are overridden to accept URL as either a `CoreURL` object, or as a `String`.

For example, the following example constructs a `CoreURL` for the core named `pooh.technion.ac.il:7000/management` and then moves a complet to that core.

```
CoreURL dest = new CoreURL("pooh.technion.ac.il", 7000, "management");
Complet manager = new ManagementAgent();
Core.move(manager, dest);
```

6.2 Complet Naming

Like with cores, a complet can also have associated with it a name in the form of a URL. A complet URL has the following structure:

core-url#*complet-name*

Where:

- *core-url* is the URL of the core in which the complet was bound to a name, and
- *complet-name* is a name that distinguishes this complet from all other complets that were bound in the same core.

For example, suppose a complet was bound to the name `manager` in core whose URL is `pooh.technion.ac.il:7000/management`, the URL of this complet will then be:

`pooh.technion.ac.il:7000/management#manager`

Notice that the rules regarding the default elements of the *core-url* hold in this case as well, that is, if any one of these elements is the default, it need not appear.

An important and convenient characteristic of complet URLs is that they remain valid even when the complet to which the URL is bound moves to another core. Notice, however, that the *core-url* part of the complet URL must still be that of the core in which that complet was bound.

A complet is not limited to be bound to a name in a single core. After arriving to a new core, the complet can be bound again to another name, where the new name will be different from its current name since the *core-url* of the new core is different from that of the old one. Thus, a complet can actually be bound to multiple names at once.

Notice that the above naming scheme is limited in the sense that it includes a static network location in it, while the named entity — the complet — changes its network location dynamically. Moreover, to locate a complet by its name the core in which this complet was bound to that name must be active, which might not be acceptable in certain cases. An example is when the mere purpose of mobility is to move complets away from machines that should shut down for repair. Section 6.3 presents FarGo's *location independent* naming scheme that solves this problem. However, for many applications the more simple location-dependent naming scheme might suffice.

6.2.1 Binding a Name to a Complet

Binding a name to a complet is done with the `bind` and `rebind` methods of the APIs `fargo.core.Core` class. Both methods receive a complet and a name, and bind the name to the complet in the current core. The difference is that `rebind` overrides any existing binding of the name, while `bind` throws the `AlreadyBoundException`.

The following example instantiates a Manager complet, and binds the name `manager` to it:

```
Complet c = new Manager();
try {
  Core.bind(c, "manager");
} catch (AlreadyBoundException e) {
  // ...
}
```

In this example, the complet was bound “from outside” by another class that has instantiated it. Sometimes, it is useful to bind a name from inside the complet. For example, when a complet functions as a server that should never be garbage collected, but rather wait for invocations from remote sites (e.g., a Web server complet), it might be useful to bind a name to it in the constructor. This is done in the following example.

```
public Server_ implements Complet {
  public Server_(String name) {
    Core.bind(this, name);
  }
  // ...
}
```

6.2.2 Referring to a Complet by Name

After a name has been bound to a complet, the complet can be looked up by that name, in other words, its name can be mapped to a complet reference that points to it. Two methods of the class `fargo.core.Core` does that mapping: `lookup` and `localLookup`. `lookup` receives a full complet URL and contacts the core that is designated by the *core-url* element of that

URL in order to lookup the complet. `localLookup`, on the other hand, receives only the *complet-name* element of the complet URL and looks for a complet to which that name was bound in the current core. Notice that the complet to which that *complet-name* is bound can reside remotely, yet the bind operation that has bound that name to a complet must have been done in the current core. That is, while `localLookup` examines only the list of names that are known to the current core, complets to which these names are bound can reside remotely.

The following example instantiates a complet and bounds a name to it. This is followed by code of another complet that looks up that complet and invokes one of its methods.

```
// In core dsg.technion.ac.il
Complet c = new Viewer();
Core.bind(c, "viewer");

// In another core
Viewer viewer = (Viewer)Core.lookup("dsg.technion.ac.il/#manager");
viewer.showText("Hi Guys");
```

6.3 Location Independent Naming

****** not available ******

Chapter 7

Development and Deployment Tools

This chapter describes FarGo's programming and deployment environment that consists of a set of tools. Information on these tools is also available from two sources. The first source is the FarGo Toolkit Documentation that is bundled with the distribution. In addition, for seeing a summary of the option for a tool, start the tool while specifying the `-help` command-line option.

7.1 Invoking the Tools

There are two methods to invoke each one of the tools. The first method is using a script that actually starts the tool (a shell script on Unix, or a batch file on Windows). For example, to invoke the fargoc compiler with this method, you should use a command such as this:

```
prompt> fargoc *.java
```

The second method is to explicitly run the Java interpreter, while specifying the command class as its argument. For example, to execute the same compilation task, you should use the command:

```
prompt> java fargoc *.java
```

Notice that there are several advantages in using the second method. First, you can specify flags to the java interpreter that runs the tool. For example, if you want Java to use more memory that its default, you can use a command such as this:

```
prompt> java -mx120M fargoc *.java
```

Second, on Windows the scripts allow you to pass no more than nine arguments to a tool. When you run the compiler, for instance, you might want to specify more than nine files to be processed. By invoking fargoc using the second method, you are limited only by the maximal length of your console's command line.

Finally, if you run FarGo on a platform other than Unix or Windows you must use the scond method, since the scripts are good only for these two platforms.

7.2 `fargoc` — The FarGo Compiler

The fargoc command compiles the source code of FarGo applications into Java bytecode.

The fargoc complier replaces the standard java compiler, e.g., javac. It is a source-to-source converter that generates several classes per each complet, and then invokes the standard Java compiler to compile them, and then rmic to generate RMI stubs and skeletons.

In a FarGo program, all complet class names must ends with '_'. The compiler verifies this by checking that all the classes whose name ends with '_' (and only them) implement the fargo.complet.Complet interface. Although fargoc can be used to compile all the classes of an application, it might be convenient to use it only for the complet classes (the ones that ends with the '_') and let javac compile all the other classes. Moreover, if the `-keepgenerated` option is specified, which causes fargoc to keep the generated java sources, javac can be used to compile all the classes in each edit-compile-debug cycle, including the ones that were generated by fargoc, if only the interfaces of the complet classes were not changed. Whenever such interface is changed, first run use fargoc (with the `-keepgenerated` option) to compile the complet class and only then javac to compile all the other classes. This can be useful when working with an IDE which uses its own built-in java compiler.

Full documentation on fargoc is available as part of the distribution at `/FarGo/docs/tools/fargoc.html`.

7.3 `fcore` — The Core Activator

This tool starts a core on the current machine. This core functions as a daemon that listens for requests such as to receive a complet from a remote core. A daemon core can be remotely administered using the FarGo Shell and the Graphical Monitor (which is not yet available).

Once activated the core gets a name in the form of a URL, that has the following structure: ¡host-name¿:¡port-number¿/¡local-name¿ where ¡host-name¿ is the name of the host on which the fcore command is executed, ¡port-number¿ is the port number to which this core listens, and ¡local-name¿ is a name that distinguishes this core from other cores on the same machine that are listening to the same port.

For example, a core that is started on host dsg.technion.ac.il, port 7000, and is assign the local name manager will have the URL: `dsg.technion.ac.il:7000/manager`.

Full documentation on fcore is available as part of the distribution at `/FarGo/docs/tools/fcore.html`.

7.4 `fshell` — The FarGo Shell

This shell is used to contact remote cores, to manage them and to start applications on them. Its interface is very much similar to command-line shells of known operating systems, such as a Unix shell.

After entering the shell, a prompt is shown at the left side of the console and shell commands can be typed to its right, where pressing ENTER issues a command. The prompt shows the URL of the core to which the shell is currently connected. For example the prompt `dsg@fshell>` shows that the shell is connected to core dsg (the full URL is `dsg.technion.ac.il:3099/core`). When the prompt is fshell@fshell, the shell is not connected to any core.?

Full documentation on fshell is available as part of the distribution at `/FarGo/docs/tools/fshell.html`.

7.5 `fmonitor` — The Graphical Monitor

****** not available ******

Bibliography

[1] K. Arnold and J. Gosling. *The Java Programming Language 2nd Edition*. Addison-Wesley, Reading MA, December 1997.

[2] O. Holder. The design of the FarGo system. Technical Report EE Pub No. 1171, Technion — Israel Institute of Technology, February 1998.

[3] O. Holder and I. Ben-Shaul. Dynamic layout of distributed applications. In *Proceedings of the 3rd International Software Architecture Workshop, in conjunction with ACM SIGSOFT'98*, pages 77–80, Orlando, FL, November 1998.

[4] O. Holder, I. Ben-Shaul, and H. Gazit. Dynamic layout of distributed applications in FarGo. In *Proceedings of the 21th Internationsl Conference on Software Engineering*, Los Angenes, CA, May 1999. To Appear.

[5] O. Holder, I. Ben-Shaul, and H. Gazit. System support for dynamic layout of distributed applications. In *Proceedings of the 19th International Conference on Distributed Computing Systems (ICDCS '99)*, Austin, TX, May 1999. To appear.

[6] 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`.

[7] Sun Microsystems, Inc. *Java Object Serialization Specification*. Available at `http://java.sun.com/products/jdk/rmi/doc/serial-spec/serialTOC.doc.html`.

[8] 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`.

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