A graphical deployment and management tool for distributed applications*

Anders Andersen and Thomas Aanensen

Department of Computer Science
University of Tromsø, Norway
aa@cs.uit.no,thoaan@online.no

Abstract. OOPP is a component based middleware platform with support for complex distributed applications. The main goal of OOPP is to create an expressive programming model for distributed applications where by default details are hidden for the programmer. When necessary, reflection is used to expose and sometimes modify these details. All interaction with an OOPP component are specified by its component model and a at set of well-defined interfaces. The component model specifies how a component interacts with the runtime. The set of interfaces specifies how a component interacts with other components. An OOPP application is created by combining a set of OOPP components. OOPP Studio can be used to create and manage such an application. This paper gives an overview of the OOPP programming model and discusses in more details OOPP Studio. OOPP Studio is a powerful graphical tool used to build, deploy, and manage a distributed OOPP application.

1 Introduction

Middleware is used to help programmers to create distributed (and complex) applications [1, 2]. It presents a set of high-level programming abstractions hiding the details of distributed programming and the platform services (i.e. providing transparencies). Software components makes it possible to develop different parts of the application independently, and then later combine and deploy them for a specific application setup. The main motivation for this is to have manageable code (code with well defined functionality and well defined interfaces) that can be easily reused.

OOPP (Open-ORB Python Prototype) [3] is a middleware platform supporting software components. It has an expressive (high-level) programming model inspired by the ISO Reference Model for Open Distributed Processing (RM-ODP) [4].

One consequence of high-level programming abstractions is that the middleware providers have made a lot of decisions on behalf of the programmer about the behavior of the provided abstractions. This might not be feasible for the

* This work has been supported by the Norwegian Research Council (IKT 2010, project number 146986/431).

requirements of a given application or a given context. An example could be an implementation of remote method invocation (RMI) using a transport protocol that does not handle disconnection. Such an RMI implementation would not work in a mobile setting where disconnection happens frequently. To expose implementation details and adapt the system at runtime OOPP provides an open implementation through the concept of reflection [5].

A distributed application in OOPP is a composition of software components connected with bindings. Components can be located in different address-spaces and on different nodes. The reflective mechanisms of OOPP makes it possible to inspect and adapt this composition at runtime. Components can be added, removed, replaced or altered. Bindings can be established, removed or modified.

OOPP Studio [6] is a powerful graphical tool used to build, deploy and manage distributed OOPP applications. The four steps in the process of creating distributed applications with OOPP Studio are:

1. Develop OOPP components
2. Combine components to an application
3. Deploy the application components
4. Manage the running application

In this paper we will present the programming model of OOPP, the tool OOPP Studio, and the role of reflection in this development platform. We will in each step use a distributed 'Hello World!' example to illustrate the process. Some examples of OOPP Studio usage will be presented and discussed. Finally, OOPP Studio will be compared with other approaches and a conclusion will be made.

2 OOPP programming model

An OOPP application consists of a set of software components deployed in one or more capsules. Each component has a set of interfaces. OOPP provides three kinds of interfaces. Signal interfaces are used for one-way signals. A signal interface is either the source or the sink (receiver) of a signal. Signals are typically produced when certain events occur. Stream interfaces are used for continuous media like audio and video. A stream interface is also either a source or a sink. Operational interfaces are used to access and/or provide a set of methods. An operational interface exports (provides) and/or imports (requests) a set of methods.

An interface can be connected (bound) with a binding to a matching interface. An operational interface i_1 matches an operational interface i_2 if i_1 exports the methods i_2 imports, and vice versa. In OOPP, a binding can be explicit. Complex bindings are composite components that might consist of components in different address-spaces and on different nodes. A composite component is represented by a component graph, where its contained components are the nodes and the bindings between these components are the edges. The external

interfaces of a composite component are mappings to a subset of the interfaces of its contained components.

Components are deployed in capsules. A capsule is a managed address-space providing a set of services used to manage the life-cycle of components. Each node has a node manager. The node manager manages all the different capsules on a given node.

An interface can be registered (with a name) in a name server. A component requesting a given interface (with a name) can use the name server to get a reference to the registered interface. With this interface reference a binding can be created to the provided services. In this example the name server provides the bootstrapping mechanism of a distributed application. The name server (or actually the location of the name server represented by an address) is the common knowledge between the different autonomous components of the distributed application.

Below, another approach to bootstrap a distributed OOPP application will be presented. This approach is based on OOPP Studio.

3 OOPP Studio explained

OOPP Studio will be explained using an example. The example is a distributed 'Hello World!'. It is implemented in Python[1] using OOPP programming abstractions.

Step 1: Components

The distributed 'Hello World!' contains two components, a client `hello` and a server `world`. Figure 1 and 2 list the Python code of the two modules implementing these two components.

In OOPP the interfaces of a component are defined in the `interfaces` attribute. This attribute is a Python dictionary[2] where a valid key is the name of an interface and the value of that key is a two-tuple describing the exported and imported methods of the interface (the first element of the tuple is the exported methods and the second element is the imported methods). The elements in the two-tuple are dictionaries where the keys are method names and their values are three-tuples defining the signature of the methods. The first element of the signature is the arguments of the method, the second element is the return value of the method, and the third element is a list of exceptions that the method might generate.

[1] `http://www.python.org/`

[2] Python provides three sequence types. A dictionary is a sequence with key-value pairs. If a dictionary `p` has the value `{"age":23,"name":"Bob"}`, then `p["age"]` is `23`. A list is a sequence of values like `[1,4,0]`, and a tuple is an immutable list (e.g. `(1,4,0)`). Both lists and tuples are indexed by numbers (starting at zero).

```
from oopp.core.component import *
from types import *

from world import world
hello = {"mhello": (
    (StringType, IntType), StringType, [AttributeError])}

class Hello(Component):
    interfaces = {
        "ihello": (hello, {}), "iworld": ({}, world)}
    def mhello(self, name, num):
        msg = ""
        for i in range(num):
            resp = self["iworld"].mworld(name)
            msg += "%d: %s\n" % (i, resp)
        return msg
```

Fig. 1. The class implementing the 'Hello World!' client component (`hello.py`). A `Hello` component export the interface `ihello` with the method `mhello`, and it imports the interface `iworld` with the method `mworld`. The external `mworld` method is called using the interface `iworld` in the implementation of the `mhello` method.

```
from time import ctime
from oopp.core.component import *
from types import *

world = {"mworld": ((StringType,), StringType, [])}

class World(Component):
    interfaces = {"iworld": (world, {})}
    def mworld(self, name):
        if name == "":
            return "Hello World (%s)!" % (ctime(),)
        else:
            return "Hello %s (%s)!" % (name, ctime())
```

Fig. 2. The class implementing the 'Hello World!' server component (`world.py`). A `World` component export the interface `iworld` with the method `mworld`.

In Figure 1 we have a `Hello` component with two interfaces `ihello` and `iworld`. The methods `ihello` exports are described in the value `hello`. Interface `ihello` exports the method `mhello` that takes two arguments (a text string and a number), returns a text string, and might produce an `AttributeError` exception. An `AttributeError` is produced when the interface `iworld` is accessed (in the method `mhello`) before it is bound to a matching interface. The methods `iworld` imports are described in the value `world` defined in the module `world`. Interface `iworld` imports the method `mworld` that takes a text string argument, returns a text string and does not produce any exceptions.

From Figure 2 we can see that a `World` component has one interface `iworld`. This is an interface matching the interface `iworld` in a `Hello` component (it exports the methods the other interface imports).

When the `Hello` and `World` component implementations are loaded in OOPP Studio the tool will parse the classes and recognize their interfaces (see Figure 3 in the Appendix). The reflective features of OOPP makes it possible to inspect and modify this implementation in OOPP Studio (e.g. adding interfaces). It is even possible in OOPP Studio to load a standard Python class and populate it with interfaces to create an OOPP component. The tool creates a component template based on the Python class and the user's description on how to map this class to an OOPP component implementation. In the current example the component is used without any modifications of the original implementation from Figure 1.

After a component implementation has been loaded the tool can be used to create one or more such components. These components can then be combined with other components to create a distributed application.

Step 2: Application

An OOPP application is described as a set of capsules, their containing components, an the bindings between the components. OOPP Studio can be used to create all those building blocks, and it does not enforce any order that they have to be created. You can either create all the capsules first and then populate them with components, or you can create all the components first and then later distribute those components among the set of capsules. You can also create a set of capsules and later decide on what node those capsules should execute on. At any time you can save your current work and terminate OOPP Studio. Later, you can start OOPP Studio, load your earlier work, and continue working on the application. As seen later, a subset of your application can be running while you are still working on it.

A typical scenario based on the distributed 'Hello World!' application could be as follows. First, two capsules are created. They will be called the client capsule and the server capsule. The client capsule will execute on `localhost`, and the server capsule on `ablab11` (see Figure 4 in the Appendix).

Then, the component classes are loaded and one of each is created in the OOPP Studio drawing area (see Figure 5 in the Appendix). You should control that the expected interfaces are recognized and visible in the tool. By selecting a

component it is also possible to see the exported and imported methods of each interface of a component. In the example, a `Hello` component named `hello` and a `World` component named `world` are created.

Now, the components can be located to a given capsule. This is in OOPP Studio simply done by dragging the component to a given capsule. The `hello` component is moved to the client capsule, and the `world` component is moved to the server capsule. Their `iworld` interfaces are connected by dragging one on top of the other (the T-shaped drawings attached to a component are its interfaces). An implicit binding between them will be created if they match (see the implicit binding between the `iworld` interfaces in Figure 6). A local binding is created if the interfaces are located in the same capsule. Otherwise, a remote implicit binding matching the interfaces will be created. This binding is implemented as a composite component, but since it is an implicit binding it is not visible in OOPP Studio. The middleware platform has decided on what binding implementation to use. The difference between a local binding and a remote binding is transparent for the user. If the user wants to influence the selection of a binding implementation an explicit binding can be created. All implicit bindings can be replaced by explicit bindings.

Step 3: Deploy

When OOPP Studio is used to draw an application as described above, no capsules, components and bindings are actually created. A running capsule is created when it is activated. This is an explicit action in OOPP Studio. Components are also only drawings in OOPP Studio until they are activated. A binding between two interfaces is activated when their components are activated. It is possible to activate only a subset of the capsules and the components currently drawn in OOPP Studio. A capsule can only be activated when it is decided on what node it should be executed on. A component can only be activated if it is deployed in an activated capsule.

Every node that should be populated with OOPP capsules have to have a core OOPP implementation installed and a OOPP node manager running. The node manager will create and activate capsules upon requests from OOPP Studio. When a component deployed in a capsule is activated its implementation is transfered from a component implementation repository managed by OOPP Studio.

The capsules of the example application are activated by selecting them and pressing the activate button . This will actually create and start the capsules on their nodes, and the deployed components will be activated in their capsules. Before this explicit is done, the client and server capsules, the `hello` and `world` components, and the binding between the `iworld` interfaces, only exists as an internal representation in OOPP Studio.

At any time we can save the current application, quit OOPP Studio, and then later load and continue working on the same application. This means that a saved application can be either non-activated (no capsules and components

activated), partly activated, or completely activated (all capsules and all components activated). When an application is loaded in OOPP Studio the tool will inspect and connect to the activated capsules and components.

In the 'Hello World!' example no name server has been used. This is not necessary since the tool has the complete view of the application and can do the bootstrapping needed to locate and connect the different autonomous components and capsules. A name server can be used to include existing capsules or components (services) in an OOPP application. If existing capsules or components (created with or without OOPP Studio) providing a set of interfaces have been announced to an OOPP name server, OOPP Studio can be used to locate them. The user of OOPP Studio has to provide the necessary information to locate the name server and the name of the capsule or interface that will be used in the application. An interface identified in this way can be bound to another matching interface. A capsule identified in this way can be browsed for components and their interfaces. Unbound interfaces can be bound to other matching interfaces.

By default, such capsules and components will in OOPP Studio not be managed as part of the current application. The provided interfaces will only be external services used by this application. If it is a capsule it is possible to try to[3] include it and its components in the current application. If successful, the capsule and its components will be viewed and can be operated on just as all the other capsules and components in the application. Later when the application is saved this capsule and its components will be included.

Step 4: Manage

When an application is up and running it has to be maintained. OOPP Studio can be used to manage all aspects of a running application. Examples could be to distribute the load on more nodes (by moving components or capsules), update the implementation of a component (to correct bugs or make it more efficient), or to add more functionality. If you load a running application in OOPP Studio then all operations described in step 1–3 can be done on this application. You can add and remove capsules and components, you can break or create new bindings, and you can even modify existing components. This makes it possible to use OOPP Studio with applications that evolves over time.

4 The role of reflection

The reflective features of OOPP makes it possible to inspect and modify components. An example discussed above was to add interfaces to objects created from a standard Python class. Reflection provided by OOPP can be used to monitor and modify the components and capsules of a running application. A good

[3] This article does not discuss any security issues and access control. Therefore, the discussion on when this will succeed is ignored.

example is an explicit binding. An explicit binding is a composite component including sub component that are marshalling and unmarshalling the contents of messages (arguments and return values). Those components can be replaced by other implementations introducing a different behavior for this underlying service (e.g. encryption or compression of the message content).

5 Conclusion and future work

OOPP Studio has proven to be a powerful graphical tool matching the expressiveness of OOPP. It makes it possible to deploy and maintain a distributed application running on a large number of different nodes from a single desktop computer. The flexibility given by the reflective mechanisms provided by OOPP makes it possible to create a tool that can be used to inspect and modify all aspects of a distributed application.

Future versions of OOPP Studio will extend this functionality even further giving access to a more complete set of reflective functionality found in OOPP. This includes more possibilities to manipulate components (e.g. replace or add method implementations), composite components (manipulation of the component graph), bindings, and capsules.

More specific contracts between a component and its runtime is another thing that we would like to add to OOPP Studio and the OOPP component model. Such a contract should include a description of the support needed by a component. An example could be a specific service (and even a specific version of that service) provided by the middleware platform or a third party library provider. Another example could be a set of Quality of Service properties (e.g. processing resources per time unit for a given component).

References

1. Bernstein, P.A.: Middleware: A model for distributed system services. Communications of the ACM **39** (1996) 86–98
2. Eckerson, W.W.: Three-tier client/server architecture. Open Information Systems **10** (1995) 3–22
3. Andersen, A.: OOPP, A Reflective Middleware Platform including Quality of Service Management. Dr. sci. thesis, Department of Computer Science, University of Tromsø, Tromsø, Norway (2002)
4. ISO/IEC: Open distributed processing reference model, part 1: Overview. ITU-T Rec. X.901 — ISO/IEC 10746-1, ISO/IEC (1995)
5. Smith, B.C.: Procedural Reflection in Programming Languages. PhD thesis, Massachusetts Institute of Technology (1982)
6. Aanensen, T.: OOPP studio: Et verktøy for dynamisk utplassering og distribusjon av komponentbaserte applikasjoner. Masteroppgave, Institutt for Informatikk, Universitetet i Tromsø (2005)

Appendix

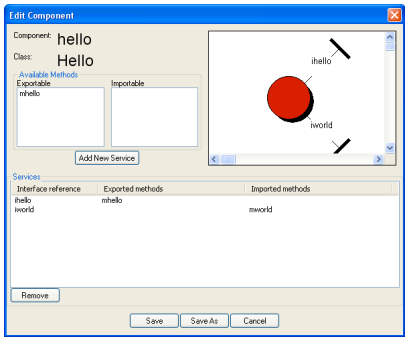

Fig. 3. When the implementation of `Hello` components is loaded in OOPP Studio, the component edit view shown above will be visible. The user can use this view to browse the identified the interfaces of the component class. This view can also be used to modify the component and its interfaces.

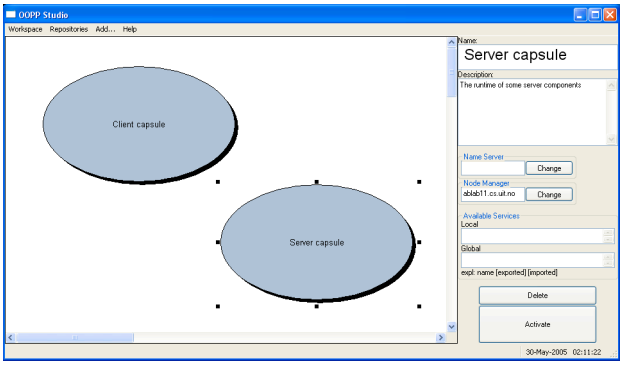

Fig. 4. The user has added two capsules in the drawing area. The server capsule is selected and we can see that is will be activated on the node `ablab11`.

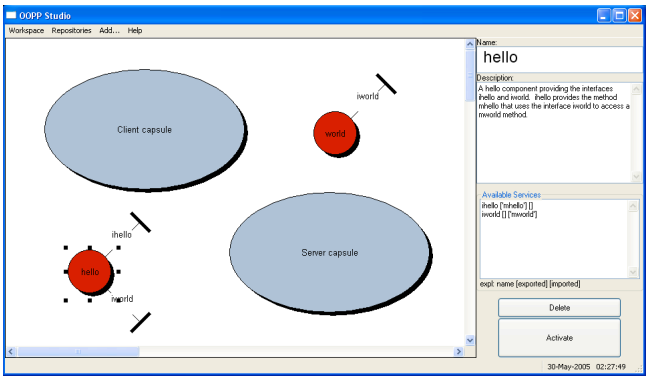

Fig. 5. A component `hello` and a component `world` have been added to the application. The components are not yet deployed in a specific capsule. Component `hello` is selected and it has two interfaces `ihello` and `iworld`. The `ihello` interface exports the method `mhello`, and the `iworld` interface imports the method `mworld`.

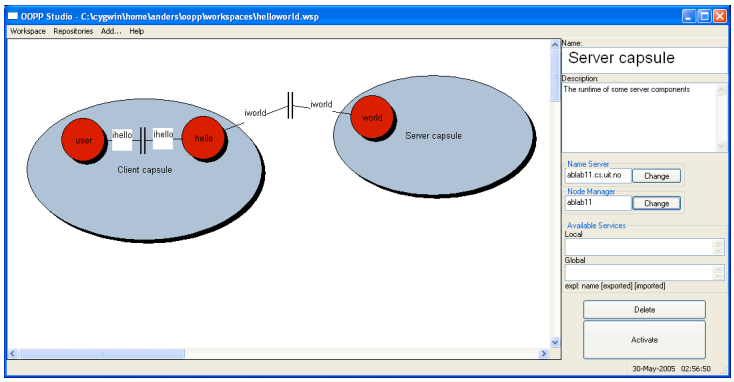

Fig. 6. The components are deployed in their specific capsules and implicit bindings are created between the interfaces. A component `user` has been added. It has an interface `ihello` bound to the `ihello` interface of the `hello` component.