A Middleware-Independent Model and Language for Component Distribution

Hans Albrecht Schmid
University of Applied Sciences Konstanz
Brauneggerstr. 55
D 78462 Konstanz
xx49-(0)7531-206-631 or -500
schmidha@fh-konstanz.de

Marco Pfeifer
University of Applied Sciences Konstanz
Brauneggerstr. 55
D 78462 Konstanz
xx49-(0)7531-206-631 or -500
mpfeifer@fh-konstanz.de

Thorsten Schneider
University of Applied Sciences Konstanz
Brauneggerstr. 55
D 78462 Konstanz
xx49-(0)7531-206-631 or -500
thosch@fh-konstanz.de

ABSTRACT

A distribution model for the new generation of component languages, which allows to plug required into provided interfaces, is presented together with the distributed component language CompJava. The distribution model models an abstraction layer on top of existing component languages and middleware. It makes distributed component programming with CompJava as simple as local programming, allows to compose components from other components scaling up very efficiently, and allows for dynamic allocation and configuration of components. The efficiency of the generated Java source code is an important characteristic of the new component distribution model: it involves no invocation overhead and practically no storage overhead for a distributed component composed from collocated components.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Contructs and Features – concurrent programming structures

General Terms

Languages, Performance, Design.

Keywords

Component, Component Distribution, Middleware, Component Distribution Model, Distributed Component Language

1. INTRODUCTION

A component defines both provided and required interfaces: a provided interface defines the services a component supplies; a required interface defines the services a component needs to get supplied to perform its responsibilities. However, with the classical distributed components models, like Enterprise JavaBeans, CORBA, Corba Component Model and DCOM, "even required interfaces are not normally available" as Szyperski [Sz97] states. Their programming model is based on passing and handling references to provided interfaces; nearly no interface plug-in is possible.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SEM 2005, September 2005, Lisbon, Portugal
Copyright 2005 ACM, 1-59593-204-4/05/09 …$5.00.

A new generation of recently developed, non-distributed component languages, like CompJ [SC00], ArchJava [ACN02a] [ACN02b] and ACOEL [Sr02], defines a component model that allows to plug together ports with provided and required interfaces. This is more natural and easier than reference handling, avoids the proliferation of references to components, and the component structure mirrors clearly the application architecture.

We present (in section 2) a language-independent model for the distribution of the new component generation, called DisComp (Distributable Component model). DisComp models an abstraction layer on top of existing local component languages and middleware (see Table 1), which makes distributed programming very simple and does not impose any distribution overhead. This is in contrast to classical distributed component models, like e.g. Enterprise JavaBeans [EJB01], CORBA [CACM98, Se98] and DCOM [Sz97], which impose for local use a high memory footprint overhead, and either a considerable invocation overhead or, like EJB 2.0, doubled interfaces with a differing semantics.

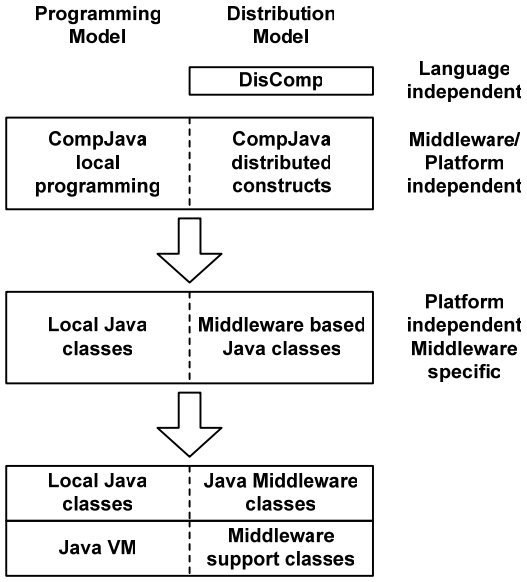

Figure 1 Transformation of local programming model and distribution model from middleware- and platform independent form into middleware- and platform specific form

The distributed component language CompJava presents a platform- and middleware-independent approach for distributed component programming, since it is based on Java and abstracts from the middleware that is used to implement the distribution. It

adds only two declarative constructs to the local version of CompJava, some main features of which are described in section 3. The distribution features of CompJava, which are described in section 4, make distributed programming practically as simple as local programming, and allow for the dynamic allocation and configuration of components

The compiler transforms the distribution constructs of CompJava into Java classes that depend on the support of some middleware platform. It is possible to exchange the transformation to a middleware platform, like Java RMI, which is implemented so far, against the transformation to another middleware platform that has a Java runtime, like CORBA or .NET. Our experience with the compiler construction lets us suppose that it would be a relatively small effort.

An important aspect of the presented component distribution model is that it imposes no distribution overhead at all for the composition of a non-distributed component from remotely accessible components, so that CompJava components scale up very efficiently. The composition of a distributed component is also very efficient since it involves no invocation overhead and practically no storage overhead it is composed from collocated components. Section 5 presents the implementation of central local features and of distribution-related features. Section 6 reviews shortly related work.

2. DISTRIBUTABLE COMPONENT MODEL DISCOMP

The **DisComp** distributable component model defines component types, ports with provided and required interfaces, called shortly provided and required ports, and components. The signature of a port p, p = (n, k, I) is given by: the name n; the kind k which is provided or required, or possibly multicast; and its interface I. An interface is defined as a set of triples of operation names, signatures and return types. We use a function notation to get the i-th port p of a type τ, and the name, kind and interface of a port p: $p = port_i(\tau)$, n = name(p), k = kind(p), I = I(p).

A **component type** τ = (π, R) is a pair where π defines the set of port signatures: $\pi = \{p_1, p_2, ..., p_n\}$; the **remote accessibility** R is of Boolean type. A **component type is remotely accessible** (shortly: **remotable**), if a component of the type may collaborate with components that are allocated on remote nodes. In this case, a restriction like serializability is imposed on all interfaces I(p) for all ports p defined by τ. That means, all parameters and result types of all operations must be accessible over the network.

The requirements with regard to remote accessibility are as follows**:**

R1: remotely accessible and non-remotely accessible component types should be identical, except for minimal restrictions that may be imposed for remote accessibility.

R2: remotely accessible component types should have the same semantics for local and remote collaboration.

R3: if an operation of a port interface is invoked in a local collaboration, a local method call should be performed. That excludes remote interfaces (like RMI) as port interfaces, since these imply remote invocation overhead also for a local call.

A remotely accessible component type τ meets requirements R1 to R3, if it has local interfaces $I = I(port_j(\tau))$ for j=1,...,n, that expose (i.e. have operations with parameters and result types) only remotable types. In Java, these are primitive types and serializable reference types, both with copy semantics; if component types are allowed as parameter or result types, there are also remotable component types with reference semantics. Remote access may be done via proxies.

A **component** implements a component type τ, same as a class implements an interface. A component (τ, D, C) is characterized by its type τ, by the **distribution** D of Boolean type which indicates whether the implementation is distributed, and by its composition C from subcomponents. The (rough) **composition C** is defined by a set of subcomponents, neglecting the connections etc. That means, C is a set of triples consisting of the subcomponent reference name, the component name and the component type:

C(τ) = {<name, component name, component type>}.

Distribution is the property of a component. A **component is distributed**, if it is composed from subcomponents that may be allocated on network domains (i.e. node and virtual machine) different from the component domain. The component allocation A is a mapping from a subcomponent reference name to the domain where the subcomponent is to be allocated.

		Component type	
		Non-remotable	remotable
Component implementation	distributed		② ③ ④
	local	①	②

Table 1 Combinations of the remote accessibility and distribution characteristics.

There are four combinations of the remote accessibility characteristic and the distribution characteristic (see Table 1):

1. A **completely local component** is of a non-remotable type and has a local implementation. **Requirement R4** is that it should have no distribution overhead at all, neither with regard to memory footprint nor to invocation cost.
2. A **remotable component** is of a remotable type. Its implementation may be local or distributed. **Requirement R5** is that if a remotable component is used in a non-distributed environment, there should be no distribution overhead at all.
3. A **distributed component** is composed from remotable subcomponents that may be allocated to different network domains. It may have itself a local or a remotable type. The allocation may be bound at compile time or at deployment time.
4. A **completely distributed component** is remotable and has a distributed implementation.

3. COMPJAVA OVERVIEW

This section gives an overview about local language constructs of CompJava, which are of interest with regard to distribution. It presents a local version of the running example, which is a *CustomerAndAccountAdministration* component. It is composed from the subcomponents *CustomerView*, *CustomerHandling*, and *Customer* (see Figure 2). *CustomerHandling* combines several use cases like enter and edit customer data. It collaborates with *Customer* over the *out2*-port to get and set the customer data; and with *CustomerView* over the *out1*-port to set the presented or get the entered view data, and over the *in*-port to be notified of user input events.

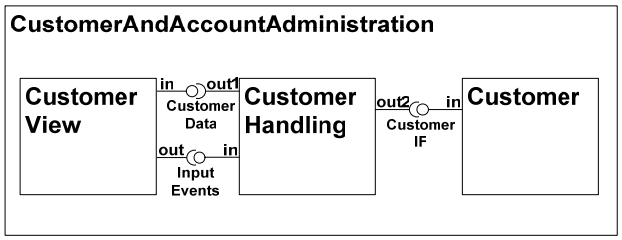

Figure 2 *CustomerAndAccountAdministration* component with subcomponents

CompJava distinguishes component types and components. Each component is of a component type; subtype and similar relationships are defined among component types. There is no inheritance between components. One may create with the new-operator any number of component instances from a component.

A component type defines the provided and required ports with Java interfaces (which do not include references of a component type). For example, the component type *CustomerAndAccount-AdministrationType* is the type of a top-level component that has no ports. The component type *CustomerHandlingType* defines a port *in* that provides the operations of the *InputEvents* interface, and ports *out1* and *out2* with the required interface *CustomerData* resp. *CustomerIF* (see Figure 3).

```
component type CustomerAndAccountAdministrationType { }

component type CustomerHandlingType {
  port in provides InputEvents;
  port out1 requires CustomerData;
  port out2 requires CustomerIF;
}
```

Figure 3 Top-level component type *CustomerAndAccount AdministrationType* without ports, and component type *CustomerHandlingType* with provided port *in* and required ports *out1* and *out2*.

A component is of the component type indicated by the **ofType**-clause (see Figure 4); it implements the provided interfaces, using possibly operations of required interfaces.

```
component CustomerHandling ofType CustomerHandlingType{
  //port in provides InputEvents;
  //port out1 requires CustomerData;
  //port out2 requires CustomerIF;
  ...                    //implementation with e.g. an inner class
}
```

Figure 4 a Component *Customer Handling*

```
component CustomerAndAccountAdministration
            ofType CustomerAndAccountAdministrationType {
  CustomerViewType theCustomerView = new CustomerView();
  CustomerHandlingType theCustomerHandling =
                                  new CustomerHandling();
  CustomerType theCustomer = new Customer();
  connect theCustomerHandling.out1 to theCustomerView.in;
  connect theCustomerView.out to theCustomerHandling.in;
  connect theCustomerHandling.out2 to theCustomer.in;
  //initialization, main method etc.
}
```

Figure 4 b Component *CustomerAndAccountAdministration*, connecting ports of the *theCustomerView*, *theCustomerHandling*, and *theCustomer* subcomponent instances

A bottom-level component like *CustomerHandling* (see Figure 4 a) contains classes or code that implement provided ports or call the operations of a required port. How that is done is not the topic of our paper. But let us mention shortly, that e.g. an inner class might implement the *InputEvents* interface of the provided port *in*, and might call operations defined in the *CustomerData* interface of the required port *out1* to get the data a user has entered. Then it might call operations defined in the *CustomerIF* interface of the required port *out2* to put the data in a customer component.

A higher level component, like *CustomerAndAccount Administration*, is composed from subcomponents, like *CustomerView, CustomerHandling,* and *Customer* (see Figure 4 b). If it has provided interfaces, it may use provided interfaces of subcomponents or own code to implement them. Similarly, if subcomponents have required interfaces, a component may use its required interfaces or own code to implement them.

A connect-statement like

```
connect theCustomerHandling.out2 to theCustomer.in;
```

plugs a required port of a subcomponent instance, like *out2* of *theCustomerHandling,* in a provided port of (usually) another subcomponent instance, like *in* of *theCustomer*, binding each operation of the required interface to the matching operation of the provided interface. When e.g. *theCustomerHandling* invokes an operation defined in the port *out2*, the matching operation provided by the port *in* of *theCustomer* is executed. Port matching ensures at compile time that the provided interface provides, at least, all operations defined by the required interface.

4. LANGUAGE CONSTRUCTS FOR DISTRIBUTION

This section presents distribution-related language constructs of CompJava that follow the DisComp distribution model. CompJava allows defining remotely accessible component types with the same semantics for remote and local access, and distributed components. We use a distributed version of *CustomerAndAccountAdministration* with subcomponents of a remotable type as an example.

4.1 Remotely Accessible Component Types

A **remotely accessible component type** (as indicated by the keyword ”remotable”, see Figure 5) has local port interfaces which must expose (i.e. have operations with parameters and result types) only remotable Java types, i.e. primitive types and serializable reference types. A component of a remotable type is used in a local, i.e non-distributed environment exactly in the same way as one of a non-remotable type. Thus, requirement R1 is met.

```
remotable component type CustomerHandlingType {
  port in provides InputEvents;
  port out1 requires CustomerData;
  port out2 requires CustomerIF;
}
```

Figure 5 Remotable component type with port interfaces that expose only remotable types

In order to encapsulate networking failures like RMI remote exceptions, we generate either events, via a singleton, for which an application may attach itself as a listener, or application level exceptions (which is also recommended by the Business Delegate pattern [ACM2001]).

4.2 Distributed Components

A **distributed component** has a distributed implementation. The keyword distributed in a component declaration (see Figure 6) indicates that the subcomponents may be allocated to domains different from the component domain. The allocation may be bound at compile-time or at deployment-time.

```
distributed component CustomerAndAccount Administration
                ofType CustomerAndAccountAdministrationType {
  CustomerViewType theCustomerView = new CustomerView();
  CustomerHandlingType theCustomerHandling =
                                    new CustomerHandling();
  CustomerType theCustomer = new Customer();
  connect theCustomerHandling.out1 to theCustomerView.in;
  connect theCustomerView.out to theCustomerHandling.in;
  connect theCustomerHandling.out2 to theCustomer.in;
  //initialization, main method etc.
}
```

Figure 6 Distributed component

For the deployment-time binding which we present, an allocation table in XML-format gives the allocation domain (i.e. virtual machine and network node) of each subcomponent. The CompJava compiler does not have or use a knowledge of the allocation. The collaboration among components is optimized at subcomponent allocation time such that local method calls are executed over connected ports of two subcomponents, if these are collocated (i.e. allocated on the same domain); and remote method invocations, if the subcomponents are remote (i.e. allocated on different domains).

All subcomponents of a distributed component must be of a remotable type (shortly: be remotable). The semantics of a remotable component is exactly the same for local and remote access, i.e. when used as a subcomponent of a local or a distributed component. Thus, requirement R2 is met.

The remotability and distribution characteristics are orthogonal, following DisComp:

- A distributed component may have either a non-remotable or a remotable component type.
- A component of a remotable component type may itself be local or distributed.

5 IMPLEMENTATION OF COMPJAVA

The CompJava compiler translates CompJava components into plain-vanilla Java classes, and in Java classes that have dependencies on the middleware used. We sketch the translation of local, remotable and distributed components, using RMI as a simple middleware, and show how the DisComp requirements are met.

5.1 Component Interface and Core Class

CompJava generates both for non-remotable and remotable component types a Java interface (see Figure 7). It defines port access methods; that is, for each provided port a getPort-operation which returns a reference to the provided port, and for each required port a setPort-operation which sets the required port to a reference.

CompJava generates for local and remotable components a local class called ”core” class (which may contain other classes). A core class has a member variable of the port type for each port (see Figure 7); it implements the port-access methods of component type interface, and it contains local code for subcomponent creation, connect-statement execution etc. Thus, requirement R4 is met.

```
interface CustomerHandlingType {
  InputEvents getPortIn();
  void setPortOut1( CustomerData out);
  void setPortOut2( CustomerIF out);
}
class CustomerHandlingCore
                        implements CustomerHandlingType {
  private InputEvents in;
  private CustomerData out1;
  private CustomerIF out2;
  public InputEvents getPortIn() { return in;};
  public void setPortOut1( CustomerData out) {out1 = out; }
  public void setPortOut2( CustomerIF out) {out2 = out; }
  ...
}
```

Figure 7 Generated interface for a component type and generated component core class

Figure 8 illustrates the implementation of a bottom-level component, like *CustomerHandling*. An inner class of the core class may implement the *InputEvents* interface of the provided *in*-port, and the port variable *in* may reference an instance of the inner class. The inner class may call operations defined in the interface of the required *out1*-or *out2*-port via the port variable *out1* resp. *out2*.

It is not allowed to distribute code or classes that implement a component; only subcomponents may be remotely allocated.

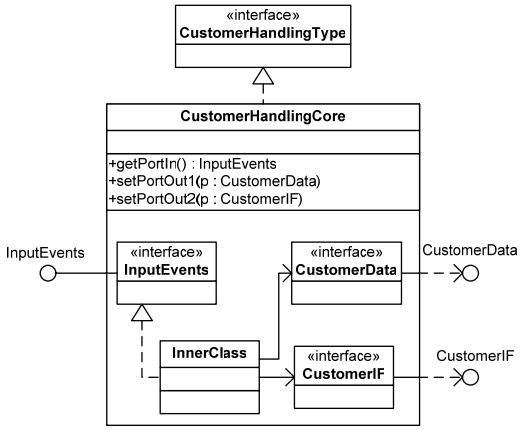

Figure 8 Core class with inner class and interfaces for port member variables

When a (non-distributed) component is composed by subcomponents, the parent component allocates the core classes of the subcomponents with the new-operator and constructor of the corresponding core classes, so that the generation of the Java source is straightforward.

CompJava avoids the overhead that is caused by connector classes generated for each connect-statement e.g. with ArchJava [ACN02a]. It generates for a connect-statement like

connect *theCustomerHandling.out2* **to** *theCustomer.in*

code that assigns the port-reference of the provided port to the port-variable of the required port:

theCustomerHandling.setPortOut2(theCustomerView.getPortIn())

In this way we accomplish that code within *theCustomerHandling* component, which invokes an operation of the required port *out*, calls directly the matching method provided by the *in*-port of *theCustomerView*, so that there is no invocation overhead.

5.2 Remotable Core, Remote Port and Port Proxy of a Remotable Component

CompJava generates for a component of a remotable type exactly the same core class as for one of a non-remotable type, and additionally a remotable core class that extends the core class and implements transitively the component type interface (see Figure 9). A remotable core class, like *CustomerHandling RemotableCore*, is a local (i.e. non-RMI) class that has provisions to create facilities for its remote access.

CompJava provides remote access to a remotable core class by a remote access structure (see Figure 9) that is generated for each provided port. The port *in* of the *CustomerHandlingRemotable-Core* class has the provided *InputEvents* interface. A remotely accessible class like *RemotePortIn* (which implements a RMI Remote interface, and extends the class UnicastRemoteObject) forwards remote invocations by a local call to the remotable core class; and a proxy class, like *PortInProxy*, implements the local interface and forwards local calls remotely to the remotely accessible class. A remotable core class, like *CustomerHandling-RemotableCore*, extends the corresponding core class for each provided port by a method, like *createRemotePortIn*, which creates a remote port (see Figure 9).

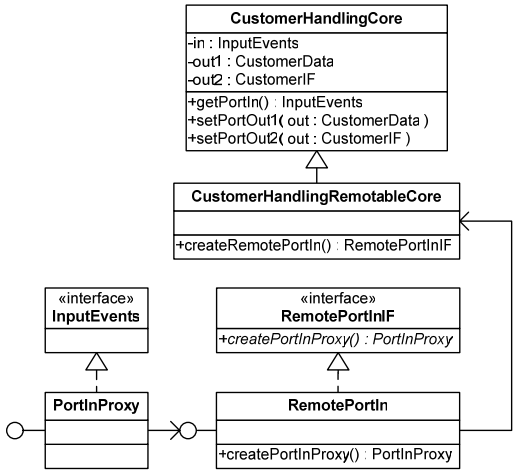

Figure 9 Remotable core class extending the core class and remote port class and proxy class.

A remote port class, like *RemotePortIn*, implements a RMI remote interface, like *RemotePortInIF*. A remote port interface has operations that match those of the local port interface, like *InputEvents*, with the same semantics except for the remote exceptions. This is possible due to the restriction for remotable component types that all port interfaces must expose only remotable types. A remote port class forwards method calls to the remotable core class; it has additionally auxiliary methods, like *createPortInProxy()* that allows to create a port proxy.

A proxy, like *PortInProxy*, is a serializable class that implements the local port interface, like *InputEvents*, and has a reference to the remote port class, like *RemotePortIn*, forwarding method calls via this remote reference.

If a remotable component is a subcomponent of a non-distributed component, only its core class is used to implement the subcomponent. Therefore, a component of a remotable component type has in a non-distributed environment exactly the same performance characteristics (i.e., no overhead at all) as one of a non-remotable component type. Thus, requirements R3 and R5 are met.

5.3 Allocation of Distributed Subcomponents

When the execution of a component, like *CustomerAndAccount-Administration*, is started, its subcomponents are allocated. A dynamic allocation during the execution is possible if required.

Non-distributed components invoke the new-operator with the constructor of the corresponding core class as described, whereas distributed components use a distributed naming structure for components, called CompNaming for the allocation of subcomponents. Thus, CompJava generates for a distributed component the following Java code for subcomponent creation:

```
theCustomer = (…) CompNaming.create( theCustomer, ...);
theCustomerView = (…) CompNaming.create(
                                    theCustomerView, ... );
theCustomerHandling = (…) CompNaming.create(
                                    theCustomerHandling, ... );
```

An instance of CompNaming is running on each network domain with distributed components. CompNaming provides, in principle, two create-operations, a local operation: *static Object create(String compName, ...),* and a RMI remote operation, *SetupProxy create(String compName, ...) throws RemoteException.* A distributed component, like CustomerAndAccountAdministration, invokes always the local create-operation of the local CompNaming. The local create-method checks in the component allocation table, whether the component instance is to be allocated on its local domain.

- If so, it performs the creation and allocation of the remotable core of the corresponding component instance and returns the local reference to that instance.
- Otherwise, it invokes the remote create-operation of CompNaming on the domain where the component instance is to be allocated, which returns a serializable SetupProxy of the corresponding component.

The remote create-method of CompNaming invokes the local create-method which returns a local reference. Then it creates a RemoteSetup instance and a SetupProxy for the component (which form a similar remote access structure for components as described for ports), sets the references among them and returns the serializable SetupProxy as a result.

In this way, CompNaming returns optimized references to collocated components: a returned reference to a collocated component is always a local reference, whereas a returned reference to a remote component is formed by a SetupProxy that remotely references a RemoteSetup that references a remotable core class.

Figure 10 shows the implementation of the *CustomerAndAccount Administration* component after the allocation of its subcomponents via CompNaming. Suppose *CustomerRemotable Core* is allocated on the same domain, and *CustomerView RemotableCore* and *CustomerHandlingRemotableCore* are allocated on a different domain. Then, *CustomerAndAccount AdministrationCore* has a local reference to the collocated *CustomerRemotableCore*, and local references to the *CustomerViewSetupProxy* and *CustomerHandlingSetupProxy* (which reference *CustomerViewRemoteSetup* and *Customer-*

HandlingRemoteSetup, which reference in turn *CustomerView-RemotableCore* and *CustomerHandlingRemotable Core)*.

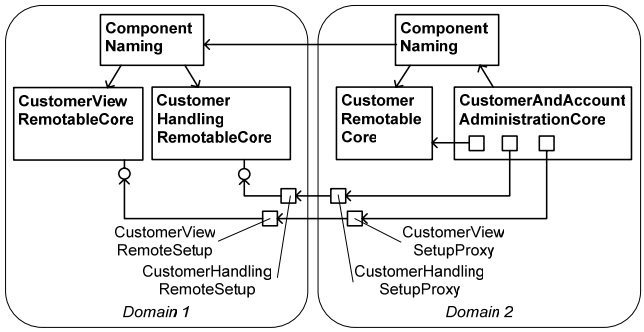

Figure 10 Implementatin of distributed *CustomerAndAccount Administration* component after the allocation of its subcomponents with remote acces structures, and CompNaming

This is a simple example for the automatic collocation optimization that is done when performing the appropriate reference transformations.

5.4 Connecting Ports of Distributed Subcomponents

This section presents the execution of connect-statements as a more complex example for automatic collocation optimization by reference transformation.

The getPort- and setPort-operations of a remotable core class, RemoteSetup and SetupProxy perform automatically reference transformations. Consider the reference transformation performed by a getPort-operation. It returns:

- a local reference to the corresponding port, when the receiver variable contains a local reference to the component remotable core;
- a PortProxy, when the receiver variable references a SetupProxy.

Similarly, a local receiver of a setPort-operation accepts a local reference to a port as a parameter and stores the local reference, whereas a SetupProxy as a receiver transforms a local reference into a PortProxy before forwarding it via the RemoteSetup to the component core class, which stores it.

Consider now the execution of a connect-statement in the parent component, like

connect *theCustomerHandling.out2* **to** *theCustomer.in*;

The CompJava compiler generates always the same code for the parent component:

theCustomerHandling.setPortOut2(theCustomer.getPortIn());

independently on where the subcomponents are allocated. The execution of that code performs an automatic collocation optimization, depending on whether the receiver variables contain a local component reference or a reference to a SetupProxy.

With the allocation assumptions made above, the receiver variable *theCustomer* contains a local component reference, so that the *getPortIn*-operation returns a local reference to the *in*-port. The receiver variable *theCustomerHandling* contains a reference to a SetupProxy of the *CustomerHandling* component. The *setPortOut2*-method of the *CustomerHandlingSetupProxy* transforms the local reference into a local reference to *CustomerPortInProxy* that references *CustomerRemotePortIn*, as shown in Figure 11, and forwards it via *CustomerHandling-RemoteSetup* to *CustomerHandlingRemotableCore*, which stores the reference to the *CustomerPortInProxy* in its *in*-port, as Figure 11 shows.

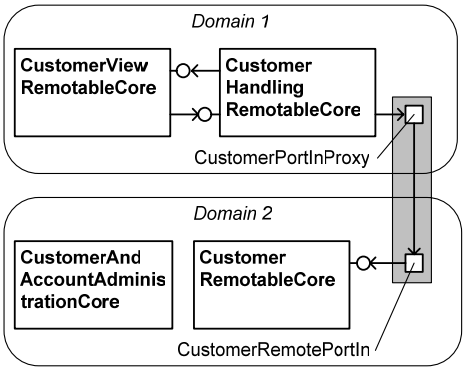

Figure 11 Distributed *CustomerAndAccountAdministration* component with connected subcomponents (i.e. after execution of the connect-statements)

The automatic collocation optimization performed for the connect-statement

connect *theCustomerHandling.out1* **to** *theCustomerView.in*;

where the CompJava compiler generates in principle the same Java source as shown:

theCustomerHandling.setPortOut1(theCustomeView.getPortIn())

delivers a different result, since the *theCustomerHandling* and *theCustomeView* components are collocated, but both remote to the parent component, and perform therefore different reference transformations.

In this case, the receiver variable *theCustomerView* contains a reference to a *CustomeViewSetupProxy*, so that the *getPortIn*-operation returns a reference to a *PortInProxy*. The receiver variable *theCustomerHandling* contains a reference to *CustomerHandlingSetupProxy*. Its *setPortOut1*-method passes the *PortInProxy* reference to *CustomerHandlingRemoteSetup*. It transforms in the *setPortOut1*-method the ”remote proxy” reference into a local reference to the *in*-port of *CustomerViewRemotable-Core*, and forwards it to the *CustomerHandlingRemotableCore* that stores the local reference, as Figure 11 shows.

Summarizing, a RemoteSetup and a SetupProxy of a component perform an automatic collocation optimization during the set-up when connecting the ports. Its result is that

- for collocated subcomponents (see left domain of Figure 11), a required port contains always a local reference to the implementation of the provided port, independently on the allocation (remote or collocated) of the parent component
- for non-collocated subcomponents (see *Customer-Handling* and *Customer* subcomponents of Figure 11), a required port contains a reference to a PortProxy of the provided port.

This is an optimal result with regard to memory and invocation overhead in a distributed environment:

- Collocated subcomponents with connected ports have practically no distribution overhead since they have the same invocation efficiency and practically the same memory footprint as completely local components.

- Remotely allocated subcomponents with connected ports have an increased memory footprint, due to the created RemotePort and PortProxy instances. Their invocation overhead is determined by the remote invocation overhead of the used middleware; the overhead of two additional local calls is, in comparison, negligible.

6. RELATED WORK

Classical component models, like Enterprise JavaBeans [EJB01] [MH00], CORBA [CACM98] [Se98], Corba Component Model [CC], and DCOM [Sz97], are typically distributed models that provide a remote interface also for collocated access. The collocated invocation overhead may be eliminated for CORBA by collocation optimization [SWV99] [PRSGWK99] that is performed a posteriori by the object request broker (ORB), or a priori for EJB 2.0 [EJB01] by providing local interfaces in addition to the remote interfaces. However, EJB 2.0 has not preserved the uniform interface semantics, the location transparency and the simplicity of the distributed component structure of EJB 1.1 (compare [S02]).

Other component models. Emmerich gives an overview on distributed component technologies and their software engineering implications [E02]. KOALA, a component technology for resource-constrained environments like TVs [OLKM00] [O02] is a local technology though distributed component technologies may be embedded in it. Dejay [BWL99] has distributed components with external remote interfaces which are clustered set of objects with internal local interfaces. Collocated external component invocations are done via the remote interfaces. Doorastha [D00] intends to go a step towards distribution transparency and has uniform interfaces.

Component Languages CompJava is based on and improves on local component language concepts from ArchJava [ACN02a] [ACN02b], ComponentJ [SC00] and ACOEL [Sr02].

A version of ArchJava [ASCN03] extends the syntax of connect patterns and expressions, so that a user may realize remote collaborations among components with user-defined connectors. These may throw type-related run-time exceptions or specify additional type-checking rules, regarding e.g. the serializability of reference types. This has the undesirable effects that structural distribution problems are detected only at run-time, and that a component may type-check correctly with one kind of connector, but not with another one.

7. CONCLUSIONS

We have proposed DisComp as a distribution model for the new generation of component languages. It distinguishes different distribution-related characteristics clearly, like remote accessibility and distributed implementation, relates them to basic concepts of a component language, and establishes both semantic equivalence and performance requirements. CompJava is a component language that does allow to distribute components, following the DisComp principles and meeting its requirements.

CompJava allows constructing either completely local components or remotable and/or distributed components. Non-remotable and remotable component types are identical except for the serializability of reference types in the interfaces. Non-distributed and distributed components are identical except for the requirement for remotable types of subcomponents.

Completely local components and remotable components in a non-distributed context, do not impose any distribution overhead, neither with regard to memory footprint nor to invocation cost. Collocated subcomponents of a remotable type in a distributed context do not impose any invocation overhead at all, and practically no memory overhead. The automatic collocation optimization is performed with reference transformations during deployment time.

As a consequence, CompJava makes it very simple to develop clearly structured distributed applications with very competitive performance characteristics, in particular when components collaborate on the same domain, which are scalable from a local environment to differing distributed environments.

8. REFERENCES

[ACM2001] D.Alur, J.Crupi, D.Malks: Core J2EE Patterns; Prentice Hall, Upper Saddle River, 2001

[ACN02a] J. Aldrich, C. Chambers, D. Notkin: ArchJava: Connecting Software Architecture to Implementation. Procs ICSE 2002, May 2002.

[ACN02b] J. Aldrich, C. Chambers, D. Notkin: Architectural Reasoning in ArchJava. Procs ECCOP 2002, Springer LNCS, Berlin 2002.

[ASCN03] J. Aldrich, V.Sazawal, C. Chambers, D. Notkin: Language Support for Connector Abstractions. Procs ECCOP 2003, Springer LNCS, Berlin 2003.

[BWL99] M.Boger, F.Wienberg, W.Lamersdorf: Dejay: Unifying Concurrency and Distribution to Achieve a Distributed Java. in: TOOLS99, 1999

[CACM98] Special section on CORBA; Communications of the ACM, Vol.41, No10, October 1998

[CC] CORBA Components; at cgi.omg.org

[D00] M. Dahm: Doorastha, a step towards distribution tranparency; Proc. Net.ObjectDays2000, Illmenau, 2000

[EJB01] Sun Microsystems, Enterprise JavaBeans Specification Version 2.0, www.java.sun.com, 2001

[E02] W.Emmerich: Distributed Component Technologies and their Software Engineering Implications, Procs. ICSE 2002, 2002

[GHJV95] E.Gamma, R.Helm, R.Johnson, J.Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software; Addison-Wesley, 1995

[MH00] R.Monson-Haefel: Enterprise JavaBeans, O'Reilly, Sebastopol, 2001

[OLKM00] R.van Ommering, F.van der Linden, J.Kramer, J.Magee: The KOALA Component Model for Consumer Electronics Software, IEEE Computer, March 2000

[O02] R.van Ommering: Building Product Populations with Software Components, Procs. ICSE 2002, 2002

[PRSGWK99] .Pyarali, C.O'Ryan, D.Schmidt, A.Gokhale, N.Wang, V.Kachro. Applying Optimization Principle Patterns to Design Real-Time ORBs, Procs. COOTS 99 Conf, 1999

[Ro96] Rohnert, The Proxy Pattern Revisited, PLOP2; John Wiley & Sons, Chicester, UK, 1997

[S02] H.A.Schmid: On the Use of Enterprise Java Beans 2.0 Local Interfaces, SEM 2002, Springer LNCS 2596, pp.144-156, Berlin 2002

[SC00] J.C.Seco, L.Caires: A Basic Model of Typed Components. Proc. ECOOP 2000, Springer LNCS, Springer, Berlin, 2000

[Se98] K.Seetharaman: The CORBA Connection; in: [CACM98]

[Sr02] V.C.Sreedhar: Mixin' Up Components. Procs ICSE 2002, May 2002

[SWV99] D.Schmidt, N.Wang, S.Vinoski: Collocation Optimizations for CORBA, SIGS C++ Report, Sept.99

[Sz97] C. Szyperski: Component Software, Beyond Object-Oriented Programming. Addison-Wesley, 1997