Martin Steckermeier, Franz J. Hauck

The VPC programming model for omnipresent applications

Technical Report TR-I4-01-05
2001-05-30

Friedrich-Alexander-University Erlangen-Nürnberg, Germany

Informatik 4 (Distributed Systems and Operating Systems)
Prof. Dr. Fridolin Hofmann

The VPC Programming Model for Omnipresent Applications

Martin Steckermeier, Franz J. Hauck

{mstecker, hauck}@informatik.uni-erlangen.de

Informatik 4, University of Erlangen-Nürnberg
Martensstr. 1, 91058 Erlangen, Germany
Tel.: +49.9131.85.27906, Fax: +49.9131.85.28732

Abstract: *As more and more computers are connected to the global Internet, users increasingly expect omnipresent access to their data and to their favorite applications. But existing programming paradigms as well as operating and middleware systems are not prepared for omnipresent applications. We propose a system called Virtual Private Computing (VPC) which addresses the requirements of omnipresent, globally distributed applications. This paper will primarily concentrate on the component-based programming model of the VPC. It uses an architecture description language (ADL) that clearly distinguishes between the concepts of components types, component implementations and applications. The ADL allows the VPC runtime mechanisms to efficiently and automatically partition, distribute and bind VPC applications on demand.*

1 Introduction

The overall number of personal computers and computing devices used all over the world has tremendously risen in the last few years. Parallel to their overall number, the ratio of computers connected to the worldwide Internet has also continuously increased. People have got used to surfing through the World Wide Web, to sending e-mails, participating in news groups, etc. This increase in connectivity is currently causing a paradigm shift, away from the initial intention of having a personal computer as an own stationary device, which stores all the user's data and programs. Instead of being restricted to use only their own machines for accessing their data and favorite programs, users want to have omnipresent access to their programs and data, independently of where they are and what device they have currently available. Especially new mobile devices like mobile computers, personal digital assistants and mobile phones push this desire.

There are several approaches that try to address omnipresence. Especially thin-client systems, like the *Sunray* system from Sun Microsystems [Sun99a], the *VNC* (Virtual Network Computer) system [RiSt+98] or *Application Service Providing* systems (ASPs) are used in this field. But these models are all server-centric with several well known drawbacks, like limited scalability and low availability in the presence of server failure or network partitioning. To overcome the deficiencies of server-centric approaches, systems like Microsoft's *Farsite* [BoDo+00] try to use a peer-to-peer network of untrusted hosts to achieve high efficiency, scalability, and availability combined with global accessibility.

Although Farsite solves the problem of omnipresent access to data, it still lacks mechanisms to realize omnipresent applications. For omnipresent applications, a user should be able to run his favorite applications from any device connected to the Internet without having to deal with questions about the amount of available local resources, the necessity to run an application locally, remotely or even in a distributed fashion, the location of data needed for the computation, the availability and location of application code and the way it has to be installed, etc.

To accomplish these tasks, we have developed a system called *Virtual Private Computing* (VPC). It uses a peer-to-peer approach to create a worldwide runtime environment for applications and creates the illusion of a single virtual computer that allows a user to access his data and applications from every participating machine. Regarding data integrity and safety, as well as runtime behavior, the VPC system is comparable to a local personal computer. To guarantee high efficiency and performance and to realize the flexibility to adapt to the highly diverse and heterogeneous environment of the Internet, the VPC system automatically partitions and distributes applications.

In the following section we will show several requirements for omnipresent applications and will then introduce our VPC architecture and its programming model that bases on an own architecture description language. Some existing architecture description languages for distributed applications are presented in Section 2.3 to point out deficiencies, which led to the development of our own language. Section 3 will present the concepts of our description language in detail. The paper finally concludes with a summary of improvements gained by the proposed architecture.

2 Omnipresent Applications

While it might be possible to realize an omnipresent application on common systems with standard programming techniques and with support from operating systems and the middleware, it is not feasible, as all of the tasks mentioned above have to be done manually, requiring a highly experienced user with permissions to manipulate all machines that are concerned with a certain computation. Moreover, the requirement that omnipresent applications should be executable in a highly diverse environment cannot be satisfactorily solved by monolithic applications developed in a standard programming model and running on a traditional operating or middleware system. Those applications and runtime environments are not flexible enough to adapt to different resource availabilities, and they can hardly deal with different hard- and software platforms.

2.1 Requirements for Omnipresent Applications

Both, the applications and their runtime environments have to fulfill several requirements to successfully implement omnipresent applications. Some of these requirements, including efficiency, scalability and support for heterogeneity will be discussed in this paper. However, there are a lot of further requirements like security and accounting of resource usage, which are not in the focus of this paper and are subject of further research.

Depending on the computer or computing device that is used to start an omnipresent application, the amount of local resources available for the application can vary by orders of magnitude. The only possibility to cope with this diversity is to automatically partition applications into a number of independent, communicating components that are automatically distributed onto multiple participating machines. Depending on the resource availability, especially on the user's local machine, the runtime environment must be able to decide whether the application needs not be partitioned at all, whether it has to be partitioned in a classical client-server style, or into a huge number of parts with nearly every component distributed onto another host.

If an application shall be distributed over a set of computers, it cannot be guaranteed that all application parts will run on the same kind of platform. On the contrary, restricting applications to use a common kind of platform for all application parts is not feasible, as the example of a personal digital assistant as the main user device clearly shows. In that case, all other application parts would have to run on such devices, too. Omnipresent applications must therefore be able to deal with different hard- and software platforms for their components and have to provide a common communication mechanism between these components.

Depending on policies and capabilities of a participating host, e.g., regarding security and integrity, only application code developed in a certain programming language or compiled by a certain compiler might be allowed to run. The system must therefore allow to use different kinds of implementations. This can additionally help to improve the efficiency of an application by realizing highly compute-intensive application parts in an optimizing compiled language like C++, whereas other parts, e.g., for the graphical user interface, might be developed in a more portable language like Java.

Gaining a suitable overall application performance is essential for the success of omnipresent applications. While a specialized implementation can significantly improve the efficiency of a single component, the efficiency of the entire application, which might consist of a huge number of cooperating components, is influenced by additional effects. Performance degradations in distributed applications are typically due to the communication overhead between distributed parts of the application. The programming model for omnipresent applications therefore has to provide distribution entities that minimize communication to remote entities.

Beside this communication, another source of overhead may not be neglected. In order to run an application part, appropriate code has to be available at its host. In an environment like the Internet, where participating machines are not centrally administered, it cannot be assumed that this code was installed in advance or may be installed manually by its users. Instead it will typically be downloaded on demand. As the language Java shows, this dynamic download can result in severe delays. To reduce the frequency of downloads and thereby the average delay, the programming model has to take care that application code can be reused as efficiently as possible. This is also the basis for an adequate numerical and geographical scalability of the overall system, as the load for code servers and the necessary networking bandwidth is reduced.

2.2 Virtual Private Computing

To satisfy the requirements stated above, we developed a system called Virtual Private Computing (VPC), consisting of a runtime environment together with an appropriate programming model. The term VPC originates from two central aspects of this system:

It is a virtual computer: A huge number of probably globally distributed and heterogeneous hosts can participate in the system and can contribute resources like processor time, storage capacity and networking bandwidth. All these hosts and their resources act like one large, globally available computer offering a huge amount of resources.

It behaves like a private computer: Similar to personal computers, this virtual host allows a user to access his private data and favorite applications independently of the actual client host he uses. Data safety and security is comparable to existing paradigms. The same is true for the execution behavior of applications in response times and execution speed.

In the next two sub-sections, we will sketch the runtime architecture and the programming model that bases on an own architecture description language (ADL). This language will be presented in detail in Section 3.

2.2.1 The VPC Programming Model

In typical component-oriented systems, application developers choose component implementations out of a pool of existing implementations. Generic implementations together with special, application-specific implementations are combined via some kind of glue code and bound to a single binary application. Although there are systems to automatically partition binary software, like *COIGN* [Hunt98] which is able to analyse and decompose DCOM applications for a distribution over two hosts, the majority of applications created via traditional programming paradigms is not distributable.

In the VPC system, an application developer describes the structure of an application, consisting of component instances, their interconnections and their configuration. However, instead of determining an implementation for a component, a VPC developer specifies all the information that is necessary to choose an appropriate implementation at runtime. Only this application description is deployed in contrast to a whole binary application in case of traditional systems. The VPC runtime environment interprets this description and has therefore all the information that is necessary to decide whether and how the application should be partitioned depending on the currently available resources. Partitioning, distribution and binding is done on demand. If a component is accessed by the application the first time, the runtime mechanisms search for an implementation that complies to that information, instantiate the component on a suitable host and connect it to existing components via an underlying CORBA-compliant middleware. The interpretation process is only responsible for instantiating the application's structure, while the actual functionality is realized by the created components, which run without further interference by the interpretation process.

2.2.2 The VPC Runtime Mechanisms

Figure 2.1 gives an overview of the components that realize the most essential mechanisms of the VPC runtime environment and which have to be installed on every participating machine.

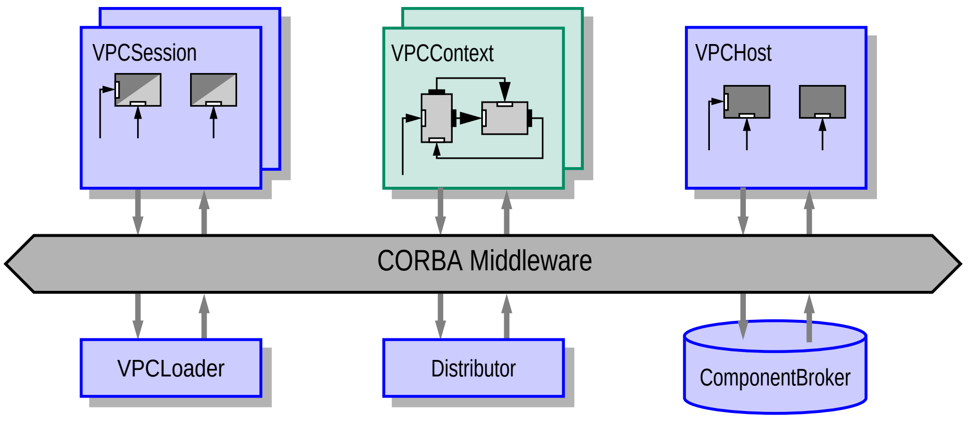

Fig.2.1: The VPC runtime mechanisms on each host

Central component in the VPC runtime architecture is a CORBA-compliant middleware system. All communication between application components as well as runtime components is done via CORBA's remote method invocation. This guarantees interoperability between different hard- and software platforms and supports different programming languages. The middleware is surrounded by several runtime services local to each host:

VPCHost: This is the main component that builds a VPC-enabled host. Its main task is to initialize the whole system by creating all the other necessary components.

VPCLoader: This service is responsible for loading and starting VPC applications, i.e., it receives an application description and starts the necessary components that build the runtime environment for the application components, like the `VPCSession` component and one or more `VPCContext` components for the initial application components.

VPCSession: This service is the representative of a certain application on the host. It maintains all the information about components that belong to the application and run in `VPCContexts` on the same host. Moreover it knows about session components that belong to this application and run on other hosts.

VPCContext: This service provides the actual runtime environment for the components of a single VPC application. Similar to containers in systems like the CORBA Component Model (*CCM*) [OMG99] or Enterprise JavaBeans (*EJB*) [Sun98b], they can host an arbitrary number of components of a particular application, as long as all these components are of the same kind. Different context components can be used for a single application, if different kinds of component implementations have to be run, e.g., C++ together with Java implementations. A host can control which kind of implementations it is willing to run by providing only those types of context implementations. Different context components run in distinct processes of the underlying operating system and are thoroughly separated. A `VPCContext` therefore offers the same integrity and security guarantees as the underlying operating system for traditional monolithic applications.

Distributor: If a component should be accessed the first time, this service is triggered to find an appropriate place to instantiate the component. Based on the information about the application structure and about distributors on other participating hosts, it can find a distribution that assures high efficiency.

ComponentBroker: If a host is chosen by the distributor to create a component, it will be the task of the `ComponentBroker` of that host to find a suitable implementation. This code will either be found in a local code cache maintained by the `ComponentBroker` or it will be dynamically downloaded from remote `ComponentBrokers`. Downloaded code is stored in the local cache and reused as efficiently as possible to reduce the frequency of downloads. Together with the ability of `ComponentBrokers` to continuously learn about other brokers, this mechanisms create a highly scalable and available distribution service for implementations.

2.3 Related Work

Currently most applications that are intended to run on arbitrary hosts connected to the Internet, are developed in Java. The properties of Java seem to solve all requirements of omnipresent applications: Its virtual machine allows to run Java applications on nearly every target platform, its security concept assures the integrity of the hosting machine and it has the ability to dynamically download application code. However, object-oriented languages like Java are not prepared for building distributed applications. Although Java offers a mechanism to transparently invoke methods on remote objects (Java RMI), there are no means to specify the structure of distributed applications. Instead, all structural aspects have to be formulated via functional code. This results in applications that cannot provide the flexibility that is needed for omnipresent applications. Moreover, Java does not reuse implementation code at runtime, but downloads all code again, even if an application is started several times. This results in high latencies and limited scalability.

Even component-oriented systems like CCM or DCOM do not offer means to describe the structure of applications. Again, this information is either provided by functional code or it is manually configured at installation time. This is not appropriate for distributing applications especially if they should be distribute on hosts that were not known at development or installation time.

To overcome these deficiencies, architecture description languages (ADLs) have been introduced. They allow to describe the structure of an application, i.e., the components that build an application, their configurations and their interconnections. Typical representatives of ADLs are *Rapide* [Luck96], *Darwin* [MaDu+95], and *Write* [Allan97]. While most of these systems still create a single binary programm, there are others like *Regis* [MaDuKr94] and *Olan* [BaBe+98] that interpret the architecture description at runtime in order to distribute applications. Although these systems seem to be suitable for omnipresent applications, they still have some drawbacks that severely influence their usability. One of the drawbacks is that they still rely on predetermined implementations. The runtime environment has no possibility to use locally available code other than the requested one, although this code might provide the correct functionality. Instead it has to download exactly the implementations stated by the application description. Moreover the systems immediately instantiate the whole application independently of the actual use of components and therefore create additional overhead. Last but not least, they have limited support for heterogeneity, both with respect to hard- and software platforms as well as to programming languages.

3 The VPC Architecture Description Language

Due to the deficiencies of existing systems using ADLs for distributed applications, we developed an own ADL for our VPC system. All specifications in this ADL are formulated as XML documents. Similar to the ADLs mentioned above, the VPC ADL bases on the concept of describing applications in means of components, configurations and interconnections of components. Common component-based systems use the term component typically either for a component implementation or for a component instance. In the VPC ADL, we explicitly distinguish between three different concepts: the concepts of component types, implementations and instances.

3.1 Component Types in the VPC ADL

As Figure 3.2 shows, a VPC component type consists of two main parts. On the left side of the figure there are the services that a component of type `X` provides to other components. On the opposite side, the figure shows the requirements of type `X`. These have to be satisfied by other components in order to guarantee correct component semantics. Component types defined in the VPC ADL are finally mapped to IDL interfaces in the underlying CORBA middleware and have to be realized by CORBA objects.

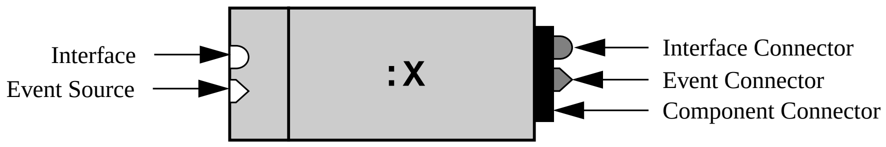

Fig.3.2: Main parts of a component type `X`

In the VPC ADL a component type is usually associated with unique semantics. Although there is research about formally specifying component semantics, we do not apply a formalism as there is hardly any chance to prove formally specified semantics against the actual semantics of an arbitrary, concrete implementation. Instead, our system relies on the correct implementation of informally specified semantics. Nevertheless the system is open for additional specifications that allow formal descriptions, like in [BueWe99]. The VPC programming model assumes that there will be standardisation committees that define component types and their semantics. We further assume that these types will then be implemented by arbitrary software vendors that offer their implementations on a global software market.

3.1.1 Description of Services

Service provision in the VPC system bases on the terms of interfaces and events. Every VPC component can provide an arbitrary number of named and typed interfaces. Every interface allows method-based access to the internal services of a component independently of whether these services will be implemented by a single object or a set of objects.

Interface types have to be declared before they can be used in the specification of a component interface. For interface types, the VPC ADL resembles the most essential features of CORBA IDL, e.g., the declaration of attributes and operations as well as inheritance specifications. As an example, the following listing shows parts of the description of a VPC interface type `Image` with a method to access pixels in an image.

```
<Interface name="Image">
     <Operation oneway="no" name="getPixel" kind="Structure" type="Types::Color" >
          <Parameter direction="in" name="xPos" kind="Long" />
          <Parameter direction="in" name="yPos" kind="Long" />
     </Operation>
</Interface>
```

Based on interface types, a component type can describe, which interface-based services it provides. The following listing shows the description of a component type `ImageSource` that provides an image and allows the access to that image via the interface `Output` of type `Image`.

```
<ComponentType name="ImageSource">
     <Provides name="Output" type="Image" />
</ComponentType>
```

Beside the possibility to provide interface-based services, the VPC system offers an event-based communication mechanism with event sources, where an arbitrary number of event sinks can register for notifications. Again, these services are specified by a name and now by the type of events. Event types are similar to structures in CORBA IDL, except for their additional ability to inherit from a base event type. The following listing shows the definition of an event type

`ImageChangedEvent` with a single member variable `serialNo`. The previous component type `ImageSource` is extended by an event source that notifies all interested entities about a change in the provided image:

```
<Event name="ImageChangedEvent">
     <Member name="serialNo" kind="Long" />
</Event>
<ComponentType name="ImageSource">
     <Provides name="Output" type="Image" />
     <Emits    name="ImageChanged" type="ImageChangedEvent"/>
</Interface>
```

Similar to component models like CCM, service interfaces as well as event sources are mapped to so-called navigation methods in the resulting CORBA representation of the component type. These methods allow to query for a reference to a CORBA object that realizes either the service or the event source. For the user it is transparent whether this is a single object or a set of objects.

3.1.2 Description of Service Requirements

Unlike objects, which also use interfaces to describe their services, components additionally describe requirements, which have to be fulfilled by neighboring components. In the VPC ADL these dependencies are modelled by so-called mandatory component connectors. Beside mandatory connectors, the ADL allows the definition of optional connectors which not necessarily have to be connected to other components. A typical example for an optional connector would be a connector to a logging component.

The description of a component connector consists of a name and the minimum component type the target component has to offer. Using a component type to specify the target type determines both, the services that the target component has to provide as well as its functionality. The following listing shows the definition of a component type `ImageConsumer`, which has one mandatory component connector to a target component of type `ImageSource`:

```
<ComponentType name="ImageConsumer">
     <ComponentConnector name="Source" kind="ImageSource" />
</ComponentType>
```

At runtime, if the target component of a connector is accessed the first time, a connection request will be triggered. This request initiates the dynamic distribution and instantiation of the target component. After the instantiation, the component connector cares for the connection to the services and event sources. Therefore every component connector maintains a set of sub-connectors equivalent to the services and event sources provided by the target component (see Fig. 3.2). The component connector uses the navigation methods at the target component to query for the servicing objects, assigns them to its interface connectors, and registers all its event connectors at their corresponding event sources.

3.1.3 Inheritance and Conformance of Component Types

Traditional component-based systems, which directly use implementations as their building blocks, do not need any conformance relationship. In our VPC system the notion of conformance between implementations is essential as it allows additional flexibility in choosing implementations at runtime and thereby helps to improve the reusability of implementations.

Similar to object-oriented systems, the conformance relationship bases on the principle of inheritance, i.e., a component type can inherit from a base component type. In this situation, the new type inherits all services and all connectors from its base type and can extend the type by additional interfaces, event sources and connectors. With respect to services, inheritance leads to a common conformance relationship between base and sub-type. As VPC component types have associated semantics, inheritance also affects the semantics of the sub-type. Not only does the sub-type provide at least the same services as its base type, but additionally these services have to have conforming functionality. Both properties assure that with regard to services, an implementation of a sub-type can be used instead of an implementation of a base type.

For the requirements specification of a component, this implicit conformance relationship is typically not true. An implementation is only usable, if all of the requirements specified by its type can be satisfied by other components. This is done by connecting all mandatory component connectors to appropriate target components in the architecture description. If a sub-type is now extended by another mandatory component connector, an implementation of this type needs a further connection which cannot be provided by the application description that was designed for the base type, i.e., this extension breaks the conformance relationship between base and sub-type. The relationship will only be preserved if the sub-type does either not extend the requirement specification at all or if it adds optional component connectors only.

As long as a sub-type is conforming to its base type, an implementation of the sub-type can always be used instead of an implementation of the base type. This enhances the chance to reuse component implementations.

3.2 Description of VPC Applications

The VPC ADL allows to describe complex components and even applications by specifying their structure, which consists of the components that make up the application, their configurations and their interconnections. For the VPC system, an application is nothing more than an ordinary component implementation, which is represented by a so-called compound component and realizes a previously defined component type. This approach allows to treat applications equal to other binary implementations. Thus, applications themselves can be part of the composition process of larger components and compound component description can be reused similar to other binary implementations.

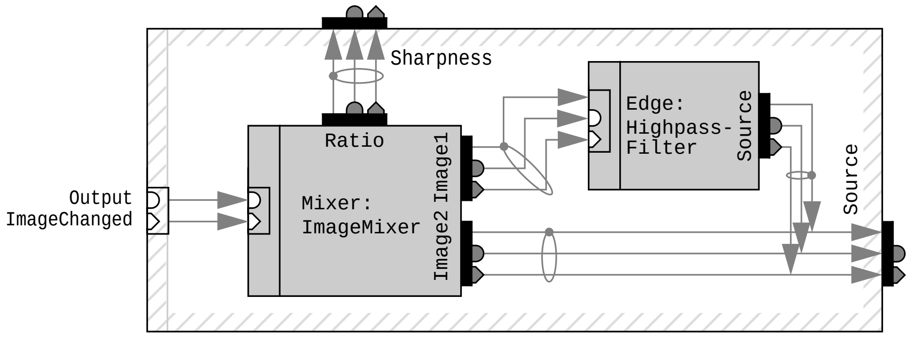

Fig.3.3: Example of a compound component

Figure 3.3 shows an example of a simple compound component that uses two sub-components to realize its own behavior. The corresponding description, which is shown below, exactly resembles the structure that is shown in the figure. The description shows how sub-components are defined via `Use` and eventually configured via the `AttValue` statement. Sub-components are then connected via the `Bind` statement, which usually creates bindings on the basis of component types, i.e., all sub-connectors of a component connector are connected to their corresponding counterparts. Alternatively service requirements of sub-components can be satisfied by binding component connectors to service requirements of the compound component. Finally, all services that the compound component has to provide are mapped to individual services and event sources provided by one of the sub-components via the `Export` statement.

```
<CompoundComponent type="ImageSharpener">
     <Uses name="Edge"         type="HighpassFilter" />
     <Uses name="Mixer"        type="ImageMixer" >
          <AttValue name="DefaultRatio" value="1.0" />
     </Uses>
     <Bind connector="Mixer.Image1"  target="Edge" />
     <Bind connector="Mixer.Image2"  target="Source" />
     <Bind connector="Mixer.Ratio"   target="Sharpness" />
     <Bind connector="Edge.Source"   target="Source" />
     <Export name="Output"           service="Mixer.Output" />
     <Export name="ImageChanged"     service="Mixer.ImageChanged" />
</CompoundComponent>
```

Descriptions of compound components do not make any statements about the structure of sub-components. This leaves the runtime environment the freedom to either choose monolithic binary implementations to realize the sub-components or to use compound components again, which then leads to a hierarchical application structure.

3.3 Specification of Properties and Requirements

So far, the description of component implementations bases on component types, i.e., on the description of services, requirements and semantics. But this description is still not precise enough to choose implementations. There could be several implementations that satisfy a specification but differ in other essential properties. As an example consider implementations that all realize a spelling-checker component. Their common component type `SpellingChecker`, which would have been standardized by some committee, exactly specifies all services, requirements and semantics. But nevertheless, possible implementations could differ, e.g., in the languages they are capable to check.

This kind of information is implementation dependent and should therefore not be included into the type description as this would lead to a huge number of incompatible type descriptions. Instead, the VPC ADL supports the specification of properties for implementations. A single property descriptions is formulated via a `Property` statement, which specifies a name and a set of values (numbers or strings) for the property and an operation that describes how to interpret these values (`AtLeast`, `AtMost`, or `Equals`). The following example shows the specification that the implementation is able to check English and French texts:

```
<Property  name="Language" value="{'English', 'French'}" op="Equals" />
```

On the other side, implementations can specify which properties other components have to provide in order to realize their own properties. These specifications are possible for components connected to one of the component connectors of the specifying component or for the sub-components of a compound component. The following example shows a requirement specification calling for an implementation that at least supports the checking of English texts:

```
<Requirement name="Lang" value="{'English'}" op="AtLeast" />
```

As the properties of a component can depend on the properties of its neighbor- or sub-components, the VPC ADL supports the derivation of requirements. If there is a requirement specification upon an implementation, the implementation can define how to map these requirements onto requirements upon other components. This mechanism to express dependencies between the properties of implementations enhances the reusability of compound components. Implementations for sub-components with enhanced properties can be introduced without the need to change compound components in order to exploit these new properties.

The possibilities to express properties and requirements as well as the derivation mechanism are carefully designed to cope even with complex architectures with mutual interdependencies and cyclic dependency paths. Nevertheless the expressiveness of the specification language is powerful enough to formulate qualitative and quantitative requirements, similar to common trading services. Moreover it replaces external deployment descriptors that are used in other systems to describe properties of implementations and to specify their runtime requirements. More details about the potential of the description language can be found in [Ste01].

4 Summary

The paper presented the vision of omnipresent applications. These applications should be accessible by any machine connected to the global Internet and allow the user to work on his data independently of which device he has at his disposal. We proposed a system called Virtual Private Computing (VPC), consisting of several runtime mechanisms and a programming model that provides the basis for effective and efficient omnipresent applications. The proposed programming model together with the runtime mechanisms has the following advantages compared to traditional systems:

- The architecture description of an application in the VPC ADL contains all the information about component instances and their interconnections. VPC runtime mechanisms can use this information to dynamically decide about the degree and form of partitioning and can distribute these parts depending on the available platforms and resources.
- Distribution, download of code, instantiation and binding are transparently done by the VPC runtime services, without any assistance by the developer or user of an application.
- Component instantiation and distribution is delayed until the application accesses a component the first time. This avoids unnecessary overhead caused by the instantiation of components that will never be used.
- As the components of the application are not specified in terms of implementations, file- or classnames, the runtime mechanisms can freely choose between all implementations available for the local platform that satisfy the description. Moreover the mechanisms can adhere to local policies, e.g., they can choose binary implementations in case of high efficiency requirements or Java implementations in case of higher security considerations.
- The decoupling of component description and implementations builds the basis for a flexible reuse of implementations that already exist in a local code cache maintained by the VPC system. This reuse can either be due to a repetitive execution of the same application or by familiarities between different applications, which leads to a demand for related implementations. The improved reusability of code significantly reduces the number of downloads and therefore improves the efficiency and scalability of the whole system.
- The programming model is uniform. Applications are realized by composing other components and are themselves seen as component implementations. Thus applications are

subject to the same treatment as small binary implementations: they can be used as building blocks for larger applications and they are reused as far as possible.

This paper concentrates on the basic language aspects of the VPC system. Several other, essential topics, like efficient distribution strategies, security issues, resource accounting, and software licensing have been neglected by this paper, but have to be addressed in order to successfully implement the proposed model.

5 References

Allan97	Allen: A *Formal Approach to Software Architecture*. PhD thesis, School of Computer Science, Carnegie Mellon University, 1997
BaBe+98	Balter, Bellissard, Boyer, Riveill, Vion-Dury: *Architecturing and Configuring Distributed Application with Olan.* In: Proceedings of the IFIP International Conference on Distributed Platforms and Open Distributed Processing, 1998
BoDo+00	Bolosky, Douceur, Ely, Theimer: *Feasibility of a Serverless Distributed File System Deployed on an Existing Set of Desktop PCs*. Proceedings of the International Conference on Measurement and Modeling of Computer Systems, 2000
BueWe99	Büchi, Weck: *The Greybox Approach: When Blackbox Specifications Hide Too Much.* Turku Center for Computer Science, 1999
Hunt98	Hunt: *Automatic Distributed Partitioning of Component-Based Applications.* Department of Computer Science, University of Rochester, Rochester, New York, 1998
Luck96	Luckham: *Rapide: A Language and Toolset for Simulation of Distributed Systems by Partial Orderings of Events.* Lecture Notes in Computer Science, 1998
MaDu+95	Magee, Dulay, Eisenbach, Kramer: *Specifying Distributed Software Architectures*. Proc. of ESEC '95, IEEE, 1995
MaDuKr94	Magee, Dulay, Kramer: *Regis: A Constructive Development Environment for Distributed Programs.* In: Proceedings of the International Workshop on Configurable Distributed Systems, 1994
OMG99	Object Management Group: *CORBA Components - Volume 1.* Aug. 1999
RiSt+98	Richardson, Stafford-Fraser, Wood and Hopper: *Virtual Network Computing.* IEEE Internet Computing, 1998
Ste01	Steckermeier: Virtuelle Private Rechner - Eine Software-Architektur für verteilte Systeme, Arbeitsbericht des Instituts für Informatik, to be published, 2001
Sun98b	Sun Microsystems: *Enterprise JavaBeans.* 1998
Sun99a	Sun Microsystems: *Sun Ray1 Enterprise Appliance, Overview and Technical Brief.* 1999