Virtual Private Computing A New Paradigm for Component-Based Distributed Applications

Martin Steckermeier,
Franz. J. Hauck

July 2000 TR-I4-00-02

Technical Report

Computer
Science Department

Operating Systems — IMMD IV

Friedrich-Alexander-University
Erlangen-Nürnberg, Germany

Virtual Private Computing - A New Paradigm for Component-Based Distributed Applications

Martin Steckermeier and Franz J. Hauck
Department of Computer Science IV, University of Erlangen-Nürnberg, Martensstraße 1, 91058 Erlangen, Germany
Martin.Steckermeier@informatik.uni-erlangen.de,
Franz. Hauck@informatik.uni-erlangen.de

Abstract:

Although the number of devices connected to the Internet or some kind of intranet has grown tremendously over the last few years, there is still no widely accepted and uniform concept for designing applications that are able to meet the diverse requirements of such heterogeneous systems. The architecture that is presented in this paper overcomes these problems by creating a vision of a Virtual Private Computer that combines participating machines to a large, omnipresent computer. The integral part of this architecture is a programming model that supports the development of network aware applications composed of components, which are automatically distributed over the virtual computer based on information about properties of the environment.

1. Introduction

The development of the Internet over the last few years has been influenced heavily by two trends. On the one hand, the number of devices connected to the Internet or some kind of intranet has tremendously risen. Not only is this caused by the increased number of traditional and network computers, but more and more consumer devices like settop boxes and intelligent communication devices are connected to the network and are able to run applications on behalf of their users. On the other hand this development was also influenced by the introduction of new transport media that allowed higher bandwidths at a significantly reduced price, where the bandwidths of typical media that are used by individuals range from a few kilobytes per second (e.g., GSM, analog modems, ISDN) up to several megabytes per second (e.g., ADSL).

Beside the possibility to transmit large quantities of data like in audio or video streams, distributed systems also allow to realize applications that exploit more then only the local computer, or an additional server in case of a client/server application, but can be distributed on a network of computers. This allows to balance resource usage between several hosts and therefore reduces the requirements to the local machine, which is especially interesting for devices that should be cheap and have only limited computing and storage capacity. In these applications, the partitioning and distribution of components has a huge influence on the overall application and system performance. Common middleware systems for distributed programming assist the application programmer in partitioning his program into independent parts, which can then be started as services on different machines. Nevertheless, the partitioning as well as the distribution is statically done at development and deployment time. This static approach was sufficient for traditional client/server applications in an environment, like a local network, where the properties of the participating machines as well as their interconnections were known in advance. This is no longer true for a heterogeneous Internet environment as both, the available bandwidth as well as properties like computing or storage resources of participating machines are not known in advance and can vary dramatically in time. Although traditional middleware systems for distributed programming can help to partition applications into smaller parts, they lack distribution mechanisms, placement strategies based on environmental conditions, and mechanisms for transparently transferring code to the chosen host. To overcome the deficiencies of such systems, we propose an architecture that establishes the notion of a *Virtual Private Computer* (VPC). Participating devices provide resources to this virtual computer. Applications that are started on the VPC are dynamically distributed on available hosts to share all resources like processors, memory, secondary storage, and networking capacity. This reduces the requirements upon the local machine as the system is usually able to additionally exploit idle resources of other machines. Nevertheless, the user has still the imagination of a private machine, which his applications and data available wherever he contacts this possibly globally distributed machine.

After a survey of existing possibilities to utilize a connection to a network, and an identification of their major deficiencies in Section 2, we will introduce our system called *Virtual Private Computing*. In this system, application composition and distribution is based on an own component-oriented programming model, which is described in Section 3.1. Section 3.2 gives an overview over the most essential services that make up the runtime environment for

VPC applications and shows how they interact to distribute and run applications. We will conclude in Section 4, with a brief summary of all essential benefits that our architecture brings for the field of globally distributed applications.

2. Paradigms for Realizing Network Aware-Applications

Applications that should profit from a network connection to the Internet and the possibility to intercommunicate with programs on remote hosts typically stick to a certain programming paradigm for distributed computing. This section will point out common deficiencies of existing paradigms by showing how they can be used to solve an exemplary problem: A large dataset, e.g., gained by a computer tomography should be examined by a human person. Therefore, an application reads the dataset, extracts dedicated parts like a slice through the given volume, does compute-extensive analysis on this data and finally renders the results to an image that is presented to the user. The user is able to influence the extraction and analysis process as well as the visualization interactively. Every paradigm will be examined for the following criteria:

- Is it possible to share the workload between different hosts and which communication costs arise from the distribution? What are the demands upon the local machine?
- Does the user have to be aware of code or data locality, and does he have to care for code and data distribution?
- Is the user urged to know about static or dynamic properties of the computing environment, e.g., hostnames or the current workload of participating machines?
- Is the computation location-independent, i.e., can the user move to another, possibly foreign computer and restart the application there to continue his work?

2.1. Non-distributed Computing

The simplest, however still most common approach is non-distributed computation, i.e., code and data reside on the local machine and the computation is completely performed on this machine without any connection to another host.This approach, which is typical for personal computers, has the advantage that no communication costs arise and the machine is completely autonomous, which is especially beneficial for a mobile device. On the other side, our sample application imposes huge storage and compute load to the client host. Especially for small or mobile devices, this approach is not practical. Moreover, the user is responsible to keep data and code consistent if he uses different hosts at different times.

2.2. Client/Server Computing

To reduce the computing and storage requirements for the client host, responsibilities can be transferred to a server host. A common approach is to store the data sets and probably the code on a central fileserver and export it to clients by mechanisms like NFS [3] or AFS [3]. For the client computer, only the computation requirements remain. But this solution has two major drawbacks. Due to the amount of data that has to be transferred and the access pattern, the costs for local storage are now replaced by possibly high communication costs. Moreover, the client host is no longer autonomous but depends on the file server. Especially in case of a mobile computer, the access to this service can not be guaranteed all over the world without additional mechanisms like VPNs. On the other side, a user is able to start his application on every computer, as long as this host has access to the server.

Another approach that uses the client/server paradigm additionally shifts the computation onto a compute server. In our sample application only the displaying responsibility may remain on the client host. The client is now relieved from storage as well as compute load, which allows very “thin” clients. However this approach requires a mechanism to remotely display the results of the computation, e.g., by using mechanism like X-Windows, VNC [14], RAWT [15] or a commercial product like Winframe. Another drawback of this solution lies in the high communication costs for updating the remote display. Depending on the frequency the user changes parameters in the sample application and the complexity of the transferred display data, this approach needs a high bandwidth and low latency network.

But the most severe drawback of this solution lies in the lack of transparency for the user. Not only does he have the possibility to delegate computation to a compute server, but now he has the responsibility to find such a server that is appropriate for his computation. This decision has to be based on considerations about the possible computing performance of the available machines, their current workload, the location of the data the application accesses, and the communication bandwidth between the host holding the data, the compute server and the client host. In most cases, the user will not be able to find an optimal decision that ensures best overall performance and at the same time minimizes costs. Moreover, the user has to be known by the compute server, i.e., he has to have appropriate permissions to start an application there. For administrative reasons, this will only be applicable in local departments, but in general not in a global range.

2.3. Distributed Applications

The above solutions had in common, that the computation was always concentrated at one host, either the client host or a compute server. To reduce communication costs further, the application could be partitioned and distributed over both hosts. In our example, dealing with the data and the main computation can be done near to the data, whereas the rendering and displaying parts can be located on the client hosts.

This partitioning can either be done by the software developer that knows the communication behaviour and requirements regarding memory and compute performance inside his application, or can be done by automatic partitioning systems like Coign [7] that measures the communication behaviour in several profiling runs and uses this information to find an optimal partitioning. However, the application is still statically partitioned into a client and server part. It is not able to react to changes in the runtime environment, like the available network bandwidth or latencies, and is still limited to a distribution over two hosts.

Another approach, especially for high performance computing is known as *Metacomputing*. Originally intended to bundle the performance of only a few special high-performance computers, metacomputing has been extended to use *Networks of Workstations* (NoWs) or even any willing computer connected to the Internet in form of so-called *Volunteer Computing*. According to the different types of machines, there is a broad spectrum of systems, from simple communication mechanisms like PVM [4] or MPI [9], over architectures like Mentat [1] and Legion [2], to Java based systems like Javelin [5], Bayanihan [8], and PopCorn [13]. Although they are able to partition applications into many similar parts that can be independently distributed, they are not generally applicable because they follow a fork-join approach with application parts that do no or only little communication between each other.

2.4. Requirements for Fully Distributed Applications

As can be seen from the above examples, a middleware system that should be able to support fully distributed applications in an network environment has to fulfil several properties:

- To be able to adapt to different runtime environments, applications should be composable of independently distributable parts like objects or components.
- Distribution decisions should be made by the runtime system with regard to application requirements and the properties of the runtime environment (e.g., workload of possible machines and the available bandwidth between these machines). Neither the developer nor the user should be urged to find a beneficial distribution.
- When a suitable host has been found, the runtime system should automatically download the necessary code for this architecture, without further manual assistance.
- Started application parts should be bound automatically and interoperate via a location-transparent communication mechanism.
- Due to the heterogeneity of the environment, different architectures and operating systems have to be supported.
- To minimize latencies for starting applications, code has to be extensively reused. This at least presumes the existence of some kind of code-caching facility.
- For the user, starting a distributed application has to be as easy and efficient as starting a local, non-distributed application. Moreover, applications should be ubiquitous in the sense that the user can start his application independent of where he is and which computer he has at his disposal.

Our architecture called *Virtual Private Computing* (VPC) addresses this properties and consists of several services that have to be installed on every participating machine. Moreover it has an own programming model to realize the partitioning and distribution.

3. Virtual Private Computing

To overcome the deficiencies of existing systems, our VPC architecture relies on components as distribution entities and a strict decoupling of component types and component implementations to maximize reuse. We assume, that in the near future there will be a small number of standardization boards like the OMG, which will define interfaces of commodity components, like for example spelling checkers, spreadsheets, and word processors. We also assume a system for uniquely naming these interfaces. On the other hand, there will be numerous companies that will realize these components, probably in different languages and for different platforms. Applications can then be built by exploiting this global component market.

Our VPC system uses these commodity components for software composition and provides the following additional properties and services:

- Components get bound to applications at runtime by mapping abstract component descriptions to real implementations on demand via a component broker. By using an abstract description instead of specifying concrete classes, the runtime system is free to choose an implementation that fits its needs best and reuse code that already resides at the local host.

- The selection process can be influenced by additional criteria like the actual platform the code should be executed on, or the available resources. This allows an automatic adaption of the application to its environment, e.g., by searching for component code optimized for the actual hardware platform.
- Component code is downloaded dynamically and transparently for the user, if it is not already contained in a local component repository. Neither the user nor any system administrator have to care for code distribution and only the code that is really used is downloaded.
- If a typical user profile is known, implementations for commodity components can be installed locally in advance by the host administrator. This additionally reduces the latency for most component instantiation requests as unnecessary downloads are avoided. This leads to a better scalability in the number of components that can be used for an application.
- Component instances are distributed automatically via a distribution service to optimize the overall system performance. Users as well as administrators do not have to keep track of the workload of machines, or the bandwidth and latencies of networks. Additionally application developers do not have to care about distribution decisions either.

3.1. The VPC Programming Model

Different middleware systems favour distinct distribution entities. In modern systems this are either objects or components. Our system relies on components as distribution entities for two reasons:

1. In many systems, most objects tend to be small entities. As an example, a survey of over 4500 classes in the Sun JDK 1.2 runtime library showed an average class size of about 2200 Bytes. As downloading files over the Internet involves high latencies these small class files usually have to be clustered to larger entities like libraries for efficiency reasons anyway.

2. Components are by definition software entities that are designed for reusability. In our system, reusing component implementations is a key property that improves the overall efficiency. No statements are made about the size of components in a VPC, i.e,. small self-contained objects like a simple button can be interpreted as components just as well as large modules like a spelling-checker can be seen as a component, but we assume that our system will offer the highest improvements for applications composed of medium sized components.

In most component-oriented systems, components are only used in the development process. Existing component implementations are utilized for the assembly process of new applications. Several specification mechanisms or textual descriptions together with tools assist the developer to find appropriate implementations according to his requirements. Beside simple type checks, some systems support additional checks in form of pre- and postconditions for methods to prove the suitability of an implementation and the correctness of the assembly process. However, the result of these assembly process is mostly a single, monolithic binary application, which is intended to run in a non-distributed fashion.The information about the internal structure, the types of components and the requirements that led to the selection of a specific implementation are typically lost.

In our VPC programming model, applications are not assembled to a single binary, but instead a VPC application is represented by a description of component types, requirement specifications and the necessary links between those components. Our runtime system does late binding of components on demand, i.e., it searches for appropriate component code and instantiates components at the first time they are used. No component code or other application functionality is expressed in the description language, as components should be realized in the language that is best suited for the problem they solve. Therefore no complex programming language is necessary. Instead we encode the descriptions in XML [6], which is already an approved means used for similar purposes in form of XMI (XML Metadata Interchange, [11]).

Component Types: The description of component types used for our VPC applications is similar to the OMG component model [12], but is enhanced by additional information. It bases on the concepts of interfaces, events, components, and connectors, which are expressed in XML and are currently mapped to CORBA IDL, as CORBA is our favorite communication platform. But our model is not restricted to CORBA, alternative mappings would enable us to map our component specifications to other middleware systems like RMI or DCOM.

Interfaces: Similar to CORBA IDL, interface descriptions comprise datatypes, exception types, attributes and signatures of methods. Multiple inheritance of interfaces is supported, with the same restrictions as in CORBA regarding overriding and overloading [10].

Events: Events as a simple message passing mechanism are also supported by the language. Specifications for events comprise three parts: the type of an event, event sources and event sinks. Whereas event sources and sinks are part of the component description, event types can be declared beforehand, similar to data types.

Components: Components are the units of distribution in the VPC system and are therefore the central entity of the specification language. Instead of a dedicated component interface, components support one or more of the previously defined interfaces. A component must be able to hand out references to objects that implement the supported interfaces at runtime. Interfaces specify which services the component offers to other components and how these services can be accessed. Similar to interfaces, a component can contain several event sources. Effectively, an event source specified by a name and an event type is nothing more than a special type of interface, that allows to register one or more event sinks. Events are pushed to these event sinks by method invocations.

Connectors: The counterpart of interfaces and event sources are connectors. They specify points where components can be connected with others and requirements upon those components:

- *Interface Connectors*: This connector type declares, that the specified component will communicate with another component that offers a service with this specific interface, i.e., interface connectors are directed, typed communication paths.
- *Event Connectors*: Event connectors are the means to specify event sinks. Binding an event connector to an event source results in registering the sink object there.
- *Component Connectors*: These connectors express a “uses” or “needs” relationship to another component. They do not state whether or how the related components will communicate, but can be used to initiate connections of event or interface connectors. Moreover they influence the selection and binding process of components at runtime as will be shown later.

Component Implementations and Properties: The type that a component implementation realizes is one of its major properties. Nevertheless it is a well known fact that this single property is not sufficient to select an implementation.To come back to our sample application, suppose for example a component that reads the unformatted volumetric data from a generic file component and offers an interface to retrieve the values at arbitrary positions in this three dimensional array. Although the component type declaration will be able to precisely specify the interface that is offered for retrieval and the connector to the file component, there is generally no means to state which dataformats a concrete implementation is able to read.

Our system allows developers to describe such properties for their implementations. They can annotate their code with additional property declarations consisting of property names and values, which can be numbers as well as character strings. In case of character values, only single values or sets of values are allowed, whereas in case of numeric values, ranges are also possible. This declaration, which again is expressed in XML, could for example be used to state that an implementation of the above mentioned component is able to read unencoded as well as run-length encoded data.

Application Description and Requirements: Based on the description of component types, more complex constructs like applications and compound components can be built, which are both handled similarly in the VPC language.Their description has to fulfil three basic tasks:

1. It has to state which subcomponents should be created for the application or the compound component.
2. It has to define how these subcomponents should be connected.
3. In case of a compound component it has to map the interfaces and connectors of the type it realizes to those of its internal subcomponents.

The most delicate task is the first one, as it resembles the task of a human software developer who has to decide which implementation to use for a component. A feasible candidate has to implement the requested component type and must usually also fulfil additional requirements. Referring to the previous example, an application might request an implementation that is able to read at least the above mentioned formats.

Similar to the property declarations, our language therefore allows to specify which properties it expects from a component. Again, these XML specifications consist of property names together with a single or multiple values, but now an additional comparison operator can be given. An implementation is said to be conforming to a requirement specification if for every single requirement it has a corresponding property declaration that fulfils this requirement. A property fulfils a requirement, if the comparison of property and requirement values as a function of the specified operator evaluates to true. The system currently supports the following operators:

Equals: Property and requirement values have to be equal, sets or ranges have to contain exactly the same values.

AtLeast: The property values have to contain at least the requested elements. Single values are interpreted as a set containing this value.

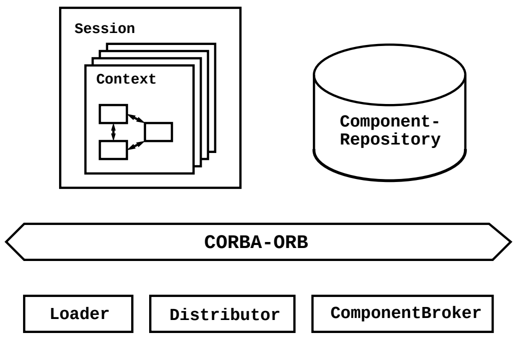
AtMost: The property values may not exhibit values outside the specified set or range. Again, single values are interpreted as a simple set.

Requirements can be globally specified for a subcomponent of an application or compound component. Additional requirements can be specified at a component connector of another component. The latter is a sign that the specifying component expects special properties from the target components to which it wants to connect. All requirements, global as well as those from connectors are collected and are used by our component-broker service to find an appropriate implementation.

But component connectors have further responsibilities in an application: If the running application tries to access the connector's target component and there is still no connection established, the connector can initiate this connection, either by initiating the instantiation of the target component or by searching for an already running component similar as in common trading services. If it succeeds, it connects both components by connecting its subordinate interface and event connectors, i.e., connectors are not mapped to simple object references in the middleware, but to first-class objects that can react on bind and unbind request.

3.2. The VPC Services

Beside the programming model, our architecture consists of several services that have to be installed on every participating machine and which form the runtime environment for VPC applications. The figure below shows an overview over the basic architectural services.



The central service shown in the figure is a CORBA-compliant object request broker. This communication middleware was chosen because of its language and platform independence, but every other middleware like DCOM or RMI would have been equally applicable. All communication between distributed components of the application as well as the VPC services is done via this system.

When a new application should be started at a host, the application description is given to the *Loader* service. The *Loader* creates a so-called *Session* that represents the application on this host. After that, the *Loader* interprets the application description and starts to instantiate the first components, which will be typically part of the user interface. The *Loader* asks the *ComponentBroker* for an implementation of the specified component type that fulfils the given set of requirements and conforms to additionally possible local policies, e.g., regarding the allowed languages. The *ComponentBroker* first searches through its local *ComponentRepository* to find an appropriate implementation. If it fails, it forwards the request to other brokers. Returned component code is stored in the local *ComponentRepository* for later use, before it is returned to the *Loader*. Depending on the type of code, the Loader starts a suitable *Context*, e.g., for a component in Java bytecode, it would start an appropriate Java virtual machine. It is then the task of the *Context* to load the component code and instantiate the component.

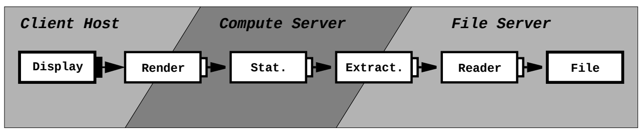
Later instantiation requests are forwarded to the *Distributor*, which communicates with other distributors to find a suitable host for creating a component. This distribution bases on information about the component that should be created, about the placement of already existing components as well as knowledge about the current conditions of the runtime environment, like workload or available network latencies and bandwidth.

Every time a new component should be created at a host, which was chosen by this distribution mechanism, its runtime system checks whether there is already a running *Context* for the requesting application, and whether it fits to the chosen component code. Although our system supports that components that are realized in different programming languages co-operate in one application, for every type only one *Context* is started. All components of the same implementation type share this *Context*.

3.3. A Sample VPC Application

The next Figure shows the simplified structure of our previous sample application in a VPC environment. Computer tomography data is read from a file located on a fileserver by a component. From this huge data volume, parts, e.g., a single slice, have to be extracted and analyzed by another component. The results are then rendered to an image and displayed at the user's machine.

For the real application more then the shown components would be necessary but have been left out for clarity reasons.



There are two dedicated components in the application: the *Display* component and the *File* component. Both are fixed to their host for obvious reasons: the displaying component has to run on the user's host whereas the file component should to be placed at the host where the real data resides. All other components can be distributed freely by the VPC. In the given example, a *Reader* components reads the contents of the file and therefore reside close to the file component. Depending on the workload of the fileserver, the component that does the extraction process can either be collocated with the *Reader* or can be placed on a compute server, i.e., a participating machine with sufficient free compute performance. This machine will typically also do the required computations on the extracted data and will forward the results to the *Render* component. The placement of this component now depends on the available resources on the client machine and the available network resources. In case of a low-performance machine, the rendering component will also be placed at the computeserver. Otherwise it can be placed on the client host. Independent of the chosen distribution, the system will care for the transmission of component code to the chosen host and the connection of all components.

Beside the overall structure shown in Figure, the application description will include requirement specifications to control the selection process of implementations. For example, for the *Reader* component a possible requirement would be that it is able to read so-called RLE slice files. Similar, the extraction component can be specified to allow arbitrary cuts through the given volume, not only orthogonal cuts.

4. Conclusion

This paper introduced a new software architecture we developed for fully distributed applications and concentrated on the language, which is used to describe components and applications. Moreover it showed the most essential services that are necessary to interpret this description and to distribute and connect components.

Components are not statically, but dynamically bound at runtime via a component trading mechanisms that searches in a potentially global software market for implementations that fulfil an abstract specification. By using this abstract specification instead of concrete classes, we raise the probability that a locally cached implementation can be reused. This avoids unneccessary downloads and improves overall system performance. Code that is not contained in a local code repository is downloaded dynamically to any host affected by the application without any further user assistence.

Based on knowledge about the distribution of other application components and the properties of the runtime environment, the system automatically distributes newly created instances onto suitable hosts and connects them to the application. Again, to avoid unneccessary downloads, instantiation and creation are done on demand, i.e., only the code that is actually needed gets downloaded to a host.

Exploiting these mechanisms, developers can compose applications consisting of a huge number of components without regarding for distribution decisions. The system cares for efficient distribution of component code and instances. On the other hand, the user is able to start his favorite application everywhere, as long as he is able to access the application description, e.g., via WWW.

Beside the issues addressed in this paper, further research has to be done on topics like security aspects, controlling the distribution, administrative problems like accounting for resource usage, licensing of components in the assumed component market, etc. Our short term target is the completion of our Java prototype to provide a testbed for new strategies and mechanisms. Sample application should then be used to gain experience with the new programming paradigm.

References

[1] Andrew S. Grimshaw. The Mentat Computation Model, Data-Driven Support for Object-Oriented Parallel Processing. Department of Computer Science, University of Virginia, May 1993

[2] Andrew S. Grimshaw, Wm. A. Wulf. Legion - A View From 50,000 Feet. Department of Computer Science, University of Virginia, August 1996

[3] Andrew S. Tanenbaum, Modern Operating Systems. Prentice-Hall, 1992

[4] Al Geist, et.al. PVM: Parallel Virtual Machine - A Users' Guide and Tutorial for Networked Parallel Computing. MIT Press, 1994

[5] Bernd O. Christiansen, Peter Cappello, et.al. Javelin: Internet-Based Parallel Computing Using Java. Department of Computer Science, University of California, Santa Barbara, June 1997

[6] Extensible Markup Language (XML) 1.0. W3C Recommendation, Febuary 1998

[7] Galen C. Hunt, Automatic Distributed Partitioning of Component Applications. Ph.D. Dissertation, Department of Computer Science. University of Rochester, July 1998

[8] Luis F. G. Sarmenta. Bayanihan: Web-Based Volunteer Computing Using Java. MIT Laboratory for Computer Science, Cambridge, April 1998

[9] MPI Forum: MPI-2: Extensions to the Message-Passing Interface. University of Tennessee, 1997, http://www.mpi-forum.org/docs/mpi-20.ps.Z

[10] Object Management Group: The Common Object Request Broker: Architecture and Specification, Rel. 2.2. February 1998. OMG Document formal/98-02-01, February 1998

[11] Object Management Group: XML Metadata Interchange. OMG Document ad/98-10-05, October 1998

[12] Object Management Group: CORBA Components. OMG Document ad/98-10-05, March 1999

[13] Shmulik London. POPCORN - A Paradigm for Global-Computing. Master Thesis, Institute of Computer Science, The Hebrew University of Jerusalem, June 1998

[14] Virtual Network Computing. AT&T Laboratories Cambridge, 1999, http://www.uk.research.att.com/vnc/index.html

[15] Yosef Moatti, et.al. Remote AWT for Java. IBM, November 1999, http://www.alphaworks.ibm.com/aw.nsf/techmain/remoteawtforjava