



**PEPITO**  
**IST-2001-33234**  
**PEer-to-Peer Implementation and TheOry**

**Deliverable no: D4.10**

## **Final report on Distribution Subsystem**

REPORT VERSION: first

REPORT PREPARATION DATE: 2004.12.31

CLASSIFICATION: Public

DELIVERABLE NO: D4.10      DUE DATE: Month 36      DELIVERY DATE: Month 36

PROJECT START DATE: 2002.01.01      PROJECT DURATION: 36 months

RESPONSIBLE PARTNER: SICS

PARTICIPATING PARTNERS: SICS

PROJECT COORDINATOR: Swedish Institute of Computer Science AB

PROJECT PARTNERS: EPFL Lausanne, INRIA Paris, KTH Stockholm, UCL Louvain, University of Cambridge UK



**Project funded by the European Community under the  
'Information Society Technologies' Programme (1998–  
2002)**

Project Number: IST-2001-33234  
Project Acronym: PEPITO  
Title: PEer-to-Peer Implementation and TheOry  
Deliverable No: D4.10  
Final report on Distribution Subsystem  
Due date: project month 36  
Delivery date: 2004-12-30

Responsible Partner: SICS  
Participating Partners: SICS

11th January 2005

By Erik Klintskog and Per Brand

## Contents

<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Motivation . . . . .	3
1.2	Contribution . . . . .	4
1.3	Overview of the DSS development . . . . .	5
1.4	Dissemination . . . . .	5
1.5	Pepito Integration . . . . .	5
1.6	Outline . . . . .	6
<b>2</b>	<b>The Distribution Subsystem</b>	<b>6</b>
2.1	Overview of the DSS . . . . .	6
2.1.1	Abstract Entities – the API . . . . .	6
2.1.2	The Coordination Layer . . . . .	7
2.1.3	The Messaging Layer . . . . .	7
2.2	Attached Papers . . . . .	8
<b>3</b>	<b>Generic Distribution Support Systems</b>	<b>11</b>
3.1	CORBA . . . . .	11
3.2	Web Services . . . . .	12
3.3	.Net . . . . .	12
3.4	Software Distributed Shared Memory: InterWeave . . . . .	13
<b>4</b>	<b>Software</b>	<b>14</b>
<b>5</b>	<b>Conclusion</b>	<b>14</b>

# 1 Introduction

This document, deliverable D4.10, is the final report on the distribution subsystem (DSS) of work-package 4 (WP4) in the PEPITO project. The DSS is a middleware designed to provide distribution support for programming languages, on the level of individual data structures. DSS is both a proof of concept of our model of generic distribution support, as well as a vehicle for conducting research in the development of new types of distribution support for distributed programming languages. Accordingly, results from WP2 have been incorporated into the DSS, resulting in new innovations in the area of distribution support.

## 1.1 Motivation

Development of applications that are deployed over sets of computers interconnected by a network, i.e. a distributed system, is known to be complicated. To start with, the programmer is faced with the intrinsic challenges of software development such as correctly implementing a specification, providing quality of service etc. In addition, properties of the distributed systems add further challenges, such as partial failures, true concurrency, and latency. There exists numerous different types of tools and abstractions to help the programmer of a distributed system, ranging from simple communication abstractions such as sockets to distributed programming languages where the distribution support or middleware is integrated in the programming language. These distributed programming languages/systems are characterized by, to varying degrees, transparent distribution [5, 11, 2, 6].

Programming languages where data structures may be local, freely distributed between machine or shared between machine under the same semantic model, modulo failure and latency, greatly simplifies program development for distributed applications. This is distribution transparency. One consequence is that the program may be developed and functionally tested on a single machine, despite later deployment on many machines. Another consequence is that the changes in the distribution structure of threads may be changed with only minimal or no changes in the program. (Changes may be necessary for dealing with partial failure or for performance optimization to reduce the effect of latency). We denote a programming language implementation that provides distribution support for parts or all of its programming model a distributed programming system.

The DSS specifically targets the development of distributed programming systems. Extending a programming system with distribution support is difficult, among other things this requires implementing a messaging abstraction, automatic memory management in a distributed setting, remote access of data structures, references to remote data structures, replicated data structures, security provisioning and failure detection/masking. The DSS reduces the burden of developing distributed programming systems. The DSS is a language-independent middleware library that can be coupled to centralized programming systems to produce a distributed programming system. The DSS can support almost any high-level programming system, include object-oriented, functional, and data-flow based programming languages. The rationale for DSS is summarized in the goals of WP4:

*“The objective of this work package [WP4] is to provide a language-independent distribution subsystem. This distribution subsystem will offer a wide range of distributed services/protocols/ algorithms. The distribution subsystem is specifically designed to allow close coupling with centralized virtual machines. This will allow the integration of distribution services at the language level.”* – Pepito technical annex (p29)

## 1.2 Contribution

The primary contribution of WP4 is the implementation of the DSS middleware implementation to show the feasibility of providing generic distribution support for high-level programming languages. This is, we argue, of great benefit to the programming language community. Extending programming systems to full distribution preserving language semantics is reduced to coupling a programming system to the middleware. Potentially, this will allow many more programming languages, to be extended to distribution.

In order to fulfill the goals of WP4, to provide generic and efficient distribution support for programming systems, novelties in middleware design were required. The interface exposed by the DSS towards the programming system introduces the new concept of abstract language entities. This captures only the essential semantic qualities that have distribution consequences, other semantic properties are abstracted out. The abstract entity guarantees the semantics of operations over a set of eligible entity consistency protocols. Thus, coupling between the concrete language entity of the language and the appropriate abstract entity has to be done only once. The protocols are guaranteed to provide the correct consistency model.

The DSS implements a framework for *distribution strategies*, a distribution strategy is a comprehensive description of the protocol(s) that provide distributed access of data structures (or location-independent operation on shared language entities). In general, there are more than one possible distribution strategy for each language entity type, and the best choice in terms of distribution performance and partial failure properties is application dependent. The DSS allows this choice to be made on a per entity basis.

The framework separates different aspects of a distribution strategy into sub-protocols. The use of sub-protocols enables customization by combining different sub-protocols into distribution strategies. Moreover, implementing a sub-protocol is by definition simpler than developing a complete distribution strategy since each sub-protocol type is concerned with just a part of the functionality that makes up a distribution strategy. This is indicated by the large suite of sub-protocols available in the DSS middleware. Finally, division of distribution strategies into sub-protocols avoids what would otherwise - if the same choice was represented by distinct distribution strategies - be a combinatorial explosion in the number of distribution strategies.

Another design contribution found in the DSS is the messaging layer. In order to simplify development of the sub-protocols, the messaging layer provides a location and migration transparent process communication model. The tasks of connection maintenance and channel maintenance are located in replaceable modules. Thus, custom connection maintenance strategies, i.e. how to connect and how to detect failures, can easily be implemented. Moreover, locating the handling of channels (e.g. sockets) in a separate module caters for simple integration with different operating systems.

Finally, we have shown how middleware for programming language distribution support can make use of peer-to-peer techniques. In close collaboration with researchers from the algorithmic work package of Pepitio (WP2), the DSS was extended with an implementation of the structured peer-to-peer system DKS[10] in order to implement a self-organizing, fault tolerant directory service. The directory service was used internally in the DSS to realize a decentralized and fault tolerant home-migration protocol. In addition, we have shown how flooding based Gnutella style of algorithms can be incorporated into the messaging layer of the DSS in order to traverse NATS and firewalls. Moreover, the flooding technique is used to locate processes that have migrated, i.e. changed their addresses.

### 1.3 Overview of the DSS development

Development of the DSS middleware has been a continuous task over the last three years of Pepito. The first year was spent on developing the basic services of the DSS, reported in deliverables D4.1 and D4.2. Prominent results were the development of a first version of the abstract entity interface and the distribution strategy frameworks. The second year was focused on making the DSS robust and for dissemination. DSS was extended with secure messaging (D4.3 and D4.4) and the ability, by using peer-to-peer techniques, to traverse firewalls and locate migrating nodes was incorporated. Dissemination was primarily achieved by publishing papers describing various aspects of the DSS.

The final year, partly reported in D4.6, was devoted to integration of results from WP2 into the DSS. The structured overlay-network system DKS was made into an internal service of the DSS, and was used to implement new functionality in the DSS. In addition the DSS was used as middleware for the MBlog peer-to-peer demonstrator, reported in D5.2.

### 1.4 Dissemination

The work on the DSS has currently resulted in three papers published in conference proceedings and one paper submitted to a conference (COORDINATION 2005). In addition, a PhD dissertation describing the DSS is being compiled by Erik Klinskog, expected to be completed during the first half of 2005.

A software prototype of the DSS exists and is available for download from <http://dss.sics.se/>. The software prototype, from now on referred to as the DSS, has been successfully coupled to the Mozart system [5], resulting in the OzDSS system. Moreover, a C++ library that provides different types of distributed data structures have been developed based on the DSS. Both systems are available for download from the DSS website.

The Mblog demonstrator (D5.2) is based on the C++ library on top of DSS and is used to show how weblogs can be deployed over a structured peer-to-peer network. The demonstrator has been successfully displayed at two national Swedish events for technical transfer from academia to industry.

In an activity conducted by UCL outside PEPITO the OzDSS prototype is being merged with the official release of Mozart. The resulting system will be used as a vehicle for further research in secure distributed programming languages.

### 1.5 Pepito Integration

Within WP4, the DSS has been jointly developed by researchers from three of the partner sites, SICS, KTH and UCL. Moreover, collaboration has taken place with researchers for INRIA in the development of a scalable distribute garbage collection algorithm (D4.8).

In corporation within WP3 between SICS and UCL has the OzDSS system been developed. A Java interface on top of the DSS has been developed at EPFL, with input from SICS.

The integration of the DKS system, from WP2, in the DSS is a result of close collaboration between KTH and SICS.

Deliverable D5.2, the P2PMblog is constructed using the DSS in collaboration between people from WP4 and WP5, all working at SICS.

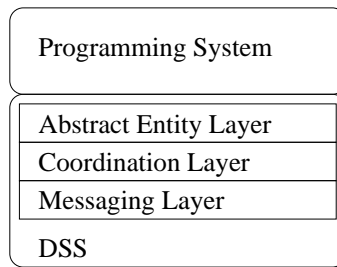


Figure 1: The DSS attached to a programming system. Note the three internal layers of the DSS.

## 1.6 Outline

This report consist of an introduction and overview of the DSS and a set of attached documents. The introduction continues with an overview of the DSS and the attached document that describes the system (2). Section 3 presents an overview of existing system that provides distribution support that could be used by programming languages. Section 4 gives an overview of the software found at the official DSS webpage. We conclude in section 5.

## 2 The Distribution Subsystem

The design of the DSS reflects the goal behind the middleware, efficient and generic distribution support for almost all high-level programming languages. First, we provide an interface that is easy to integrate with a programming system. Second, the distribution support is efficient, and by close integration with a programming system an efficient distributed programming system can be realized

This section gives an overview of the DSS and presents the various associated documents that have been placed in the appendix to this report. Each document is shortly summarized and situated in the description of the DSS.

### 2.1 Overview of the DSS

The DSS is internally realized as a three layered structure (see Figure 1). The topmost layer, the *abstract entity layer* provides the generic data structure interface that efficiently hides the internals of the library. The second layer, the *coordination layer*, implements the large suite of distribution strategies provided by the DSS. This layer can be seen as a collection of appropriately packaged distributed algorithms. The bottom layer, the *messaging layer*, provides communication support for the protocols of the Coordination Layer. The layer is also available as a standalone component.

#### 2.1.1 Abstract Entities – the API

The API of the DSS allows for sharing of programming system data structure in the form of Abstract Entities. The DSS has no knowledge of the structure of the programming system data structures. Instead, the DSS and the programming system communicate over a set of operations, e.g. install/extract state and pass operation/result. Abstract Entities are of different types (there are three main types: **immutable**, **mutable**, and **transient**), the type defines the operations that can be performed on the

shared data. The mutable abstract entity for example, allows both read and write access. The immutable allows for only read access, i.e. no updates. The transient allows for read access and one single write operation and is important to support data-flow programming languages.

### 2.1.2 The Coordination Layer

The coordination layer implements the distribution strategies used by the abstract entities to guarantee consistent access to the shared entities. A distribution strategy is divided into three different sub-protocols, each with its defined task. In reality, a distribution strategy is constructed by putting instances of the three different types of sub-protocols together. This is an expressive model since: (i) distribution strategies can be customized on a fine grained level, (ii) development of sub-protocols is easy (at least when comparing with implementing a complete distribution strategy or consistency protocol). Moreover, adding a new sub-protocol instance does not result in just one new protocol, but every combination of the instances of the other sub-protocol types. The complete range of distribution strategies provided is thus very large and this is achieved without a combinatorial explosion in the implementation.

The distribution-strategy framework, the sub-protocols, describes three different dimensions of distribution support. First, the *reference sub-protocol*, responsible for implementing a distributed garbage collection service. Second, the *coordination sub-protocol*, provides the communication and arbitration framework for a distributed language entity. Third, the *consistency sub-protocol*, implements the actual consistency protocol, such as remote execution, mobile (migratory) state, and replicated state with invalidation. By relieving the consistency protocol of the tasks dealt by the reference and the coordination sub-protocols the development of consistency protocols is much simplified.

The DSS defines a *coordination network* as being the mini-network that connects all machines that are actively referencing a distributed language entity. The structure of the network changes over the course of time, as references go out of scope, and are imported/exported. Whether or not all machines that reference an entity do so actively is dependent on distribution strategy.

The distribution strategy is executed over the coordination network. The coordination sub-protocol defines the communication pattern over the coordination network. The coordination network holds one arbitrating unit, called the coordinator. The coordinator has different roles in the different consistency sub-protocols. The coordinator is, however, identical to the home in home based protocols.

### 2.1.3 The Messaging Layer

The Messaging Layer provides a seamless communication abstraction for the sub-protocols of the coordination layer. Details regarding Internet messaging such as connection establishment and failure detection are hidden behind an interface of process identifiers. The interface provides asynchronous, FIFO communication. Detected failures are reported as events, allowing for simple separation of concerns.

Internally, the messaging layer is divided into easy to replace sub-modules. Issues regarding connection maintenance and channel maintenance are located in separate modules. Connection maintenance includes connection establishment, and connection monitoring. Channel maintenance is the low-level task of implementing socket like abstractions over the functionality provided by operating systems. Both modules increase the applicability of the messaging layer. By implementing custom connection maintenance modules, context information regarding where an application will be deployed can be easily included in the middleware. Consider the difference in connection maintenance

between an application that is to be deployed exclusively on a small LAN and an application that is to be deployed on an ad hoc wireless network.

## 2.2 Attached Papers

The DSS is thoroughly discussed in the seven documents attached to this report. Some of the documents have been attached to earlier deliverables, but we have chosen to include them in this report to give a complete description of the DSS. **Paper-1**, **paper-2**, and **paper-3** were attached to deliverable D4.9. **Paper-4** is a technical report derived from deliverable D4.4. **Paper-5** is a research paper derived from the results presented in D4.6, and currently submitted for review to the conference COORDINATION 05. **Paper-6** and **paper-7** are technical reports that describe the implementation of the DSS.

### **Paper 1: The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities**

E. Klinskog, Z. El Banna, P. Brand and S. Haridi. The DSS, a Middleware Library for Efficient and Transparent Distribution of Language Entities. In *Proceedings of HICSS'37*, Hawaii, USA, 2004.

The paper describes the design and implementation of the DSS on a conceptual level. Focus of the paper is on the interface of the DSS towards a programming system. The abstract entity model is introduced together with a description of the abstract operations interface. The interaction between a programming system level thread and an abstract entity is explained by examples. Finally, an example of how a simple data structure is coupled to an abstract entity is presented in pseudo C++ code. Moreover, the internals of the DSS are briefly described. Elements of the coordination layer are introduced and the messaging layer is described.

This paper serves as an overview paper of the DSS. It is an introduction to the approach, the design and the implementation of generic distribution support in the form of a middleware.

### **Paper 2: The Design and Evaluation of a Middleware Library for Distribution of Language Entities**

E. Klinskog, Z. El Banna, P. Brand and S. Haridi. The Design and Evaluation of a Middleware Library for Distribution of Language Entities. In *8<sup>th</sup> Asian Computing Conference*, Mumbai, India, 2003.

The second paper describing the DSS, actually presented before the **paper 1**, focuses on evaluation of the DSS approach in respect of efficient distribution support. The major contribution is evaluation of the DSS implementation and approach to efficient distribution support by choice of distribution strategy. The OzDSS system is introduced and evaluated in respect of basic messaging and capability to handle different types of distribution scenarios. More precisely, the ability to distribute language entities over various numbers of nodes and various degree of concurrency is measured. The evaluation clearly shows that the right choice of distribution strategy for a distributed data structure can improve the performance by orders of magnitude compared to a protocol that does not match the use case of a data structure.

The paper can be seen as a complement to **paper 1**, together the two papers give a complete description of the DSS on a conceptual level. However, still the messaging layer has only been briefly described. The evaluation of the DSS shows that the DSS is efficient and indicates that the possibility

to choose distribution strategy is the most important factor in order to create efficient distributed applications.

### **Paper 3: A Peer-to-Peer Approach to Enhance Middleware Connectivity**

E. Klinskog, V. Mesaros, Z. El Banna, P. Brand and S. Haridi. A Peer-to-Peer Approach to Enhance Middleware Connectivity. In *OPODIS 2003: 7<sup>th</sup> International Conference on Principles of Distributed Systems*, Martinique, France, 2003.

Providing a single system image, i.e. transparent distribution, over a set of nodes that communicate over the internet requires connectivity between the nodes. However, administrative domains, such as NATS and firewalls hinder the connectivity. In addition, processes can change their network addresses and thus hinder connection establishment until the new network address has been propagated into the distributed system. This paper shows how the implicit overly network created by a distributed application can be used by a Gnutella type of flooding approach to find indirect communication paths over the overlay network between nodes. The model is based on unique node identifiers together with possibly volatile network address information for each node of a distributed system.

The internals of the messaging layer of the DSS is described on the level of conceptual objects. The modular design of the messaging layer that allows for simple integration with different environments, both operating system environments and network environments. An example of a custom connection maintenance module is presented; the example implements the Gnutella flooding over existing connections between processes. The implementation is compared with a raw socket application in order to evaluate the overhead imposed by the functionality of the messaging layer when it comes to messaging.

Apart from the DSS specifics, the paper contributes in the description of how the processes of a distributed application can be used to create a stable system in the presence of partial failure. Together with **paper 1** and **paper 2** the paper gives a conceptual description of the DSS, from programming system interface down to the operating system interface.

### **Paper 4: Securing the DSS**

E. Klinskog, Z. El Banna and P. Brand. Securing the DSS. Technical Report T2004:14, Swedish Institute of Computer Science, SICS, November 2004.

The model of distributed language entities supported by the abstract entities of the DSS is referentially secure. A reference to a distributed language entity cannot be forged or guessed; only by proper delegation can a thread get access to language entities originating at remote processes. Referential security provides a fundament for secure distributed applications. By programmatically restricting access to distributed language entities to only trusted nodes, a distributed application can be made secure. However, for this to be true, referential security must be supported on the level of implementation. This paper describes how the DSS is made secure, how the referential secure model is preserved in a distributed setting.

Different types of attacks are described together with the means to prevent them. For instance, a process can attempt to acquire unauthorized access to a coordination network and thus access information that by the language security model it should not have. Alternatively, the process can merely try to stop functionality of a coordination network of which it is not a legitimate member.

Extensions required to the DSS to cope with the identified security breaches are presented on the level of design and implementation. A secure messaging layer, with non-forgable process references,

is the key to a secure data distribution system. This means non-forgable process identifiers; a secure connection establishment protocol based on authorization, and encrypted communication channels. The paper show how the messaging layer presented in **paper 3** is made secure according to the latter proposed extensions.

### **Paper 5: Home migration using a structured overlay network**

E. Klinskog, P. Brand and S. Haridi. Home migration using a structured overlay network. Submitted to the COORDINATION 2005 conference.

Any home-based protocol in general and the coordination-network in particular are vulnerable to coordinator/home loss and/or a suboptimal placed coordinator/home. A natural solution is to allow migration of the coordinator to move from nodes that are to be taken down or to nodes that are more optimally located. However, the challenge is to locate the coordinator after it has moved. The paper describes the design and implementation of a migrating coordinator sub-protocol that makes use of a structured peer-to-peer system, the DKS, to implement a distributed directory service that stores correct coordinator locations. The solution presented is DSS specific, but could easily be applied to any home-based protocol.

The structured peer-to-peer overlay system DKS is integrated into the DSS middleware and provided as a basic service for the sub-protocols of the coordination layer. The DKS system is used to organize the nodes of a distributed application such that a highly available, fault tolerant, distributed directory service is realized.

In addition, the paper shows the strength of the sub-protocol model. One new coordination sub-protocol is introduced to be able to migrate the coordinator. This can be used in conjunction with all the existing consistency, and reference sub-protocols to provide an additional large set of useful distribution strategies.

### **Paper 6: Internal Design of the DSS**

E. Klinskog. Internal Design of the DSS. Technical Report T2004:15, Swedish Institute of Computer Science, SICS, 2004.

This technical report is solely dedicated to the implementation of the DSS middleware. The implementation of the DSS is described with focus on the design of the contributions of the DSS, such as the abstract entity interface and the coordination layer. Key concepts are lifted forward and described, on the level of C++ classes. Examples from **paper 1** and **paper 2** that where described on a conceptual level are revisited and described at the level of the C++ implementation.

This paper, together with the three conceptual papers **paper 1**, **paper 2**, and **paper 3** gives the complete picture of the DSS library. The three previous papers give a conceptual description, and this paper shows how the concepts are realized in practice. The paper is not a straightforward interface description, but puts focuses on how selected classes of the middleware interact in order to provide service.

### **Paper 7: Coupling a Programming System to the DSS, a Case Study**

E. Klinskog. Coupling a Programming System to the DSS, a Case Study. Technical-report T2004:16, Swedish Institute of Computer Science, SICS, 2004.

The final paper describes coupling of a programming system, Mozart, with the DSS. The result, OzDSS, is described in detail. Essential when coupling a programming system to the DSS is how the internal model of threads and language entities is mapped to the model of the DSS. The model of threads and language entities of Mozart is described on a detailed enough level to explain the design choices made when developing the *glue* layer between Mozart and the DSS.

Similarly to the description of the DSS found in **paper 6** the description is on the level of C++ classes. Thus the description is a complement to the conceptual descriptions of how to couple a system to the DSS found in **paper 1** and **paper 2**. In order to show the challenges associated with different thread implementations the C++DSS system is introduced. C++DSS is a C++ library that uses the DSS to implements different types of distribute language entities in the form of C++ classes. Mozart emulates threads, thus there is no risk of multiple threads accessing the DSS simultaneously. C++DSS, on the other hand, make use of POSIX threads, thus simultaneous access of the DSS from multiple POSIX threads can happened. The fundamental differences in how threads are treated in a system that emulates threads (Mozart) to a system that make use of native-threads(C++DSS) is lifted out and discussed.

The paper is concluded by a performance comparison between the OzDSS system and other distributed programming systems. We see that the OzDSS system outperforms both Java-RMI systems and Java systems that make use of CORBA for their distribution support.

### 3 Generic Distribution Support Systems

The approach taken in the DSS, to provide generic distribution support for programming languages, is not unique. Other systems exist, as described in this section. However, they all have limitations. One limitation is that existing systems are geared towards the either the object oriented and/or the imperative programming paradigms. Little support exists for programming languages that do not adhere to those two programming paradigms, i.e. functional, and data-flow.

A limitation of most object-oriented distribution support systems is that they support only remote objects (e.g. not migratory or replicated). It is known that there are a number of usage patterns where other distribution strategies are more efficient by orders of magnitude. Another limitation exist in systems that target interoperability, losses of efficiency for distribution of data structures within the same language because of a too general data structure model. Further comparison with object-oriented distribution support systems is made below.

Finally, the architecture of the DSS, demonstrates that distribution support systems, themselves, can be built in a more modular way. New protocols, new distributed algorithms, are easily incorporated. Protocols (or distribution strategies) are divided into sub-protocols for maximum reuse. This, of course, carries over in making extending a programming system coupled to the DSS (or something similar) easier.

#### 3.1 CORBA

The Common Object Request Broker Architecture, referred to as CORBA, is not a distributed system in itself, but the specification of one. The purpose of CORBA is to provide generic object oriented distribution support for programming systems. Moreover, CORBA focuses on programming language interoperability, i.e. different object oriented programming languages can use CORBA as an interoperability fabric. Remote (stationary) objects is normally, the only distribution strategy supported. Implementations exist that provide for replicated objects, but in a more limited way (i.e. replicated

objects that cannot reference other replicated objects).

The CORBA standard is comprehensive, and can be seen as a wish-list of every functionality conceivable operation that is possible with remote components[12]. The sheer amount of functionality that has to be implemented for a distributed programming system to be CORBA compliant is a strong argument against CORBA. However, C++ implementations, such as TAO [9], of the CORBA standard exists that can be used as generic distribution support middleware.

CORBA is designed for interoperability and allows for integration of applications written in different programming languages. By the use of IDL (Interface Description Language) descriptions, objects in one application can communicate with objects in another application, disregarding their internal representation. Thus it is possible to wrap anything behind an IDL description. It is even possible to integrate non-object oriented legacy code with the CORBA standard.

### 3.2 Web Services

Web Services are an attempt to standardize application to application communication over the internet. The unit of interaction is a service that has a location and exposes an interface of eligible operations it can perform. Interaction between services is done by remote invocation using the SOAP (Simple Object Access Protocol) [13] standards, with data encoded in XML (eXtensible Markup Language) [7]. The use of those standards makes it possible for different systems to interact, thus achieving interoperability.

A service, the unit of distribution, is a larger concept than the average data structure found in programming languages. A service exposes an object type of interface, i.e. methods, thus simplifying interaction. The model is explicitly client/server oriented in that a SOAP object models a server, whose functionality is described in WSDL (Web Service Description Language). SOAP objects are stateless, i.e. invocations on a SOAP object should not manipulate the objects state. In that a method of a SOAP object resembles traditional RPC. In summary, web-services are a framework for development of loosely coupled Internet applications, not for development of distributed programming languages.

### 3.3 .Net

The purpose of the .Net framework [6] is to provide a platform for object oriented programming over the Internet[8]. Automatic loading of objects and component-descriptions at runtime is implemented as core functionality by the platform. The model of distribution, distributed objects are prepared for Internet computing by supporting both intra .Net distribution and seamless interaction with SOAP objects. Consequently a remote object can either exist at another node executing the .Net framework or be SOAP object residing at a web-server.

The foundation of the .Net platform is the common language runtime (CLR). CLR implements core functionality such as memory management, thread management, remoting and managing language security. The CLR is generic in that it executes annotated byte code. Thus, any programming language that is compiled into the byte code of the CLR can be executed on the .Net platform. In addition to the core functionality, the .Net platform implements a class library available to any program compiled to the CLR.

Using the .Net platform as a generic distribution platform for a programming language would mean taking another approach than the one we advocate. Instead of coupling distribution support to a programming system, is the programming language compiled to the .Net platform. Thus, a compiler has to be developed that compiles the programming language to the byte code of the CLR. As long

as the model of a programming language matches the model provided by .Net, this is a viable approach. In addition, the .Net platform provides marshaling routines, automatic loading of code and components. However, the platform is strongly geared towards the statically typed programming languages. Despite initial efforts from Microsoft to financially support projects that tried to compile various programming languages to the .Net platform, few of the projects were successful. In addition the model of distribution is limited to remote objects.

### 3.4 Software Distributed Shared Memory: InterWeave

Software Distributed Shared Memory(S-DSM) systems are primarily geared towards parallel computations over clusters of workstations located at a single LAN[1]. The type of distribution support, a shared virtual memory between the processes of the distributed system, fits programming languages such as C and FORTRAN. Still, the model has been applied to other programming languages, for example Java [15]. InterWeave[4] is a system that attempts to provide the S-DSM model of distribution for a wider community of programming languages in other distributed environments than clusters of workstations.

Sharing memory between processes demands a common memory representation, thus S-DSM systems have traditionally been targeting homogenous systems, i.e. processes executing on the same type of hardware, over the same type of operating system. Interweave is designed to overcome this restriction associated with the S-DSM approach. The distributed shared memory is represented in a machine independently format. The machine independent structuring is on the level of primitive types, such as integers, pointers and floats. A layer between the application and the memory translates the generic format into a machine/software specific format[14].

Another problem related to the S-DSM approach is *false sharing* that stems from granularity abnormalities between application data structure size and block size of shared memory. In order to overcome this problem, InterWeave supports distribution of dynamically sized, fine-grained memory blocks (i.e. not memory pages as common in DSM systems) called segments[4]. In order to minimize the inherent problem of shared memory systems, false sharing, a second level of memory access is provided called views [3]. A view allows manipulation of only parts of a segment. Thus minimizes the dependencies between different processes, i.e. two processes can use non-overlapping views simultaneously access the same segment. Moreover, distribution strategy is defined per segment; this caters to customization of distribution behavior, i.e. more efficient distributed applications. In addition, new distribution strategies can be added to the InterWeave system.

Even though Interweave targets distributed systems in general, not only cluster of workstations, decentralized Internet computing is not the primary deployment environment. The unit of distribution is memory segments and the distribution method is replication of memory segments. Interweave makes use of dedicated servers that maintain up-to-date copies of memory segments. In order for an InterWeave system to provide service, the servers must always be accessible. Similarly to traditional S-DSM systems, focus is on performance, i.e. minimize bandwidth utilization and latency at invocation of the shared memory.

With the ability to distinguish between memory blocks and views, whose size is application dependent, and the ability to distinguish immutable data blocks from mutable data blocks, Interweave comes much closer to the DSS in terms of the range of protocols offered than traditional DSM. However, these choices are fewer and they are not integrated into a programming language, so for the programmer there is a multiplicity of models.

## 4 Software

The work on the DSS has resulted in the middleware library as well as two applications making use of the DSS. The three different software components can be downloaded from the DSS webpage <http://dss.sics.se>.

The file `dss-1.0.tar.gz` contains the complete set of files necessary for creating an instance of the DSS middleware. Included in the package is the C++DSS library and a set of examples of how the C++DSS library can be used to create distributed applications.

Included in the software distribution is also the library written on top of DSS used in the P2PMBlog application (D5.2). The library provides a key-based-routing service and a publish/subscribe service, both constructed using the DKS implementation found in the DSS.

The OzDSS distribution, `oz_dss-1.0.tar.gz`, contains the source code for creating a version of Mozart that is coupled to the DSS.

## 5 Conclusion

The work invested in WP4 has resulted in the DSS middleware, three papers published in conference proceedings, one paper submitted to a conference, the OzDSS system, and the C++DSS library. In addition, the design, development and evaluation of the DSS is the topic an upcoming PhD dissertation (2005). Apart from the achievements above we will revisit the initial goal of WP4 and consider the work in the light of the initial objectives:

*“The objective of this work package [WP4] is to provide a language-independent distribution subsystem. This distribution subsystem will offer a wide range of distributed services/protocols/ algorithms. The distribution subsystem is specifically designed to allow close coupling with centralized virtual machines. This will allow the integration of distribution services at the language level.”* – Pepito technical annex (p29)

From the objective three goals can be derived. First “A language independent distribution subsystem”, that means a middleware that can be coupled to virtually any programming system. Second, “Offer a wide range of distributed services, protocols, and/or algorithms”, the middleware should be expressive and provide a comprehensive services when it comes to distribution support. Third, “Integration of distribution services at the language level”, distribution support should be transparently included into the programming model of a programming language. Below we discuss how the three objectives were successfully fulfilled:

**A language independent distribution subsystem.** The DSS is generic and provide programming paradigm independent distribution support in the form of abstract entities. The middleware has been successfully coupled to the multi-paradigm programming system Mozart. Moreover, the DSS has been used to implement a C++ distribution support library.

**Offer a wide range of distributed services, protocols, and/or algorithms.** The DSS provides distribution support in the form of distribution strategies efficiently hidden behind the abstract entity interface. A wide range of different algorithms/protocols have been implemented, including the DKS algorithm from WP2, and simply presented to programming systems in the abstract entity interface.

**Integration of distribution services at the language level.** The OzDSS system shows that this is achieved; it provides the same model of distribution as original Mozart. The two systems differ in how distribution support is implemented; Mozart implements distribution in the virtual machine and OzDSS is coupled to an external software library, the DSS. In reality, OzDSS is often more efficient since distribution support can be customized on a single language entity level (on the basis of the entity use pattern), something that was not possible in Mozart.

In summary, the project was successful, the goals were fulfilled. Moreover, following the pre-study of how distribution could be integrated into Mozart, resulting in the OzDSS system, and the DSS is now being integrated into the official Mozart system. The next release of Mozart will implement distribution support by hosting the DSS and offer a new type of language distribution support to the Mozart community. Not only will the programmer be able to annotate distribution behavior on a single language entity level, moreover, the Mozart system will be able to traverse asynchronous network connections (NATs and firewalls), handle mobile processes, allow for secure capability based distributed systems and provide for decentralized mobile data structures.

## References

- [1] C. Amza, A. L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. Treadmarks: Shared memory computing on networks of workstations. *IEEE Computer*, 29(2):18–28, 1996.
- [2] Joe Armstrong, Robert Viriding, Claes Wikström, and Mike Williams. *Concurrent Programming in Erlang*. Prentice-Hall, second edition, 1996.
- [3] D. Chen, C. Tang nad B. Sanders, S. Dwarkadas, and M. Scott. Exploiting high-level coherence information to optimize distributed shared state. In *Proc. of the 9th ACM Symp. on Principles and Practice of Parallel Programming*, 2003.
- [4] DeQing Chen, Chunqiang Tang, Xiangchuan Chen, Sandhya Dwarkadas, and Michael L. Scott. Multi-level shared state for distributed systems. In *ICPP'02*, 2002.
- [5] Mozart Consortium. <http://www.mozart-oz.org>.
- [6] Microsoft Corporation. Microsoft .net development. [msdn.microsoft.com/net/](http://msdn.microsoft.com/net/), 2002.
- [7] W. A. Domain. Extensible markup language (xml). [http:// www.w3c.org/XML](http://www.w3c.org/XML).
- [8] D. Q. M. Fay. An architecture for distributed applications on the internet: Overview of microsoft's .net platform. In *17th International Parallel and Distributed Processing Symposium (IPDPS 2003)*, 2003.
- [9] Fred Kuhns, Douglas C. Schmidt, and David L. Levine. The design and performance of a real-time i/o subsystem. In *IEEE Real Time Technology and Applications Symposium*, 1999.
- [10] Per Brand Luc Onana Alima, Sameh El-Ansary and Seif Haridi. Dks (n, k, f): A family of low communication, scalable and fault-tolerant infrastructures for p2p applications. In *3rd IEEE International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, 12-15 May 2003, Tokyo, Japan. IEEE Computer Society, 2003.

- [11] S. Microsystems. Java remote method invocation specification, 1998.
- [12] Ingo Rammer. *Advanced .Net Remoting*. Apress, 2002.
- [13] J. Snell and K. MacLeod. *Programming Web Applications with SOAP*. O Reilly, 2001.
- [14] Chunqiang Tang, DeQing Chen, Sandhya Dwarkadas, and Michael L. Scott. Efficient distributed shared state for heterogeneous machine architectures. In *ICDCS'03*, 2003.
- [15] Weimin Yu and Alan L. Cox. Java/DSM: A platform for heterogeneous computing. *Concurrency - Practice and Experience*, 9(11):1213–1224, 1997.