KEYSTONE: A Federated Software Environment

**Geoffrey M. Clemm*, Dennis M. Heimbigner,
Leon J. Osterweil*, Lloyd G. Williams****

Department of Computer Science
University of Colorado
Boulder, Colorado 80309

CU-CS-284-84 November, 1984

(Revised April, 1985)

*Supported in part by the U.S. Department of Energy under contracts no. DE-AC02-80ER10718 and DE-FG02-84ER13283, and in part by the National Science Foundation under grants no. MCS80-00017 and DCR-8403341.

**On leave from Hampshire College, Amherst, MA 01002.

Abstract

Individual workstations, based on powerful personal computers, suggest interesting possibilities for important advances in software development environments. The use of personal workstations to support large software development projects, however, requires interconnection via a local area network to make it possible for the individuals working on large projects to communicate, cooperate and share resources. This paper describes the design of a prototype distributed software development environment. The prototype is based on the Odin software environment. Odin provides an integrated interface to tools and makes it possible for users to easily define and add their own tools. The distributed nature of the environment is supported using the Federation concept. This approach, originally developed for distributed database systems, supports the sharing of information among autonomous databases which do not necessarily share a common schema. The Federated approach offers several advantages for software development environments. By enabling each developer to specify the form and extent of his or her cooperation, it is possible to allow a high degree of autonomy while simultaneously providing for cooperation among individuals working together on a project.

Table of Contents

Introduction 1

Issues in Developing a Distributed SDE 2
- Data Centering 2
- Extensibility 3
- Controlled Autonomy 4
- Locality 5
- Transparency 6
- Robustness 6

Related Work 7
- Jade 7
- Cedar 8
- DSEE 9

Present Work 10
- Odin 10
- The Federated Approach 13

Design 15
- Databases 16
- Odin 17
- Coordinator 17
- Cohort 18
- Policy Manager 19

Examples 20
- Example 1 20
- Example 2 21
- Example 3 22

Implementation 22

Summary 23

References 25

Figure 1 27
Figure 2 28

1. Introduction

Individual workstations, based on powerful personal computers, suggest interesting possibilities for important advances in software development environments. These workstations are capable of supporting most, if not all, of the functional activities of an individual developer including editing, compilation, analysis and testing of small modules or programs, document preparation and personal data management. Workstations also offer significant user interface advantages over more traditional technology including high-resolution graphics displays and non-keyboard input (such as a mouse or graphics tablet).

The use of personal workstations to support large software development projects, however, requires interconnection via a local area network to make it possible for the individuals working on large projects to communicate, cooperate and share resources. This paper describes a project to construct and evaluate a distributed software development environment (SDE). By "distributed" we mean that the various functions of, as well as resources for, software development are distributed over a network of workstations, one per developer.

The purpose of this project is to build an experimental prototype distributed software environment in order to better understand the questions surrounding such an endeavor and to gain insight into how the availability of such an environment might affect the software development process. The prototype environment is based on the Odin software environment [Clem84]. Odin provides an integrated interface to tools and makes it possible for users to easily define and add their own tools to the environment.

The distributed nature of the environment is supported using the Federation concept [Heim82], [Heim81]. This approach, originally developed for distributed database systems, supports the sharing of data among autonomous

databases which do not necessarily share a common schema. In a Federated system, cooperation replaces central authority and procedures are provided to support the intiation, modification and termination of data as well as functional tool sharing. Each member of the Federation specifies the information that it is willing to export and the information that it wishes to import from other members. The Federated approach offers several advantages for software development environments. By enabling each developer to specify the form and extent of his or her cooperation it is possible to allow a high degree of user autonomy while simultaneously providing for cooperation among individuals working together on a project.

2. Issues in Developing a Distributed SDE

Many of the issues which arise in developing a distributed SDE are those which arise in constructing any software development environment. Others are common to any endeavor involving a distributed software system. Here we focus on those issues, many of which are related, which are unique to distributed SDEs or which assume special significance in the construction of a distributed software development environment.

While experience and intuition provide some guidance, the resolution of these issues is by no means clear. Construction of a prototype will provide a vehicle for studying these and other issues in the design of a distributed software development environment.

2.1. Data Centering

We strongly believe that a software environment is most profitably viewed as being an information utility which gathers, stores and maintains the large, complex and interconnected body of information needed to construct and main-

tain a piece of software. Thus, an SDE is, in an important sense, an information management system devoted to providing effective and up-to-date information about an evolving software system to all those involved in creating, managing and using that system. This view emphasizes the importance of making all of the needed data readily available, and expresses the notion that tools are simply the devices needed to create data, transform it and manage it.

If the environment is to be distributed, this view suggests that the central issue is how the data is to be distributed, shared and kept up-to-date for the effective exploitation of all users. In fact this issue is one of the most important ones to which we are addressing this research effort. In this paper we propose just such a scheme for data distribution and sharing. We are less concerned with how tools are to be distributed. We suggest that most high-performance personal workstations are completely adequate to host reasonable tool ensembles.

2.2. Extensibility

We believe that in order for any software development environment to be user-friendly it must be extensible. There are so many different ways in which users perform the variety of software development tasks, that it seems inconceivable that one could produce a single, standard, canonical set of tools and aids capable of comfortably and effectively supporting everyone. Even if that were possible, we believe that users change the ways in which they get their work done as they become more sophisticated about their work and their tools. Thus, an environment which might seem satisfactory to a user at first, may well seem constraining and confining to that same user after some extended use. After a while, that user seems likely to regard such an inflexible, inextensible environment as unyielding and unfriendly. Finally, a given individual may be

involved in more than a single project at a time. These projects may have very different requirements and it is unlikely that a single toolset will provide optimum support in all situations.

In order for an environment to be considered friendly, and useful, over an extended period it seems to us that the environment must be flexible and extensible. In a distributed environment, the concept of extensibility takes on new meaning. Here each individual user must be assumed to need to make alterations and extensions to the environment's toolset at his or her own individual rate of speed and in his or her own individual way. Thus the distributed SDE must be capable of supporting modifications and extensions to some original toolset by each user, while still supporting effective communication and sharing of data among all users.

2.3. Controlled Autonomy

We have assumed that each workstation is used by a single developer who will want to control the way in which that workstation and its resources are used. Thus, it seems important that each user be able (within the constraints of the project) to determine the extent to which he or she is willing to share resources. In particular, if a user chooses to operate autonomously, we believe it is important that the user be able to do so. If two or more users agree that they would like to share such items as tools and data, the environment should support and facilitate that as well.

The need to offer autonomy requires that each workstation have its own independent operating system, a basic, fully functional SDE and copies of all tools required by all users of that workstation. In addition, autonomy requires that each user be able to configure his or her individual environment to accommodate personal needs and/or tastes. It is desirable that each user be able to

select the tools such as editors, formatters, etc. which he or she feels are most productive.

The need to provide for varying degrees of sharing, on the other hand, requires that it be possible for each user to define the form and extent of cooperation offered to others sharing the overall environment, and that the environment accommodate and facilitate this sharing. This requirement implies that the distributed SDE support the sharing of data objects and police attempts to alter shared objects so that only authorized users perform such changes. The environment should also monitor changes to shared objects so that those involved in sharing always have access to up-to-date versions. Support for the sharing of tools must assure that all users have access to tools that they specify and require, but also allow some users to have access to tools to which others may not have access, because they may not be aware of such tools or may find no use for such tools.

In addition, as a range of sharing arrangements is to be supported, the distributed SDE must provide support for the negotiation of entering into, changing and withdrawing from such arrangements. Further, if a developer is involved in two or more projects, it should be possible to adapt the environment to the requirements of each project. It should also be possible to perform this customization rapidly, easily and without the need of intervention by a system administrator.

2.4. Locality

Locality in this context refers to both the location of objects within the system and to the specific node on which a process might be executed.

The location of objects (files) in a distributed SDE presents some interesting questions. If objects are located only on the node which creates them,

partitioning of the network will make those objects unavailable to some users. On the other hand, if objects are allowed to migrate to other nodes, there may be problems with consistency from node to node.

The possibility that a process may execute on any one of several nodes opens the questions of which node is most appropriate and how such decisions are made. For example, a remote node may request a file for editing using an editor which actually edits a temporary copy of the file. In this case it may be most appropriate to send a copy of the file to the remote node for editing and replace the local copy when the session is completed. If the request is for an object module derived from several source files, it may be better to perform the compilation locally and send the object version to the remote node. This will reduce the number of internode I/O requests.

2.5. Transparency

Users cooperating on a project frequently need to communicate and access data in each others' directories. It should not be necessary for users of a distributed SDE to have detailed knowledge of the structure of the network to accomplish these tasks. Naming conventions should, insofar as possible, be consistent with those provided by the environment on an individual workstation. Location and retrieval of files for use in constructing a software system should be performed by the system with no intervention from the user.

2.6. Robustness

The environment as a whole should provide continuous service in the event of failure by one or more nodes or partitioning of the network. In order to continue working in cases of failure, each node must have its own independent operating system, a fully functional environment and copies of all tools

used by that node. These requirements have already been noted under "autonomy." In addition, it must be possible to preserve access to remote data in the face of failures.

Since it is important to be able to access global objects in spite of failures, we believe that a mechanism which relies on a file server is inadequate. We have instead chosen an approach in which multiple copies of objects may exist and copies may be provided by more than one source. This approach increases the likelihood that an object will be available even if its primary provider is out of service. Of course, the ability to replicate objects and access them from more than one source raises questions of consistency and propagation of access rights. As noted above, this issue is one of the most important ones addressed by this project.

3. Related Work

A great deal has been written about software development environments [Ivie77], [Hunk80], [Oste83], [Sand78], [Wass81]. Relatively little, however, has been done in the area of distributed SDE's. Although general purpose distributed computing systems such as LOCUS [Walk83] or the Eden project [Jess82] might support software development with the addition of the proper tools, that is not their primary goal.

Three projects which are more directly related to the present work are summarized below.

3.1. Jade

Jade [Witt83] is an experimental environment under development at the University of Calgary. It is intended to support the design, construction and testing of distributed software for both embedded and non-embedded systems.

Plans for the Jade environment include a variety of general purpose tools such as editors, compilers, documentation facilities and source code control, as well as tools specifically oriented toward the development of distributed software systems. Support for distributed software includes: interprocess communication facilities (Jipc); simulation capabilities (using Simula); and non-procedural specification and proof tools (using Prolog). There are also plans to include the ability to perform animated simulation of systems using high-resolution graphics terminals.

The Jade software is itself distributed, constructed as an extension to UNIX*. The emphasis, however, is on support for development of distributed software and the primary purpose of the distributed capabilities of Jade are to support simulation and testing of distributed systems. While Jade terminals will have some limited processing capability to support animated simulations, they are not planned as stand alone workstations and must be connected to the Jade host computer. Thus, although Jade is supported by a distributed system, the software development environment which it provides is in reality both conceptually and physically centralized.

3.2. Cedar

Cedar [Lamp83] is a distributed workstation environment used at Xerox PARC's Computer Science Laboratory. Cedar supports system development using the Cedar language (derived from Mesa) and the Cedar system modeller enables a developer to build a program by specifying the modules to be used in assembling the system (the "model"). When a new version of a module is created, the modeller is informed by the editor and the new version is automatically included in the working model. The modeller also provides the capability

*UNIX is a Trademark of AT&T Bell Laboratories.

of locating and retrieving files (modules) from remote nodes.

Cedar provides a true distributed SDE. Each user can have a personal workstation and the Cedar system supports communication and sharing of resources. The environment provided by Cedar is somewhat limited, however, by the fact that the Cedar editor and compiler are necessary to inform the modeller of changes in source modules and recognize Cedar binary modules. While it is in principle possible to extend Cedar with the addition of other tools, these tools will require modification to interface with the modeller.

3.3. DSEE

The DOMAIN Software Engineering Environment (DSEE) [Lebl84] was developed at Apollo Computer to support software development in a distributed workstation environment. The DOMAIN system is a host/target environment which provides a wide array of tools including source code control, configuration management, task and advice management and the ability to monitor dependency relationships. The DSEE History Manager provides source code control, including parallel development on distinct branches, and informs other instances of DSEE managers of the existence of new versions of objects. The Configuration Manager enables a user to build a "system model" which specifies the components of the system. The specific versions of the components to be used are specified in a "configuration thread" which is bound at the time that the system is assembled. Different versions or releases of a system can be built by specifying different configuration threads. DSEE does not require that a particular editor or compiler be used and third-party tools for design, verification, and cross-machine development may be added to extend the environment.

DSEE provides a powerful environment which supports software development on a network of distributed workstations. The environment is extensible via the addition of new tools, including those supplied by third-party developers. In order to obtain this extensibility, however, parts of DSEE were implemented directly in the operating system. It is therefore necessary to have detailed knowledge of the operating system to add a new tool. This makes it likely that tools are likely to be added by system administrators or even the operating system manufacturer rather than by individual users, making the environment difficult to customize to accommodate individual needs, tastes or styles.

4. Present Work

This project is addressing the issues described above by constructing a prototype distributed SDE. This environment will be based on an existing centralized environment augmented with capabilities borrowed from the realm of distributed database management. It will enable users on separate nodes of a network to communicate and share resources in a controlled fashion and it will make it possible for each user to define and add their own tools. The Odin system which forms the basis for this prototype and the Federated approach to databases are described below.

4.1. Odin

Odin [Clem84] is a system developed to provide an extensible software environment. While the Odin environment presently contains a number of tools, Odin provides very few tools of its own. Those tools that are provided (internal tools) are intended to support the addition of user-defined (external) tools. Odin enables the user to extend a host file system through the addition

of user-defined file types and operations on those types. This capability makes it possible to integrate existing tools into Odin without modification. The basic objects in the Odin system are files and tools are considered to be transformations or derivations that convert an object (file) of one type into an object of another type. The results of derivations are maintained automatically by Odin and the result of a derivation is available for satisfying subsequent requests. An object-oriented command language is provided for manipulation of objects and a specification language is provided to facilitate addition of new file types and operations. A subset of Odin is presently being used as the command interpreter and tool integration mechanism for the TOOLPACK/IST project [Oste83].

The Odin file system recognizes two types of objects: simple and compound. Simple objects in Odin correspond to host system files while compound objects are sets of files. The set of files that comprises the source code for a single program is a compound object; the individual files are simple objects. Objects in Odin can be further classified into two subtypes: primitive and derived. Primitive objects correspond to host system files. Derived objects are files which can be produced from atomic objects (or other derived objects) via the use of one or more tools.

Derivations in Odin are controlled using an internal directed graph, called a "derivation graph," which specifies how the various file types known to Odin are related. Each node in the derivation graph represents a file type (denoted by an extension such as .c or .txt) and each edge represents the tool required to produce the object at its tail from the object at its head. If a needed object is missing or obsolete, it is constructed from its constituents by application of the appropriate tools as specified in the derivation graph. It is possible for the user

to specify the maximum amount of space to be occupied by Odin files. When that size is exceeded, derived files are automatically deleted using a least-recently-used strategy.

The specification language allows the user to easily extend the derivation graph by defining new file types and their associated operations. A pretty-printer might be specified as follows:

```
fmt "Result of C pretty-printer"
        "pp <$(src_in) >$(src_out)"
                :c
```

In this example, fmt is the result of applying the pretty-printer. The string in quotes on the first line describes this object. On the second line, pp is the name of the host system command which invokes the pretty-printer; "$(src_in)" and "$(src_out)" are replaced with the appropriate host system file names at run-time. On the final line, "c" defines the type of file that is accepted as input to "fmt."

The Odin command language provides two operations on Odin objects: display and transfer. The display command is invoked by simply naming the object. For example,

```
example.c
```

would display the file named "example.c" from the current working directory.

```
example.c:run
```

would display the result of compiling and executing the file "example.c."

The transfer command is used to copy the contents of one object into another. The objects are separated by a right angle-bracket and the copy occurs in the direction implied by the arrow. For example,

```
example1.c > example2.c
```

places a copy of "example1.c" into "example2.c."

An extension of the transfer command makes it possible to interface to host system commands for use as "filters" or "editors." For example,

example.c:fmt > :wc

would display the result of running the system command "wc" with the formatted version of example.c as its input while

example.c > :emacs

would invoke the host system editor "emacs" on "example.c."

4.2. The Federated Approach

The Federation concept was first proposed in the context of database systems [Heim82], [Heim81], but the basic approach is also applicable to other distributed systems that share and exchange information. The Federated architecture has two principal features: substantial autonomy for components and an integrated set of inter-component sharing facilities. Autonomy is supported by allowing each component to explicitly state which information it will share as well as which information it will access. To support sharing among components this architecture substitutes cooperation plus communication for the global data manager plus subordinate components of traditional distributed databases.

A Federated database is organized as a collection of components representing individual users, applications, or information systems that wish to share information. Each component has significant interior structure in support of the Federated architecture. In particular, each contains three databases: the *private database,* the *export database,* and the *import database.*

The private database describes that portion of a component's data that is stored locally. The bulk of this information is devoted to describing the application data available in the component's own database. Much of this information will always remain local, but some of it will be exported to other components.

The export database specifies the information that a component is willing to share with other components. This database is actually derived from the private database. The two may, however, not be identical because it is possible to export only a portion of the private database.

The import database is the complement of the export database. It specifies the information that a component is using from other components. Normally, importing the description of some object does not imply the physical movement of that object to the importing component. When the object itself is referenced, the importer requests the actual data from the exporter. Sometimes, that data may be referenced so frequently that it is copied over to the importer for storage. This copy can be specified to have three degrees of consistency vis-a-vis the original data:

(1) It can be frozen, in which case, it is never updated.
(2) It can be periodically updated. To do this, the importer must also specify a policy that determines the frequency with which consistency is checked. At the importer's convenience, the exporter can be interrogated for modifications.
(3) It can be kept as consistent as possible. In this case, any modifications by the exporter are transmitted to the importer.

The import database is enlarged by a process of *negotiation* in which one component coordinates its import of information with the exporter of that information. A negotiation is a multi-step, distributed dialog between two components. The dialog is represented by a specialized graph stored at each component. The nodes of the graph correspond to actions by some component,

and the edges are the allowed transitions. The negotiation is distributed in the sense that the node actions are carried out alternately by the two components involved in the negotiation.

The Federated architecture provides a novel model for distributing a software environment. It seems natural that a collection of programmers desiring to share programs and data files should control the sharing by setting up individual components and by agreeing among themselves about which data objects are to be shared and how they are to be shared. For example, some objects might by mutual agreement be made public for all to see, but not modify. Other data objects might be ceded to a coworker for modification. Still other data objects would remain private and protected.

A major objective in designing a Federated software environment is the identification of the objects to be shared. In Odin, the basic objects are atomic files of information (source code, text, data) from which Odin can derive additional objects (files) by applying a sequence of tools. Both the atomic files and the derivations are potentially shared. Thus, if Odin is to make use of imported derived files, it must see not only the final derivation, but also the derivation path itself. In a Federated environment, the private database of each component would consist of the standard collection of atomic files and Odin derived objects. The export database would specify those atomic files and derivations that a component is willing to share with others. The import database would contain the names of the corresponding set of exported files and derivations actually used by a particular component.

5. Design

The architecture of the distributed SDE consists of four types of process on each workstation. The relationships among these processes are shown in

Figure 1.

(1) *Odin* interprets user requests and manages the local file system. The Odin process may be invoked by the user or by a cohort process running on the same node.

(2) The *coordinator* process manages the local end of network transactions. The coordinator may only be invoked by an Odin process running on the same node.

(3) The *cohort* handles requests from remote nodes and maintains databases which identify global objects and keep track of which nodes have requested a given local object. The cohort typically communicates with remote coordinators and remote policy managers.

(4) The *policy manager* provides an interface for initializing and maintaining the import and export databases. The policy manager is invoked by the user.

5.1. Databases

The Federated SDE uses four databases to control the sharing of information and provide network transparency.

(1) The *export list* contains a series of (object, importer) pairs and is maintained by the cohort. The coordinator uses this database to determine which nodes have received an object so that changes to global objects can be propagated correctly.

(2) The *permissions list* contains triples which describe an object, an importer, and the level of permission granted to the importer for that object. The permissions list is maintained by the policy manager and is consulted by the cohort to determine if a given request is valid.

(3) The *import list* pairs global names with exporters. It is used by the coordinator to determine which nodes export a given object. The import list is maintained by the policy manager, the cohort, and the coordinator depending on whether the information is provided by the user, received from a remote policy manager (in response to object moves) or received as the result of a broadcast request by the coordinator.

(4) The *translation table* is used by the coordinator to translate local names into global names when requesting an external object. It consists of (local name, global name) pairs which are maintained by both the policy manager (in response to user policy decisions) and the cohorts (when an object is permanently moved from one site to another).

Together the export list and the permissions list make up the export database needed for the Federated approach. The import list is the equivalent of the import database. The translation table provides network transparency for

the file system.

5.2. Odin

The principal functions of the Odin process are those described above. Odin is responsible for interpreting commands from the user, performing derivations on local objects and maintaining the local object database. If Odin is unable to locate or derive an object, it passes the request to the coordinator to see if the object (or an object from which it can be derived) is available from another node. To insure that changes in global objects are visible to other nodes, Odin is also responsible for informing the coordinator when changes are made to shared objects.

In addition to the user-invoked Odin, each node may have one or more Odin processes created by the cohort to satisfy requests from remote nodes. These processes receive their commands directly from the cohort, perform the required derivation and signal the cohort when the derivation is complete.

5.3. Coordinator

Each node has one coordinator process which is invoked as needed by the local Odin process. The coordinator is responsible for handling requests for non-local objects. When a request is received from Odin, the coordinator must translate the local name to a global one, consult the import list and poll selected cohorts on other nodes to determine if the object (or a precursor) is available.

When Odin requests a non-local object, the coordinator consults the import database. If either the object or something that can be used to derive it is in the database, the coordinator then transmits messages to cohorts on the appropriate nodes. Responses from the cohorts indicate what objects are avail-

able together with an estimate of the cost associated with providing them. If only one node is able to fill the request, the coordinator simply requests the object, stores it in the local file system and returns an OK status to Odin. If more than one node can fill the request, the coordinator selects the node that provides the least expensive means of obtaining the object. For example, if the request is for an object-code module and one node can supply the compiled version while another can supply only the source, the coordinator would send the request to the node which can supply the object code. Once the object has been received, the coordinator stores it and informs Odin of success as before.

If the requested object is not in the database, the coordinator broadcasts a request for the object. If the object is available, the coordinator updates the import list and proceeds as before. If the object is not available, a FAILED status is returned to Odin.

When changes are made to global objects, the coordinator consults the export list and notifies all affected nodes.

5.4. Cohort

The cohort is responsible for evaluation of remote requests. Requests are divided into two parts. First, there is an information request to determine the cost of obtaining an object. Second, the requester may ask for a particular object.

When a derivation request for an object is received from a remote coordinator, the cohort must determine how much of that request can be satisfied at this node, and at what cost. There are two steps to this process:

(1) The cohort checks to see if the object or a precursor is available by invoking a local copy of Odin. For each object in the derivation chain for the desired object, Odin estimates the cost to construct that object. If the cost is zero, then the object already exists. If the cost is infinity, then

Odin is not capable of constructing that object.

(2) Given the Odin costs, the cohort consults its export database for each object in turn to see if access has already been granted, and under what policy conditions. This results in a modified cost function, which is returned to the requesting node.

If the object itself is requested, the cohort transmits the object after invoking an Odin to perform any necessary derivations. When an object is sent to another node, the cohort also updates the export list.

5.5. Policy Manager

The policy manager process is the means by which the user specifies the policies to be enforced vis-a-vis the Federation. There are four major policy activities:

(1) The user may insert new elements into the translation table and the import list by engaging in a negotiation with another user to obtain access to some exported object.

(2) The user may modify his or her import list to change the preference order for obtaining a given global object.

(3) The user may insert new elements into the permissions list by specifying an object to be exported, the names of the nodes that may import it, and the access rights to be granted to each of these nodes. Access rights may be combinations of read, write, and resource-based constraints (such as load_average).

(4) The user may negotiate the movement of an object from his node to another node. This will result in further negotiations with the importers of the moving object so that they may update their import lists and translation tables.

The user may perform these actions by invoking the policy manager process. The manger converts commands into the appropriate modifications of the system databases, possibly after negotiating with other users through the remote cohorts.

The policy manager may also, at the request of the user, "move" an object from one node to another. Such a request might be generated because a user no longer wants to maintain an object in his or her file system but others still

depend on that object. The "move" request might result in the object being archived on a designated machine with large disk capacity or transferred to a node belonging to another user. In either case, the policy manager would update the local database and inform other nodes of the change.

6. Examples

To illustrate the possible operation of the Federated environment, we present three example scenarios. In the first, we show how a user simply accesses an imported object. In the second, we show how updates might occur. In the third, we show how a node failure might be handled.

6.1. Example 1

Consider the situation where Node N1 wants to access its object named "/tmp/a.out:run" To do this, the command

/tmp/a.out:run

is passed to Odin, which determines that it does not have the "/tmp/a.out:run" object or any of its precursors (e.g. "/tmp/a.out"). As a result, Odin asks its coordinator to obtain the object. The coordinator looks in the translation table (Figure 2a) and finds that the prefix of the requested object is an alias for the imported object "/root/x" at node N2. Using this global name, the coordinator consults the import list (Figure 2b) to determine possible sources for the total object ("//N2/root/x:run"). It finds that it has two choices, one for the whole object, and one for its precursor. Using this information, the coordinator asks both sources for the whole object.

On node N2, the cohort receives the request from N1 for the object "//N2/root/x:run." The cohort first invokes a local Odin to determine the costs of producing object "/root/x:run" and its precursors. Then the cohort consults

the export database to obtain a modified cost for each object. These costs are returned to N1. Similar operations are performed by N4, but it will only return the cost of "//N2/root/x:run."

As a result of its requests, the N1 coordinator might get the following information:

Node	Object	Cost
N2	/root/x	0
N2	/root/x:run	5
N4	/root/x:run	3

Suppose that N1 decides to ask N4 for "/root/x." It asks the N4 cohort for the object, that cohort will ask its Odin to materialize the object, and then the cohort arranges to ship the object back to the coordinator at N1. As the final step, N1 accepts the object from N4 and informs its Odin that the object is available.

6.2. Example 2

In this scenario, node N1 attempts to edit an external object. This demonstrates both update and change notices.

Suppose that the user gives the command

/fse.doc > :emacs

to edit the local object "/fse.doc." Odin then observes from its translation table that "/fse.doc" is the local name for the file "/root/sys.doc" owned by the node N2. In this case N1 needs to obtain modification permission from N2. Since only the owner, N2, can provide edit permission, no other cohorts are contacted. If N1 has cached a local copy of the file, then it only requests update permission from N2. Otherwise, N1 simultaneously requests both update permission and a copy of the object.

When the cohort at N2 receives the modification request, it consults its permissions list, and since the action is allowed, responds by granting permission (and transmitting the object if necessary). Eventually, the N1 coordinator returns the modified object. Since the object has been altered, N2's cohort must send a change notice to all of the importers of the modified object. It obtains this list (nodes N1 and N6) from its export list (Figure 2d) and sends appropriate messages to those nodes.

6.3. Example 3

The first two scenarios represent the normal sequences of events in the environment. Sometimes exceptional conditions occur, such as when a node fails.

In this example, node N1 asks node N2 for object "/usr/include/stdio.h" but node N2 has failed. In this situation N1 has two alternatives. First it can examine its import list to see if any other sources are available. If there are none, it can broadcast to all nodes to ask for the object. The answers (if any) are then stored in the import list as alternative sources. Examining Figure 2b, we see that no alternatives are listed. Thus, the coordinator broadcasts a request for the object and if lucky, receives a response, say from N5. The coordinator then modifies its import list to record the fact that N5 is also a source. Finally, the coordinator at N1 sends a message to the cohort at N5 to obtain the object. As a result, N1 can continue to function even though its primary source, N2, is not accessible.

7. Implementation

The prototype environment is being constructed on a network of Sun Workstations under version 4.2 of Berkeley UNIX. The implementation is

being performed in two stages.

During the first stage, the existing Odin software is being combined with parts of a distributed database that was implemented during the spring of 1984 [Heim84]. The distributed database provides many of the functions of the coordinator and cohort. This phase will provide a basic environment for communicating and sharing resources. All of the functions of Odin will be available so that users will be able to add their own tools and, to a large extent, customize their local environments.

Initially, requests for remote objects will be handled in a very simple fashion. When an object is requested, the cohort will check to see if it is present and, if so, inform the coordinator that it is able to supply that file. Otherwise, the request will fail.

Software to support negotiation between the coordinator and cohorts will consist of a database management system built on the Federated model and a simple programming language to enable a user to specify parameters for the negotiations. Development of these components will proceed in parallel with the initial integration of the existing pieces.

During the second stage the new negotiation subsystem will be combined with the basic distributed SDE. A multi-window user interface will also be added at this time to provide a convenient means of interacting with both Odin and the negotiation subsystem.

8. Summary

We have described the design of a prototype distributed software development environment. The system combines the user interface and extensibility of the Odin environment with the novel data sharing capabilities of the Federated

architecture. Users of this environment will be able to customize their local environments by defining and adding their own tools. They will also be able to specify both the form and extent of their cooperation with other users.

Construction of this environment will provide a vehicle for investigating the issues surrounding the design and use of distributed software development environments. Questions to be explored include: what are the trade-offs associated with various negotiation strategies; which approaches to storing objects on various nodes, moving objects and performing derivations are most effective; and how will the availability of such an environment influence the software development process.

9. References

[Clem84] Clemm, G. M., "ODIN - An Extensible Software Environment," University of Colorado, Department of Computer Science Technical Report CU-CS-262-84, 1984.

[Heim84] Heimbigner, D. M., "Implementing optimistic concurrency control for a distributed database", University of Colorado, Department of Computer Science Technical Report CU-CS-281-84.

[Heim82] Heimbigner, D. M., *A Federated Architecture for Database Systems,* Ph. D. Thesis, University of Southern California, also available as Technical Report TR-114, August 1982, Computer Science Department, University of Southern California.

[Heim81] Heimbigner, D. M. and D. McLeod, *Federated Information Bases - A Preliminary Report, Infotech State of the Art Reports,* Volume 9, Pergamon Infotech Limited, Maidenhead, U. K., 1981, Pages 383-410, M. Atkinson, ed.

[Hunk80] Hunke, H. ed., *Software Engineering Environments,* North-Holland Publishing Company, Amsterdam, 1980.

[Ivie77] Ivie, E. L., "The Programmer's Workbench," *Communications of the ACM,* vol. 20, no. 10, pp. 746-753, 1977.

[Jess82] Jessop, W. H., et al, "An Introduction to the Eden Transactional File System", University of Washington Technical Report 82-02-05, 1982.

[Lamp83] Lampson, B. W. and E. E. Schmidt, "Organizing Software in a Distributed Environment," *SIGPLAN Notices,* vol. 18, no. 6, pp. 1-13, 1983.

[Lebl84] Leblang, D. B. and R. P. Chase, Jr., "Computer-Aided Software Engineering in a Distributed Workstation Environment," Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Engineering Environments, pp. 104-112, April 1984.

[Oste83] Osterweil, L. J., "Toolpack - An Experimental Software Development Environment Research Project," *IEEE Transactions on Software Engineering,* vol. SE-9, no. 6, pp. 673-685, 1983.

[Sand78] Sandewall, E., "Programming in an Interactive Environment: The "LISP" Experience," *Computing Surveys,* vol. 10, no. 1, pp. 35-71, 1978.

[Walk83] Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System", *Proceedings of the Ninth Symposium on Operating Systems,* October, 1983, published as *Operating Systems Review* vol. 17, no. 5, pp. 49-70, 1983.

[Wass81] Wasserman, A. I. ed., *Tutorial: Software Development Environments,* IEEE Computer Society Press, New York, 1981.

[Witt83] Witten, I. H., G. M. Birtwistle, J. Cleary, R. D. Hill, D. Levinson, G. Lomow, R. Neal, M. Peterson, B. W. Unger, and B. Wyvill, "Jade: A Distributed Software Prototyping Environment," *Operating Systems Review,* vol. 17, no. 3, pp. 10-23, 1983.

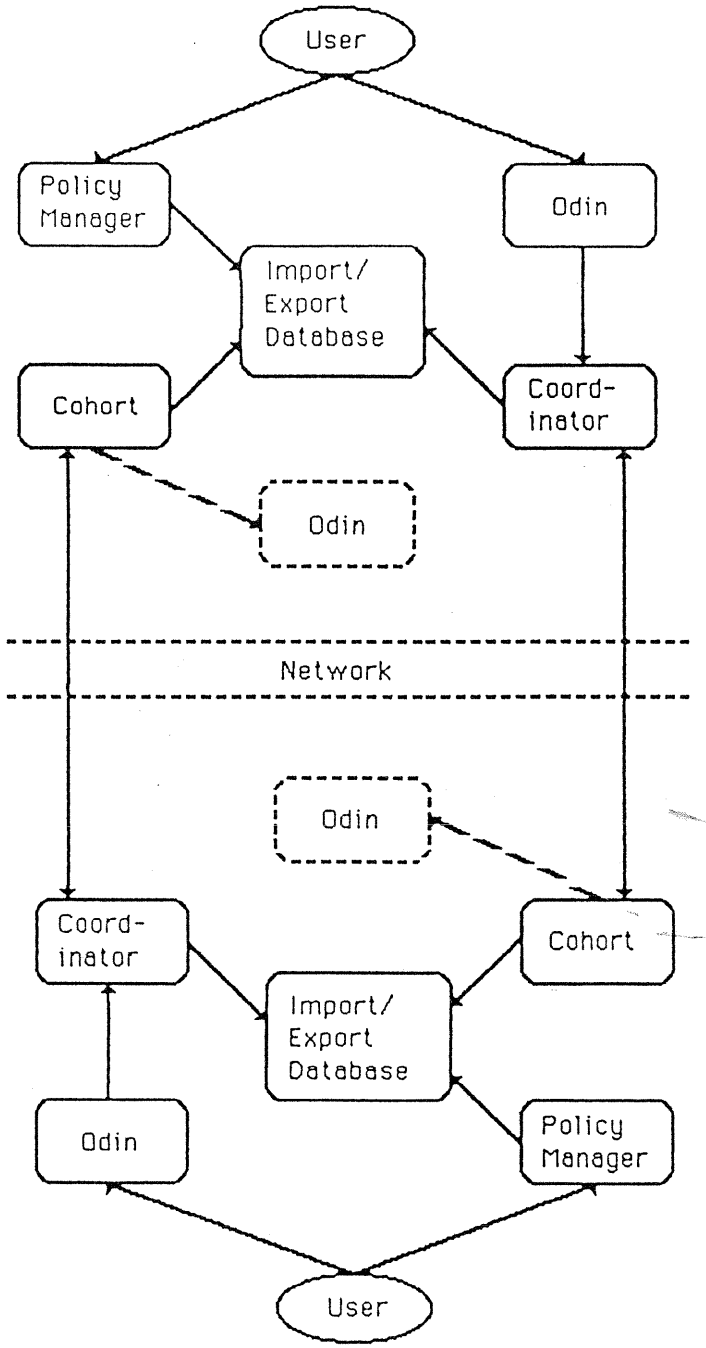

Figure 1. Process Diagram. KEYSTONE processes for two nodes are illustrated together with intra- and inter-node communication paths and database access. The Odin process in the dashed box indicates a copy of Odin which may be invoked by the Cohort to satisfy a network request.

LOCAL NAME	GLOBAL NAME
/tmp/a.out	//N2/root/x
/include/misc.h	//N2/usr/include/stdio.h
/fse.doc	//N2/dennis/fse.nroff

Figure 2a. Translation Table at node N1.

OBJECT	EXPORTERS
//N2/root/x	N2
//N2/root/x:run	N4
//N2/usr/include/stdio.h	N2
//N2/dennis/fse.nroff	N2,N6

Figure 2b. Import list at N1.

OBJECT	PERMISSIONS
/src/ddb/ccs.c	READ(N3,N6),WRITE(N6)
/root/x	READ(N1),WRITE(N1)
/root/x:run	SAME AS /root/x
/usr/include/stdio.h	READ(*)
/src/ddb/ccs.c:run	IF(load_average < 5) READ(N3,N6)
/dennis/fse.nroff	SAME AS /root/x

Figure 2c. Permission List at N2.

OBJECT	IMPORTERS
/src/ddb/ccs.c	N3,N6
/root/x	N1,N4,N6
/dennis/fse.nroff	N1,N6
/usr/include/stdio.h	N1,N5

Figure 2d. Export List at N2.

Figure 2. Example Databases