Naming of Objects in the Cluster System

Francis C. M. Lau
Computer Studies Department
University of Hong Kong

James P. Black
Department of Computer Science
University of Waterloo

Eric G. Manning
Faculty of Engineering
University of Victoria

Research Report CS-87-52
September 1987

Naming of Objects in the Cluster System

Francis C.M. Lau

Computer Studies Department
University of Hong Kong

James P. Black

Department of Computer Science
University of Waterloo

Eric G. Manning

Faculty of Engineering
University of Victoria

ABSTRACT

This paper introduces the Cluster Model as the basis for a new programming methodology, focusing in particular on the issues related to the naming of clusters (or hierarchies of processes) for efficient cluster-related operations. In this model, clusters replace conventional processes to provide a tool for imposing structure on process-based programs *at runtime*. To prove the practical feasibility of the model, a Cluster System has been built; the implementation is based on the principle of policy/mechanism separation. All cluster operations are implemented as policy routines which are subsequently mapped into kernel calls. Inter-process communication is exclusively by messages, using arbitrary names (rather than numeric process identifiers) as addresses; this avoids the need to know names internal to a cluster in order to communicate.

Correspondence: Dr. J. P. Black, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.

E-mail: jpblack@nith.waterloo.cdn

1. Introduction

Software systems are constantly evolving towards greater size and complexity. The *software problem* [6] is becoming more and more acute as this evolution continues, and as the techniques for engineering and managing software fail to keep pace. We focus on a particular piece of software, the operating system, which is the heart of any software system. We present a model of a novel operating system; such an operating system can serve as an effective basis for the development of complex software. The model is called the *cluster model*, and software built from it is called a *cluster system*. The distinguishing feature of a cluster system is its support for the structuring of process-based user programs *at runtime*. Roughly, the cluster model provides a hierarchical generalization of conventional processes, supported by a novel naming and message passing mechanism called the rendezvous store.

A cluster system, at runtime, has within its boundary a number of executing clusters. A *cluster* has within its boundary a set of one or more processes and a (private) *name space*. A cluster may also contain other clusters, therefore forming a hierarchical structure representable by a tree. A cluster has a *boundary*, and associated with the boundary is its *visibility*. Visibility dictates whether processes inside can *see* (*i.e.*, can name other processes or clusters) outside or beyond the boundary, and whether processes outside can see inside. Operations which occur across cluster boundaries must have appropriate visibility as a prerequisite. There are operations to create a process or a cluster of processes at chosen locations in a hierarchy, to migrate a process from one cluster to another (change of membership), to add or delete processes to or from a cluster, to allow clusters to communicate with one another using messages, *etc.* Messages, like processes and clusters, can be named; their names are called *tags*. A *send* operation causes a message to exist in a particular name space with a specific tag as name, and a corresponding *receive* operation names the message (using a matching tag) and causes it to be received. The use of tagging avoids any need to know a lower-level process name or ID in order to communicate.

Many trees of clusters may exist at the same time in the system. Processes may belong to (*i.e.*, have memberships in) one or more clusters in the same tree or in different trees. Trees are organized into different *dimensions* (hence a process may also appear in different dimensions), and every tree exists in exactly one of the dimensions. Dimensions can be viewed on the one hand as different ways to cluster processes that exist in the system; on the other hand, they represent different "universes of discourse" for process interaction. For example, one dimension may be for client–server interactions, and another for processes belonging to the same user.

In the next section, we give an overview of the cluster model, followed in Section 3 by some comparisons of this work with a variety of related previous work. Section 4 then discusses the cluster model in more detail, which is supported by some implementation considerations discussed in Section 5. Section 6 contains some conclusions and ideas for further work.

2. The Cluster Model

The primitive unit of computation in the cluster model is a process and the primitive unit of abstraction is a cluster. A cluster consists of a set of one or more processes and clusters, and a name space. It incorporates notions of structuring and visibility.

Unlike synchronizing resources [5] or Ada packages [12], there are no shared variables. This no-shared-variables and message-only model facilitates implementation in a distributed environment where there is no shared memory. Processes in one cluster or in a number of clusters communicate with one another using messages. Messages, like processes, are entities that exist or are associated with particular name spaces. A message may be named (using a tag), in much the same way as a process or cluster is named. Therefore, a process must utimately specify both the name of a messgae, and the name space into which it is to be sent. The cluster model possesses two unique features which distinguish it from more conventional programming models: *dynamic clustering* and *runtime modularization.*

2.1. Dynamic Clustering

During execution, a cluster system consists of many running clusters. Some of these clusters are related and form trees or hierarchies of clusters. Unrelated clusters may be in different dimensions or may belong to different trees in a single dimension. A process is created within and becomes a member of one or more clusters. During the course of its execution, it may enroll in new clusters, withdraw from clusters, or migrate to other clusters. All of these changes are subject to *visibility rules*. Likewise, cluster communications are controlled by visibility rules. Visibility rules are similar to, but more flexible than, scoping in programming languages. For example, using the tree representation, a process in a particular block in the programming language world can only see (strictly) upward; however, a process in a cluster may have a visibility that allows it to see downward (*information disclosure* versus *information hiding*), or even prevents it from seeing upward. The inability to see the name space of a particular cluster prevents a process from using the name space as a target for process creation, change of membership, or communication. A cluster is created with a specific visibility which allows its processes to see certain other clusters; details depend on the application.

Clustering is dynamic: clusters and processes can be created and deleted at runtime. Reconfiguration can be achieved via migration and multiple memberships.

2.2. Runtime Modularization

In many programming languages, modularization occurs only during the design phase, or as the program is written; the program is then compiled into a single unit with modularity information discarded, and is later run as a single process. We call this *compile-time modularization*. This approach has value; much of the program can be checked against a set of rules regarding modularization and errors can be identified at compile time. The problem, however, is that no true parallelism (*i.e.*, as parallel processes at runtime) is usually assigned to such a program since it is run as a single process by the operating system. Such is the case in Modula-2 [40], where a process in the original program is implemented by a co-routine at runtime. Therefore, to have N processes running at runtime, N programs (or N instances of a program) must be executed. Moreover, the benefits of modularity (enforcement of rules, error detection, reconfiguration,

etc.) are lost at runtime.

Our approach advocates runtime modularization in addition to compile-time modularization, and a different concept of programs. A *distributed program* is made up at runtime of a number of processes and is suitable for running in a distributed environment. A programmer constructs a distributed program by writing the code modules for the individual processes that make up the program. These code modules have the form of abstract objects and can readily be borrowed by other programmers for constructing their distributed programs. During the execution of a distributed computation, new code modules may be programmed, and added to the executing computation as processes. The programmer specifies where (*i.e.*, in which clusters) the new processes should be placed. This kind of runtime creation and modularization augments compile-time modularization, and yields a system that has *both* static and dynamic structure.

Because of runtime modularization, certain language features for expressing parallelism can be dispensed with. In fact, with our design, any traditional non-concurrent language is sufficient for producing the necessary process configuration for a distributed computation. Of course, if finer-grained parallelism is needed, such as the parallel command in CSP [19], one might still want to make the language "parallel". But this latter parallelism will have to exist inside a process itself (*i.e.*, pseudo-parallelism if a process is the primitive execution unit) and therefore is beyond the scope of our work here. If processes and message passing can be made sufficiently cheap in a cluster system, the granularity of parallelism can be made arbitrarily fine, using processes as primitive units for parallelism. UNIX [36] takes a similar approach: the C language [20] has no built-in parallelism, but programs running with multiple processes can be constructed using the *fork* and *exec* system calls, and pipes for inter-process communication. A difference between our design and that of UNIX is that the UNIX kernel knows about different computations and allows pipes to be set up only among processes of the same computations, while in ours, a process can be created into any distributed computation and a process may migrate from one computation to another.

In fact, the runtime modularization in the cluster system is not directly supported by the kernel. The kernel has no concept of individual programs and computations; it sees and manages a flat world of processes. Through a layer of *policies* above the kernel, we create an abstract world of clusters on top of this flat world of processes, and introduce rules to control communications at runtime. Thus, the kernel creates a flat world of processes, and provides message passing primitives based on messages tagged with arbitrary, uninterpreted names (in fact, byte strings). These facilities are used by the policy-level library routines to provide name spaces, clusters, and cluster communications with visibility controls. This is described in more detail in Section 4.

3. Related Work

3.1. Multicast Groups

A *multicast* is the sending of a message to a number of recipients; the latter form a *multicast group*. The use of multicast can be found in such applications as two-phase commit protocols [17] and distributed elections [14]. Two approaches exist: a list of recipients is maintained and the multicast is performed as a sequence of unicasts to each of the recipients; or a group address is used and the multicast message is sent to it. The second approach requires that the underlying transport mechanism support some form of broadcast communication.

Multicast in our cluster model is supported naturally, by clustering and cluster communications. A cluster by itself can be a multicast group. The name of the cluster becomes the multicast address of the group. Such a group is dynamic since we allow processes to join a cluster as long as the visibility rules permit it. Additionally, we can create a static group by suitably restricting its visibility. Moreover, since a process can belong to multiple clusters, a process can be in several multicast groups.

Another form of multicast can be achieved through message tagging. (Tagging is known as *name-based addressing* in other contexts [1, 15].) A message with a particular tag may be received by a number of recipients who have issued a receive with the same tag. In fact, the cluster name group address can be considered a special case of a tag. Tagging may seem insecure, since any process may receive the message if it manages to issue the right tag. We avoid insecurity because a tag (and likewise the cluster name) is local to a particular cluster; only those processes in the cluster are allowed by the policy routines to use it. In Linda [16] however, where tagging is used, such a flexible structure of scopes for multicasting is not supported.

Compared to other existing multicast implementations, ours based on clusters appears to be more flexible. In the V Kernel [9] and extended UNIX [2], a multicast group must be formed and maintained explicitly in the kernel, multicast messages must use special group addresses, and joining a group requires a special procedure. These additional provisions are eliminated in the cluster system: a multicast group is a cluster, multicast messages use the same form of address as unicast, and joining a group (cluster) is already provided as part of the clustering facility. UNIX 4.2BSD [25] has an even more restricted form of process group, which is designed primarily for job control. In StarMod [24], communication occurs via ports. There is a special type of port called a multicast port, which can be imported by a number of recipient processes. Multicast groups are set up at compile time and within network modules which are essentially distributed programs. In contrast, clusters as multicast groups are defined dynamically at runtime, and there is no distinction between groups belonging entirely to a single distributed program, and groups that cross program boundaries.

3.2. Hierarchical Process Composition

HPC [24] can be viewed as an extension of the process abstraction. A set of communicating processes can be encapsulated to form an abstraction whose status within the system is equal to that of an ordinary process. The simplest type of object in HPC is the process. An object can be constructed from component parts by combining other previously created objects, *communication channels* between sub-objects, an *encapsulation shell*, and a set of *interfaces* to the external world.

An *operation domain* is an encapsulation shell created with a designated control component, called the *controller*, responsible for modifying and maintaining the internal structure of the objects within the domain.

Compared to HPC, the cluster model provides additional flexibility, since it allows multiple trees to be formed in one or more dimensions. In principle, by modifying the policy routines, arbitrary graphs could also be formed, although we have not yet experimented with this. A cluster corresponds to both an encapsulation shell and an operation domain. Instead of a distinguished controller, every member of a cluster can change the configuration of the cluster by creating new processes and clusters, destroying existing ones, or migrating to other clusters. Communication requires no separate creation of channels, but rather, uses arbitrary names at runtime (*i.e.*, the name of a cluster, a process, or an arbitrary tag).

One other feature of HPC is the ability to implement *shared objects*. A shared object is used when two objects, whose closest common ancestor is many levels above in the hierarchy, wish to communicate. This shared object must reside in two operation domains to which the two objects belong. Our solution to the same problem is provided by dimensions: objects (such as processes) which are not related in one dimension may appear in the same cluster or related clusters in other dimensions. For example, in one particular dimension designed particularly for client-server interactions, each server shares a cluster with all potential clients. In both cases, some perhaps contrived sequence of primitive operations must be issued to achieve the sharing.

3.3. Name Spaces

A global naming convention for distributed systems has been developed by Curtis *et al* in [10]. It provides a unified notation for accessing networks, tasks, modules, and data structures. For example, at runtime, a tree of tasks and processes that resembles the UNIX file tree is maintained: the "root directory" is a system task, the "home directories" are shell tasks, "directories" are user tasks, and the "files" are processes. A notation equivalent to that of the UNIX file system is used to refer to these tasks and processes. Although not mentioned in their paper, UNIX-like access control can readily be incorporated. The access control scheme employed by the cluster model is somewhat different. Instead of specifying at each node in the tree whether it is accessible (*e.g.*, by owner, group members, or others, as in UNIX), a cluster maintains a relationship only with its immediate neighbors (*i.e.*, its parent and its child clusters). To allow access by user, group, or others, other dimensions are used. Not addressed in their paper is interprocess communication and the possibility of process migration (whose counterpart in UNIX is presumably "mv").

Another naming scheme is the V Naming Model [8]. A *context prefix server* dictates in which context a given name should be interpreted. The name is then sent to the associated server. Each of these servers has its own internal structure which may be a tree or an arbitrary graph. Interpretation continues inside the server until the final object is located, or the name is found to require interpretation at yet another server. This scheme allows a unified way to access objects of various kinds. The differences in name formats for different objects are hidden inside individual servers.

3.4. Nested Transactions

A transaction is a transformation of state which has the properties of atomicity (all or nothing), durability (effects survive failures) and consistency (a correct transformation) [18]. The transaction concept has been key to the structuring of data management applications [21]. A natural extension is to make transactions recursive. Transactions which are composed of other transactions, or *nested transactions*, have been the subject of much current research [28, 29, 34, 35]. The concept of nested transactions can be traced back to the work of *spheres of control* by Davies [11], where we can see the abstraction notion of a different kind: what is hidden is the sequence of steps threading through a transaction, while the transaction appears as a single indivisible step to the user. A transaction can therefore be viewed as an abstract object of a special kind. It consists of actions (processes) to be performed on internal data that are to be committed at the end; the processing and the final committal are transparent to the user. Nested transactions are properly nested and therefore form a tree. In terms of hierarchy and modularization, clusters have a much more flexible structure than nested transactions, due to the ability to reconfigure at runtime. This is not surprising since the two have different focuses: the focus of the cluster model is on structuring, abstraction, and process interaction, while the focus of nested transactions is on atomicity, durability and consistency. Nevertheless, they are similar in their dynamic structure.

3.5. Protection

Protection requirements cause certain structures to be introduced into the operating system. These structures dictate whether accesses can be made by one object to another. Examples of such structures include access matrices and capability lists. A large body of literature exists on this subject (see the survey in Chapter 11 of Peterson and Silbershatz [31]), but relatively little has been said specifically about protection among processes at runtime. In Mininet [27], the use of *task graphs* for enforcing expected patterns of message transmission was proposed. However, in Mininet, there was no ability to change task graphs dynamically when processes were created or destroyed.

Process interaction includes communications and such operations as suspending, resuming, and terminating a process. There must exist some kind of protection scheme by which process interactions can be properly controlled. In some systems (*e.g.*, Shoshin [38]) a process normally belongs to a *process creation tree*. A process will be destroyed when any one of its ancestors terminates. Also, a process can specifically terminate another process by issuing a *kill* signal. In this case, the protection rule is based on user identities: only processes belonging to the same user can kill one another.

In the cluster model, protection is associated with cluster boundaries. Cluster boundaries may be opaque or transparent from the outside. Opaque boundaries deny access by processes outside of the boundaries. Likewise, whether processes inside a cluster can see outside is determined by the visibility (from inside out) of the boundary. Different dimensions of clustering add different dimensions to protection: two processes which cannot interact because of visibility rules in one dimension may be able to interact in another dimension. The flexibility of cluster structuring can therefore be extended to the flexibility of protection structuring. For example, the protection rings of Multics [30] could be formed using clusters and a particular set of policies; the

latter can be associated exclusively with the particular dimension in which these rings are implemented.

Another aspect of protection can be found in inter-process communications. IPC can be viewed as a kind of *bilateral operation*, in that cooperation of both sender and receiver is required for communication to occur. Therefore, a process is protected from others if it chooses not to communicate.

Other works that discuss process structuring include PCL [4] and Script [13]. Their proposals however have only an indirect relationship with our cluster model, since they operate at a level higher than that of the programming language: they attempt to produce a program in a particular programming language from a high level specification of some process structure.

4. Process and Cluster Naming

In what follows, we look at the question of naming, and present the naming scheme adopted for our design. Naming is important because it is fundamental; virtually every useful operation performed in a system, at any level, requires reference to some kind of name, be it a name for a memory location, a register, a device, a file, a process, or a machine. For a general discussion on naming and the fundamental principles involved, the reader is referred to the excellent work by Saltzer [37]. A system can be divided into many levels, and the subject of naming may be treated according to levels, since each level may raise different issues and pose different design constraints. Watson, in [39], addresses naming in the context of distributed systems consisting of multiple levels. For our purposes, however, we concentrate mainly on two levels, the process level and the cluster level. Each of these levels provides an execution environment for a particular kind of object. Processes are the objects (and the only objects) at the process level, and are implemented by the kernel; clusters are implemented by the policy routines from the kernel facilities. Since clusters are actually composed of processes, we have a two-level naming strategy for processes.

In the following subsections, we discuss process naming and cluster naming in somewhat more detail, including a discussion of how cluster names are bound. We discuss how the basic cluster naming is generalised to match the hierarchical structure of clusters, and then how it can be generalised further to bind more arbitrary objects to names. We conclude the section with a discussion of problems arising in connection with non-existent clusters, and with an evaluation of cluster-based naming.

4.1. Process Naming

Process naming is necessary for a number of practical reasons. Some are related to the implementation of the process itself; for example, the process might have to be entered into different structures (for the purpose of scheduling) at different times, and it is important that it be identified correctly when moving from one structure to another. Or, in the case of remote IPC, a reply to a request from a remote process must somehow identify the process which originated the request. In our design, we adopt the usual convention of referring to each process in the system with a unique process ID (PID). Briefly, a PID has the following general properties:

- it is bound to a process at instantiation time (late binding),
- it is unique (*i.e.*, not shared) across the entire system, and
- it is bound directly and exclusively to a single process.

These properties have implications that dictate how processes may be employed and manipulated for distributed programming. Moreover, considering some of these implications suggests how to design the second level of naming, cluster naming, to improve functionality and flexibility.

Late binding (at instantiation time) has a few advantages. It is economical to assign a PID only to a successfully-created process. Information about a process can be embedded in the PID, such as the location of the site at which the process exists. If the PID of a process is always returned from a call to the *create* kernel routine (which we assume is the only means of creating a process), then other processes can be prevented from referring to a process prematurely. This introduces a kind of synchronization which may be useful in distributed programming.

The PID is always bound to a process. This is appropriate for operations which are to be applied directly to a process, such as *suspend*, *resume*, *kill*, and *change_priority*. However, it lacks the generality required for other operations. For example, a process may sometimes want to send a message to any process rather than to a specific one. In this case, there is no simple substitute for a PID. The solution offered by Shoshin [7, 38] is the pseudo-PID *ANY* which may be used to replace a normal PID when calling an IPC primitive. We have found that it is better to have simpler IPC primitives at the process level, while providing flexibility and power at the cluster level. That is, PIDs at the process level continue to refer exclusively to processes, and are no longer visible at the cluster level. In particular, as we discuss below, a name at the cluster level can be bound to virtually anything, be it a cluster, a channel for communication, or an attribute of an application.

Our scheme for cluster naming incorporates ideas discussed by various researchers. A discussion and evaluation of these naming systems can be found in [22]. We address some of these requirements in the context of cluster naming, at the end of this section. In the following, we describe how cluster naming can be used to achieve highly flexible (cluster-oriented) system operations.

4.2. Cluster Naming

Apart from its unique PID, each process has associated with it at least one cluster name (actually stored by the policy routines in the process' "policy data segment"). This name identifies the cluster to which the process belongs, and in general, is associated with more than one process. However, processes within the same cluster only "name" each other in the sense of agreeing on message tags specified in IPC operations. Like kernel operations that can be invoked and applied on PIDs (and hence their corresponding processes), there are cluster operations that take cluster names as parameters. But unlike process operations, cluster operations may affect more than one process or cluster.

4.3. Binding of Cluster Names

Cluster names are bound at programming time by the user (very early binding). They are arbitrary character strings. One rationale behind this choice of binding time is to eliminate the *creation bottleneck* [13]. This bottleneck is evident in the way the *create* call returns the PID of a newly spawned process. In a distributed program consisting of multiple processes, creation is the only means of obtaining the PID of a newly-spawned process. That is, a process must be the creator (caller of *create*) of a second process in order to obtain its PID directly. It may then propagate this PID to other processes in order to enable communication between these other processes and the newly spawned process. Although PIDs are assumed to identify communication channels here, the same applies to other kinds of channel and names, such as capabilities for dynamically-created ports (*e.g.*, Accent [33]).

The problem of the creation bottleneck does not exist if the PID is assigned by the user. We retain the usual semantics of PIDs (for the reasons listed above) while making use of the second level of naming, cluster naming, to implement user-assigned names. We call this method of naming clusters *local naming*, since the user chooses the names for the clusters in his own local context. By default, all distributed programs (consisting of multiple clusters) written by the same programmer are fully-connected graphs and communication takes place directly in the cluster name space, without having to set up communication channels first.

Even with local naming, the need to propagate a name from one cluster to another still exists. This is the case when programs written by different users need to communicate. Such propagation may however be done at programming time and does not necessarily introduce message overhead at runtime. Servers, or programs that are of interest to many users, may "publish" the names they use instead of propagating them on an individual basis.

4.4. Name Conflicts and Contexts

The use of arbitrary local names creates an obvious problem—conflicts of names chosen independently by different naming agents such as users and programmers. This does not happen with PIDs at the process level, where there is a hierarchy of name components, managed collectively by the kernels at the various sites. The local ID can be considered a local name taken from a local name space, and the site ID a name from some network name space. Although LIDs may conflict, the combination of local and site IDs is unique. To resolve conflicts at the cluster level, we therefore adopt a similar but more general method of hierarchical naming. A tree structure is appropriate since it is the kind of structure into which a complex programming system tends to evolve. Such a system relies on the idea of data abstraction for program representation, construction, execution, and interaction.

A simple cluster tree is shown in Figure 1. Each cluster also defines a (local) name space; within it are subspaces represented in the figure by the child clusters of this cluster. Cluster A is the root cluster; it has two child clusters and hence two subspaces B and C. Note that although the local name C appears again further down, the cluster it refers to is distinct from the cluster referred to by the first C. In fact, each cluster is uniquely identified by its *absolute name* which is the label just outside the square. (The label within the square is the cluster's local or relative name.) To identify a cluster in the tree, either absolute or relative naming may be used. The policy module can transform relative names into their absolute form based on the location (in the

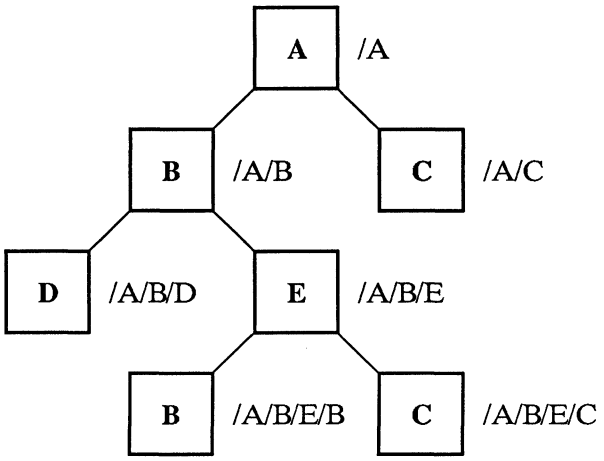

Figure 1 – A Simple Cluster Tree

tree) of the cluster that specifies the name. Syntactically, an absolute name begins with a / such as /A/B/D, while a local name begins with a name component such as B/D.

The tree structure provides the user with the ability to create naming contexts, which is are a standard feature of most naming schemes. A context, which is a local name space associated with a cluster, may be considered as the execution environment of a distributed computation, or it may be considered as a catalog for objects such as files or programs. As an execution environment, the names in the local name space correspond to tags that are used for messages passed between clusters; as a catalog, these names correspond to files, programs, or other types of object, and are necessary for making a reference to and performing operation on these objects. Both uses are possible simultaneously.

4.5. Generalizing Name to Object Binding

Until now, we have considered names to be bound to clusters. We now propose generalizing the binding, so that names can refer to arbitrary objects rather than clusters alone. This is possible with the introduction of the idea of rendezvous. The detail of rendezvous and its implementation, the rendezvous store, are touched on briefly in Section 5, while fuller information can be found in [23]. Here, we present the basic rationale behind such a mechanism and its implications for naming.

All operations provided by the rendezous store are in terms of tagged IPC. For normal Send and Receive operations, a tagged message is deposited into the store via an S-request, and an R-request containing a matching tag retrieves the message. Various styles of IPC may be built at higher levels, based on this simple mechanism for exchanging messages. Since all operations are in terms of tagged IPC and must go through the rendezvous store, clusters names and message tags become location-independent; that is, all name-to-location mappings are hidden in the rendezvous store, a prerequisite for dynamic reconfiguration.

Briefly, a rendezvous is said to occur when two rendezvous requests, each carrying a *tag* or a *rendezvous key*, match. The match may be based on the keys alone or on additional attributes as well. One or more responses which lead to certain actions in the system are then generated from the occurrence of the rendezvous. Figure 2 shows the series of steps involved.

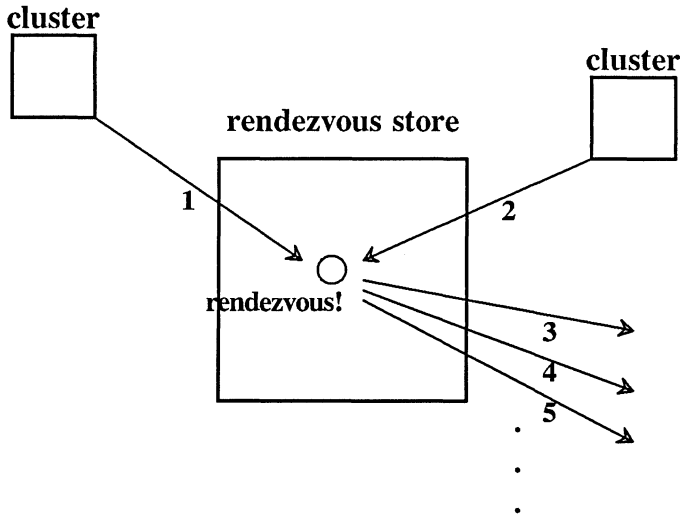

Figure 2 — A Rendezvous

Based on this simple mechanism, a variety of actions to be performed on a variety of objects may be implemented, each taking the form of a pending rendezvous request in the rendezvous store.

Cluster operations are classified into unilateral and bilateral operations. A unilateral operation is applied to an object (*e.g.*, *kill*), while bilateral operations consist of the exchange of a message between two or more objects. In this system, we transform all unilateral operations into bilateral operations implemented through tagged IPC. Tagged IPC allows much flexibility since a tag may be bound dynamically to one or more real or imaginary objects. Both types of operation can be implemented by the rendezvous store, as depicted in Figure 3. In the case of unilateral operations, the third party will generally be trustworthy software such as the kernel or policy routines.

If we treat each square in the tree structure of Figure 1 as merely a local name space, then clusters (as well as processes) can be considered nameless. When a cluster is first created, it exists as a set of processes scattered in one or more local name spaces in the tree. However, to qualify as a cluster, these processes must have something in common: a local name for termination (the "kill key"). This local name is entered into the rendezvous store in the form of a rendezvous request at creation time. When termination of this cluster is requested, a matching request is sent and the resulting rendezvous causes all the processes (and hence the cluster) to terminate. Although this special key of the cluster seems equivalent to a name for the cluster, it is possible to have a cluster which does not have such a key. Hence, there is no need to identify a cluster or a process during normal operation of the system. Each cluster is created for the function it provides; the function may be invoked through the particular protocol associated with it.

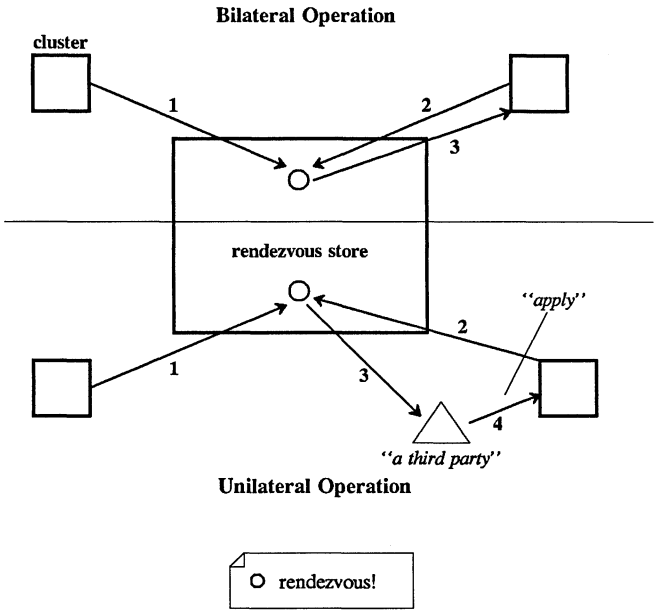

Figure 3 — Unilateral and Bilateral Operations

A protocol in this case is simply a specification of the series of rendezvous requests which a user of the function must submit. A cluster may terminate by itself or due to a *kill* operation from another cluster, as described above. In either case, no explicit name for the cluster is necessary.

Currently, we limit ourselves to the case of each cluster mapping to exactly one local name space and vice versa. The reason for this is ease of illustration; also, such mapping is sufficient for most of the applications we consider. As such, and for the sake of convenience, we consider the name of a local name space to be the name of the corresponding cluster. However, the view above of nameless clusters is a valid and more precise one.

4.6. Non-Existent Clusters

Since local names are known in advance of object creation, there is a danger of referring to a non-existent cluster. A cluster may not exist because it has not yet been created, its creation operation failed, or it has terminated. In order that a cluster not wait indefinitely for a non-existent cluster to respond, the system must be able to inform the former cluster of the above facts. The last two cases may be dealt with by leaving a "trace" behind, indicating that the cluster creation failed, or that the cluster terminated. The system would keep track of all these traces and respond to those objects wanting to contact non-existent clusters. One obvious problem arises: When should these traces be deleted? If a trace is for a cluster which is part of a distributed program, then perhaps when the program exits, the trace can be deleted. This requires that the program have a defined "end" which, when reached, would signal the system to begin garbage collection for this program. If a trace maps to the whole program (*i.e.*, one-cluster program) or if it is not tied to any program at all, then some kind of timeout may be set for this trace; deletion would then follow the expiry of this timeout. Such a timeout may be hard to set in practice, but in general, we would expect the timeout to be set to infinity and the trace to be

deleted when the non-existent cluster is created or recreated. For the first case above (the cluster has not yet been created), the only reasonable solution would be for the user who is referencing the non-existent cluster to use a timeout parameter when a call is issued. In many environments, the use of timeout may well be a practical alternative to the use of traces.

4.7. Evaluation of Cluster Naming

The idea of rendezvous as the means for interaction among clusters leads to several properties of our naming scheme. These properties satisfy requirements proposed in [22].

First of all, using the rendezvous store, services can be decoupled from their implementation. The protocol for requesting a certain service is represented by a series of rendezvous requests in the rendezvous store, but the mapping from these requests to actual clusters implementing the service is hidden. All that is required of the client is to issue rendezvous requests according to published instructions. Similarly, the rendezvous store may be used to suppress the details of object types. This leads to protocols for dealing with generic services or objects.

It is simple to introduce new types of objects, unrelated to clusters. For example, in the policy module, we may attach a special prefix to the local names used for these new objects, in order to distinguish them from clusters or other objects. Manipulation of these objects (such as communication and termination), however, can remain the same as for clusters — the rendezvous store can simply treat the special prefix as ordinary characters. For example, a distributed disk block service can be implemented which provides the clients with access to disk blocks on local and remote disks. The server hides the physical locations of the disks and creates an image of a single, large pool of disk blocks identified by logical disk block numbers. To make use of the rendezvous store, a message containing a request for the disk block service would carry a tag with a special prefix. When a rendezvous occurs, the request will then be sent to a disk block server. The prefix serves to identify the domain for which the message is targeted. As a result, we have two domains for two types of objects: processes and disk blocks. The prefix (an object type identifier) of an incoming message dictates which path to a domain is to be taken. Alternatively, the disk block service can be implemented by ordinary processes, and as such, no special prefix is necessary and the disk block service exists at the application level. The difference is essentially whether the disk block number appears in the tag or the data of the message.

Attribute-oriented naming is possible and straightforward to implement using the rendezvous store. An attribute is coded as just another local name which is known to a number of clusters wishing to communicate using attributes without exposing their identities. An example of this is type matching as in Linda [3], which involves ensuring that the types of send and receive arguments match. The type in our case would simply be a subfield in the tag of a message.

With our scheme, any action may be associated with rendezvous. An association between a particular rendezvous and a particular action may be established by the system administrator, the user, or the joint effort of both. With such flexibility, extended protection, monitoring, and aliases can be implemented easily.

Finally, the rendezvous store can be distributed easily, since the individual requests held in it (awaiting rendezvous) are independent of one another. Adding a new user site to the network requires no change to the rendezvous store. Withdrawal or failure of a user site similarly would

not have any impact on the rendezvous store and the naming service it provides. In order to make use of the rendezvous store, a user host need only equip itself with a simple module for executing a protocol to access the rendezvous store. The simplest of such modules might consist of a hash table (from keys to rendezvous hosts in case the rendezvous store is distributed) and an interface to the communication module. The policy module is not part of the rendezvous mechanism and does not necessarily have to be provided, if the site only requires access to the rendezvous store.

5. Implementation of a Prototype

We have implemented a prototype of the cluster system as a client system on Shoshin [7]. One of the hosts is dedicated to the operations of the rendezvous store, which is interfaced to all other (user) hosts via an IPC server (a light-weight process called a *semi-process*) in the kernel of each user host. On top of the IPC server are one or more transport clients (corresponding perhaps to different dimensions) which implement the buffering and synchronization facilities required by the policy module at the next higher level. The policy module mediates between user calls and the kernel. It is implemented as a *protected shared library*, similar to the kernel interface subsystem in the UCLA secure kernel [32], but its routines are call-only. It is shared so that only one copy exists in memory, it runs in the address space of the user, and may access user data. This in effect transforms the normal process into an extended one with a policy code segment and a policy data segment. With a single-site implementation of the rendezvous store, the performance is encouraging: a message transaction (with moderate load) originated at the user level by a cluster takes approximately 5 msec. to complete. A re-implementation of the rendezvous store in a distributed fashion is one of our major future projects.

Above the rendezvous store, we have implemented policy routines which intercept cluster *send* and *receive* operations, and transform the local or relative names in them into absolute cluster pathnames, by making use of the absolute cluster names stored in the policy data segment for each process. These routines also implement visibility controls, by checking that the pathnames issued by a cluster as targets for IPC operations agree with the visibility constraints encoded in the absolute cluster pathnames. Thus, one can view these policy routines largely as trusted software whose responsibility is the guaranteed vetting and transformation of character strings for intra- and inter-cluster communication into otherwise uninterpreted message tags used by the rendezvous store.

6. Conclusion

We have developed a cluster model, whereby clusters replace processes as the primitive objects on which user programming is based. A cluster may contain zero or many clusters or processes, each of which represents one or more separate threads of control. On one hand, the model facilitates the writing of programs in the style of multi-process structuring; on the other hand, clusters are well-suited for distributed systems where multiple distributed resources (such as processors) are to be exploited.

Clusters are organized into a hierarchy at runtime, and hierarchical names are used to identify them. Hierarchical naming allows name components to be assigned by the users. Therefore, clusters can readily refer to one another at runtime without first establishing a channel or

publishing a cluster identifier. The typical concept of a program becomes superfluous here. A cluster (not a program or a process) is a unit of action and it may interact (as client, server, or both) with any other cluster in the system, as long as the visibility rules allow it. Visibility is imposed on each edge in the hierarchy to control the flow of information. This controlled flow of information constitutes base-level protection which is used to implement abstraction (information hiding) and cataloging services (information disclosure). Higher levels of protection (such as those based on user identifiers) can be implemented by policy routines or the clusters themselves.

Clusters and cluster operations are implemented by a layer of policy routines on top of the kernel. Clusters and their operations are not known to the kernel, which only implements and supports processes. Clusters have no explicit data structures, and the components of a cluster may be freely distributed. The result is a highly modular implementation where clusters can be migrated easily, and the system as a whole can be made more fault tolerant.

In summary, a new programming methodology based on the cluster model has been proposed: we have also fully exploited the principle of policy/mechanism separation [26] by implementing an entire user system using policy routines alone, and by providing a simple, flexible, and powerful mechanism called the rendezvous store to support the execution of these policy routines. All these have been integrated in a coherent manner in the implementation of a prototype cluster system as described in the previous section.

At the present, we are hoping to continue our investigation of the possibilities for software development offered by the cluster system and rendezvous store. We have shown their feasibility, but have yet to construct a significant enough body of runnning software to demonstrate that they do indeed help to solve the software problem. In particular, we need to investigate the use of visibility constraints as a primary means of protection, and better understand their interaction with dimensions (where processes belong to a number of unrelated clusters). On a more prosaic level, we are hoping to build a more distributed prototype of the rendezvous store, perhaps involving some form of hardware support.

7. References

1. M. Ahamad and A. J. Bernstein, An Application of Name Based Addressing to Low Level Deistributed Algorithms, *IEEE Trans. on Software Engineering* **SE-11**(1) pp. 59-67 (January 1985).

2. M. Ahamad and A. J. Bernstein, Multicast Communication in UNIX 4.2BSD, *Proc. 5th Int. Conf. on Distributed Computing Systems*, pp. 80-87 (May 13-17 1985).

3. S. Ahuja *et al*, Linda and Friends, *IEEE Computer* **19**(8) pp. 26-34 (August 1986).

4. V. P. Lesser *et al*, PCL: A Process-Oriented Job Control Language, *Proc. 1st Int. Conf. on Distributed Computing Systems*, pp. 315-329 (October 1979).

5. G. R. Andrews, Synchronizing Resources, *ACM Trans. on Prog. Lang. and Syst.* **3**(4) pp. 405-430 (October 1981).

6. M. R. Barbacci *et al*, The Software Engineering Institute: Bridging Practice and Potential, *IEEE Software* **2**(6) pp. 4-21 (November 1985).

7. J. P. Black, W. H. Cheung, E. C. Lam, F. C. M. Lau, and E. G. Manning, Shoshin: Developing and understanding distributed system software, UW/ICR report 87-04, 20pp in ms. (May 1987). Submitted for publication to *IEEE Trans. on Software Engineering*.

8. D.R. Cheriton and T. P. Mann, Uniform Access to Distributed Name Interpretation in the V-System, *Proc. 4th Int. Conf. on Distributed Computing Systems*, pp. 290-297 (May 14-18, 1984).

9. D. R. Cheriton and W. Zwaenepoel, Distributed Process Groups in the V Kernel, *ACM Trans. on Computer Systems* **3**(2) pp. 77-107 (May 1985).

10. R. Curtis and L. Wittie, Naming in Distributed Language Systems, *Proc. 4th Int. Conf. on Distributed Computing Systems*, (May 14-18, 1984).

11. C. T. Davies, Data Processing Spheres of Control, *IBM Systems Journal* **17**(2) pp. 179-198 (1978).

12. Department of Defense, *Reference Manual for the Ada Programming Language,* Superintendent of Documents, U.S. Goverment Printing Office, Washington, D.C. (1980).

13. N. Francez and B. Hailpern, Script: A Communication Abstraction Mechanism, *Operating Systems Review* **19**(2) pp. 53-67 (April 1985).

14. H. Garcia-Molina, Elections in a Distributed Computing System, *IEEE Trans. on Software Engineering* **C-31**(1) pp. 48-59 (January 1982).

15. D. Gelernter, Dynamic Global Name Spaces on Network Computers, *Proc. Int. Conf. on Parallel Proc. 84*, pp. 25-31 (August 1984).

16. D. Gelernter, Generative Communication in Linda, *ACM Trans. on Prog. Lang. and Syst.* **7**(1) pp. 80-112 (January 1985).

17. J. N. Gray, Notes on Database Operating Systems, pp. 393-481 in *Operating Systems: An Advanced Course*, Springer-Verlag (1978).

18. J. N. Gray, The Transaction Concept: Virtues and Limitations, *Proc. 7th Conf. on Very Large Data Bases*, pp. 144-154 (September 1981).

19. C. A. R. Hoare, Communicating Sequential Processes, *Commun. of the ACM* **21**(8) pp. 666-677 (August 1978).

20. B. W. Kernighan and D. M. Ritchie, *The C Programming Language,* Prentice-Hall (1978).

21. B. W. Lampson, Atomic Transactions, pp. 246-265 in *Distributed Systems - Architecture and Implementation*, ed. D. W. Davies *et al*, Spring-Verlag (1981).

22. K. A. Lantz *et al*, Towards a Universal Directory Service, *Operating Systems Review* **20**(2) pp. 43-53 (April 1986).

23. F. C. M. Lau, Policies and Mechanisms for Distributed Clusters, Ph.D. thesis, Dept. of Computer Science, University of Waterloo (1986).

24. T. J. Leblanc and R. P. Cook, High-Level Broadcast Communication for Local Area Networks, *IEEE Software* **2**(3) pp. 40-48 (May 1985).

25. S. J. Leffler *et al*, *UNIX Programmer's Manual (4.2BSD),* Dept. EECS, UC Berkeley (August 1983).

26. R. Levin *et al*, Policy/Mechanism Separation in Hydra, *Proc. 5th Symp. on Operating Systems Principles*, pp. 132-140 (November 1975).

27. J. Livesey and E. G. Manning, Protection and Synchronization in a Message-Based System, *Computer Networks* **7** pp. 253-268 North-Holland, (1983).

28. J. E. B. Moss, Nested Transactions and Reliable Distributed Computing, *Proc. Symp. on Reliability in Dist. Soft. and Database Systems*, pp. 33-39 (July 1982).

29. E. T. Mueller *et al*, A Nested Transaction Mechanism for LOCUS, *Proc. 9th Symp. on Operating Systems Principles*, pp. 71-89 (October 1983).

30. E. I. Organick, *The Multics System: An Examination of its Structure,* MIT Press (1972).

31. J. Peterson and A. Silberschatz, *Operating System Principles*, Addison-Wesley, (1983).

32. G. J. Popek *et al*, UCLA Secure UNIX, *Proc. NCC*, pp. 355-364 AFIPS, (June 1979).

33. R. F. Rashid and G. G. Robertson, Accent: A Communication Oriented Network Operating System Kernel, *Proc. 8th Symp. on Operating Systems Principles*, pp. 64-75 (December 1981).

34. D. P. Reed, Implementing Atomic Actions on Decentralized Data, *ACM Trans. on Computer Systems* **1**(1) pp. 3-23 (February 1983).

35. D.R. Ries and G. C. Smith, Nested Transactions in Distributed Systems, *IEEE Trans. on Software Engineering* **SE-8**(3) pp. 167-172 (May 1982).

36. D. M. Ritchie and K. Thompson, The UNIX Time-Sharing System, *Commun. of the ACM* **17**(7) pp. 365-375 (July 1974).

37. J. H. Saltzer, Naming and Binding of Objects, pp. 99-208 in *Operating Systems: An Advanced Course*, Springer-Verlag (1979).

38. H. Tokuda, *Shoshin: A Software Distributed Testbed,* Ph.D. Thesis, CS Dept., Univ. of Waterloo (1984).

39. R. W. Watson, Identifiers (Naming) in Distributed Systems, pp. 191-210 in *Distributed Systems - Architecture and Implementation: An Advanced Course*, Springer-Verlag (1981).

40. N. Wirth, *Programming in Modula-2,* Springer-Verlag (1982).

Acknowledgements

This work was supported in part by the Natural Sciences and Engineering Research Council of Canada, by Digital Equipment Corporation, and by a Commonwealth Scholarship.