Characterizing Distributed Computing Paradigms†

Michael F. Schwartz
Isabelle M. Demeure

CU-CS-422-89 August 1989

Department of Computer Science
Campus Box 430
University of Colorado
Boulder, Colorado 80309-0430
(303) 492-7514

Abstract

In this paper we analyze two popular, markedly different distributed computing *paradigms*: the client/server paradigm, and the role symmetric paradigm. We define a simple symmetry condition that partitions the space of distributed computations into two parts, and use this condition to prove that given any distributed computation one can construct a functionally equivalent computation that uses one of these paradigms. We then analyze the basis of this symmetry condition in distributed computations, and show how various practical problems in distributed computing can be viewed in terms of composing subcomputations in different ways according to the symmetry condition. We then explore the structural properties of the two paradigms, considering in particular the strengths each of the paradigms presents to the implementor of a distributed computation. This characterization is important because a programmer developing a distributed computation can be aware of the properties each paradigm possesses in solving a specific problem, and thereby more clearly design each piece of the computation. We illustrate the concepts by examining a diverse collection of existing systems and computations.

†This material is based upon work supported in part by NSF cooperative agreement DCR-8420944, and by a grant from AT&T Bell Laboratories.

1. Introduction

The issues spanned by the field of distributed computing have evolved considerably in the past ten years. During the early to mid-1980's, a great deal of attention was focused on supporting the abstraction of an integrated computing environment built on top of a distributed substrate (e.g., see [Almes et al. 1985, Walker et al. 1983]). The rapid increase in the number and power of available processors, particularly in the form of practical parallel computers, has shifted some of this focus to work in parallel computing systems (e.g., see [Martin, Bergan & Russ 1987]). Additionally, the use of interorganizational, large scale computer networks has called increasing attention to issues such as security [Steiner, Neuman & Schiller 1988] and heterogeneity [Notkin et al. 1987].

As the field has grown to encompass an increasingly diverse set of issues, there have been many efforts to characterize particular problems and techniques that can be applied to them (e.g., see [Chandy & Misra 1986, Saltzer 1982, Spector 1982, Wang & Morris 1985]). Rather than exploring a specific problem, in this paper we focus on the basic building blocks used for dealing with distribution, which we call *paradigms*. This notion concerns an abstract formulation for organizing and carrying out a computation, based on the principal goals and constraints placed on the computation. For example, in the realm of sequential computations, divide and conquer is a commonly used paradigm.

There have been few attempts to examine computations at the level of paradigms. A notable exception is Nelson and Snyder's work on programming paradigms for use in parallel scientific computing applications for distributed memory multiprocessors [Nelson & Snyder 1987]. In that paper, the authors described and illustrated several "recognized to be useful" paradigms that could be considered part of a programmer's tool kit for developing parallel applications for distributed memory multiprocessors. In contrast, we focus on the abstract nature of paradigms for dealing with the distribution of resources and control in a network, using existing computations as examples of the various properties we discuss. While a general characterization of distribution would also consider various issues associated with potential parallelism (such as concurrency control and speedup), these issues are outside the scope of this work.

Two paradigms have become particularly popular in practical distributed systems as well as theoretical studies of distributed computing: the client/server paradigm (used by many file systems and name services), and the role symmetric paradigm (used by various voting and Byzantine agreement protocols). Yet, in an important sense these paradigms are not well understood, since there has been no characterization of when and how each paradigm can be applied in implementing distributed computations. In this paper we give examples of, and formally define, these paradigms. We then characterize the classes of distributed computations that can be supported by each paradigm, and explore the structural properties of each paradigm. We focus particular attention on the strengths each of the paradigms presents to the implementor of a distributed computation. We also discuss how one can compose these paradigms in various ways to construct practical distributed computations.

Throughout this paper, we use the term *distributed computing* to refer to the general discipline of computing on a network of machines. We use the term *distributed computation* to refer to a particular problem in distributed computing, such as the Byzantine Generals Problem [Lamport, Shostak & Pease 1982]. We use the term *distributed system* to refer to a collection of distributed computations assembled to handle a broad range of problems. We consider a distributed computation to be composed of a collection of *processes* executing on processors. While the issues we discuss are not specific to this particular abstraction, we use this term for the sake of concreteness. In place of "process" one could substitute any abstraction relating to the unit of scheduling and distribution in a computation, such as thread, object, or coroutine. We use the term *resource* to refer to any hardware, software, or information that might be shared among several processes, such as a CPU, a peripheral device, a record of a file, or a database lock. We use the term "user" to refer to a human, while "client" refers to a piece of software that runs on behalf of a user, making requests of a server in the client/server paradigm.

The remainder of this paper is organized as follows. In Section 2 we describe and illustrate the two paradigms. In Section 3 we introduce the symmetry condition that partitions the space of distributed computations, and use this condition to characterize the computations that can be implemented by each of the paradigms. In Section 4 we explore the basis of this type of symmetry in distributed computations, and show how various practical problems in distribution (such as support for replication, and interorganizational administration) can be viewed in terms of composing subcomputations in different ways according to the symmetry condition. In Section 5 we consider the structural properties of the two paradigms, analyzing what strengths each of the

paradigms presents to the implementor of a distributed computation. Finally, we offer some conclusions in Section 6.

2. The Paradigms

In this section, we describe and illustrate the two distributed computing paradigms under consideration. We also briefly mention some computations that are composed of multiple subcomputations, each of which uses one of these paradigms. This section is intended to give the reader an intuitive grasp on the paradigms, and to hint at our approach to partitioning the computation space. We give formal definitions and analysis of the paradigms in Section 3.

The Client/Server Paradigm

It is difficult to trace the entry of the client/server paradigm into the realm of distributed computing. In its various guises, it is probably the most widely applied distributed computing paradigm. Using this paradigm, some collection of resources is managed by one or more servers, whose collective role is to encapsulate technical and administrative details of the resource. By doing so, servers can present clients with a simple abstraction of the resource, namely, service request.

This paradigm is quite general. The encapsulated resource could be a physical device, such as a display or a workstation. In such a case, the server can "export" operations at a variety of levels. For example, in the X window system, operations are exported for creating and manipulating windows on a display [Scheifler & Gettys 1986]. In the Xerox Process Server system, operations are exported for assigning a (slice of a) workstation's CPU to a process for the duration of the computation [Hagmann 1986].

The encapsulated resource could also be some type of information, such as the information accessible from an information retrieval service [Salton 1986], a user registration service [Harrenstien, Stahl & Feinler 1985], or an object-oriented database [Hudson & King 1988]. In such a case, processes in the client role are typically provided with a very restricted interface, such as read-only access to data in a simple textual format, or a collection of object invocation primitives.

In the computations discussed above, a single server encapsulated various details of the relevant resource or resources. This approach is not always feasible, either because the environment may be physically too large for a single server to administer all of the resources (creating a bottleneck or critical point of failure), or because the environment may span several independent administrative domains, each of which wishes to administer its own resources. In this case, the server can be decomposed into a hierarchy of processes corresponding to the resource distribution. In this arrangement, servers can pass requests among each other to fulfill service requests.

As an example system that uses this "distributed client/server paradigm", the Domain Naming System allows name mapping information to reside in a distributed collection of servers, so that no server need store all mappings, and each organization can administer their name mappings independently [Mockapetris & Dunlap 1988]. To allow this, each server maintains some mappings, and is capable of finding other servers that can continue the name mapping process for other names, either by mapping the names directly themselves, or by further forwarding the query. Other name services carry the service distribution further, allowing various syntactic [Cheriton & Mann 1988] or semantic [Schwartz, Zahorjan & Notkin 1987] transformations at each server before passing the request to another server. These transformations support various degrees of heterogeneity.

In the Domain Naming System, control is distributed among the servers. The servers collectively support the abstraction that any server can answer any request, even though the data for a particular request may not reside at the contacted server. Some systems that use the distributed client/server paradigm support a more primitive abstraction, wherein clients are explicitly aware of the distributed nature of the data, and must locate and contact the appropriate server for any particular request. For example, in Sun Microsystems' Port Map mechanism, each Port Map server is capable of servicing {Program Number} $\rightarrow$ {Port Number} requests only for Sun RPC servers running on the same machine as that Port Map process [Sun Microsystems 1985]. It is the client's responsibility to find the machine on which the needed Sun RPC server and its Port Map server are running, through its own means (e.g., by reading a local file containing the host name on which the server resides).

Many distributed file systems use the distributed client/server paradigm to provide clients with the abstraction of a single global file system shared transparently across a network [Howard et al. 1988, Nelson, Welch &

Ousterhout 1988, Sandberg et al. 1985]. Some authentication services also use the distributed client/server paradigm in large scale environments where no single server can be trusted to authenticate all service requests. For example, Birrell et al. have developed an authentication service that allows servers to authenticate client requests without trusting all servers along the path through the hierarchy of authentication servers [Birrell et al. 1986].

Finally, the client/server paradigm has also become popular for use in various traditionally centralized computations. For example, many of the functions of traditional centralized operating systems have been replaced by network services in modern operating systems, including page servers and file servers [Accetta et al. 1987, Tanenbaum 1987]. These servers need not necessarily be on separate machines from their clients. Rather, the client/server paradigm is often used in these cases to simplify the structure of the operating system "kernel".

The Role Symmetric Paradigm

In some computations it is not possible or not desirable to give distinct client and server roles to constituent processes. Such computations utilize the role symmetric paradigm.

The role symmetric paradigm has been used in a variety of computations. Because all participants are equally involved, such computations often involve the use of broadcast. For example, the ARPANET routing algorithm uses a broadcast-based role symmetric paradigm to disseminate link status information in a fault tolerant fashion [McQuillan, Richer & Rosen 1980]. Since broadcast is expensive or unreliable in many settings, some algorithms using the role symmetric paradigm have been developed for disseminating information reliably without broadcasting (e.g., see [Alon, Barak & Manber 1987]).

Byzantine agreement protocols are another, more robust class of role symmetric paradigms. They can support distributed cooperation given relatively few assumptions about the trustability of nodes. (Each of the role symmetric computations discussed above requires that each node either execute correctly or not at all.) Byzantine agreement protocols can reach agreement at non-faulty nodes given certain basic assumptions about the environment, such as that the proportion of nodes that are not faulty is less than some fixed amount [Lamport, Shostak & Pease 1982].

Role symmetric paradigms have also been used for eliminating critical nodes in replicated databases through the use of weighted voting protocols [Gifford 1979]. Given a distributed system that suffered network partition, role symmetric paradigms have been used to deal with inconsistencies that may have been introduced into the system during the partition [Mueller, Moore & Popek 1983].

The computations discussed so far are all deterministic. There are also many probabilistic agreement algorithms, typically used in situations requiring looser coupling among computing processes. For example, the Ethernet binary exponential backoff algorithm allows nodes to arbitrate for access to the network without any centralized controller, and without knowledge of how many other nodes are on the network [Metcalfe & Boggs 1976]. Probabilistic agreement protocols have also been used to improve the adaptability of various systems for classifying objects based on an evolving consensus. Gordon describes a system in which document description categories evolve over time according to algorithms that resemble genetic mutation processes [Gordon 1988]. Schwartz uses probabilistic algorithms to organize and search a non-hierarchical space of resources, to allow users to discover the existence of resources of interest, according to classification schemes evolved according to the types of resources that exist and the types of queries users make [Schwartz 1989].

Multi-Paradigm Computations

Some distributed computations utilize more than one paradigm. For example, Chang and Maxemchuk defined a family of reliable broadcast protocols given by a resiliency parameter that trades off message count efficiency per broadcast for resiliency against processor failures [Chang & Maxemchuk 1984]. The protocols depend on a *token site* that monitors and saves all broadcasts, retransmitting them upon request to sites that missed the original transmissions. The token site is essentially a broadcast storage server, and hence the computation utilizes the client/server paradigm. The server migrates around the network to reduce past message storage requirements and avoid single points of failure. If the token site fails (or when a new computer is added to the network), the system enters into a *reformation phase*, in which the non-failed sites elect a new token site and agree on a new token site list. This election utilizes the role symmetric paradigm.

As a second example, many network clock synchronization protocols utilize the client/server paradigm to support a network time server, plus the role symmetric paradigm to elect a new server in case of server failure [Gusella & Zatti 1986]. Common to both of these examples is the use of the client/server paradigm for the main focus of the computation, with occasional use of the role symmetric paradigm to hold an election concerning process roles, to reduce storage requirements or increase robustness.

3. Partitioning the Computation Space

In this section we develop a simple symmetry condition that partitions the space of distributed computations into two parts. We then formally define the two paradigms described in Section 2, and prove that given any distributed computation one can construct a functionally equivalent computation that uses one of these paradigms.

To develop the symmetry condition, we begin with the notion of *role equivalence*. Intuitively, two processes are role equivalent if they service the same set of interfaces and support equivalent functionality. However, a formal definition cannot refer to the particular instances of software being executed, because processes implemented with different software could be role equivalent. For example, there could be different implementations of a standard communication protocol (such as TCP/IP) corresponding to each of the operating systems on which the protocol runs.

Instead, we begin by abstracting away all process state (such as variables and environment flags), input, and output into messages sent and received by the process. For example, reading program variables corresponds to receiving messages, and modifying program variables corresponds to sending messages. This abstraction is similar to the standard Turing machine abstraction of a single tape used for all instructions, data, and I/O. Abstracting process state into messages allows us to treat all computations as functional expressions, avoiding the complexities associated with computational side effects, which are not relevant to the analysis.

Given this abstraction, we introduce the following definition:

DEFINITION: *Two processes are* role equivalent *with respect to input message set* **S** *if given any sequence of input messages in* **S**, *both processes produce the same sequence of output messages.*

We now define the key property of this section:

DEFINITION: *Given a computation* C *consisting of processes* $P_1, P_2, ..., P_n$, *we say that* C *is* role symmetric *if, given the set* **S** *of all input messages that any of the* P_i *could possibly receive, the processes are pairwise role equivalent with respect to* **S**. *Otherwise, the computation is* role asymmetric.

We can use this condition to partition the computation space into two parts that we will now show correspond to the two paradigms described in Section 2. We begin by formally defining the paradigms:

DEFINITION: *A computation uses the* client/server paradigm *if it is possible to partition its constituent processes into two groups, labeled "clients" and "servers", such that if any given client and server communicate, the client always initiates contact with the server, and the server always responds.*

DEFINITION: *A computation uses the* role symmetric paradigm *if it is role symmetric.*

At first glance these definitions may appear non-uniform, because they use seemingly different issues (contact initiation vs. role symmetry) to define the two paradigms. However, contact initiation is a characteristic of role asymmetry. To see this, note that process p_1 can only initiate contact with process p_2 in response to some change in the state of p_1 (e.g., input from a user or the expiration of a timer), which is modeled as an input message. Thus, the fact that a client initiates contact but a server does not indicates that these processes respond differently to such an input message, and thus that the computation is role asymmetric.

While our definition of the role symmetric paradigm establishes its connection with role symmetry directly, intuition might lead one to believe that there are a wide variety of differently organized role asymmetric computations. In particular, one can easily think of computations that have more than two functionally distinguished components that do not use the client/server paradigm. An example would be a system in which a collection of producer processes continuously updates a database while a collection of consumer processes continuously read from the database, using appropriate locking mechanisms to ensure consistency. However, it turns out that any role asymmetric computation can be cast as a computation that uses the client/server paradigm, through an appropriate mapping of components and functions. We state this as a theorem.

THEOREM: *Given any role asymmetric computation one can construct a functionally equivalent computation that uses the client/server paradigm.*

PROOF: The proof proceeds by construction. The first step is to represent the computation as a directed graph, where each node corresponds to a piece of the computation, each edge corresponds to a communication path between two of these pieces, and each edge direction indicates which node initiates contact between those two nodes. While it usually makes conceptual sense to decompose the computation into modules (or some unit of scheduling and distribution), how one partitions the computation is immaterial for the purposes of the proof. We need only show that it is possible to map a given decomposition graph to a graph in which there is at most a single directed edge between each two pieces (the client initiator condition), without changing the computation's functionality.

To perform the mapping, note that for each pair of nodes N_1 and N_2 joined by one or more edges, there are three possibilities:

1. There is a directed edge from N_1 to N_2. This means that N_1 always initiates communication with N_2.
2. There is a directed edge from N_2 to N_1. This means that N_2 always initiates communication with N_1.
3. There are directed edges in both directions. This means that N_1 sometimes initiates communication with N_2, and N_2 sometimes initiates communication with N_1.

For case (1), we can label N_1 the client and N_2 the server to complete the construction. Similarly, for case (2), we label N_2 the client and N_1 the server. For case (3), we construct a functionally equivalent directed acyclic graph corresponding to the cycle represented by the communication paths between N_1 and N_2. The idea is to divide N_1 into three components, representing the part that initiates communication with N_2, the part that is contacted through communication initiated by N_2, and a "buffer" process through which these first two components communicate. N_2 is divided similarly. This way, each component in the transformed graph can be labeled as exactly one role that can be implemented as a client or server, as appropriate.

The formal construction is as follows. Replace node N_1 with nodes N_{1C}, N_{1S}, and N_{1B} (which will act as a buffer between N_{1C} and N_{1S}). Similarly, replace node N_2 with nodes N_{2C}, N_{2S}, and N_{2B}. Then replace the two edges between N_1 and N_2 with the following six edges:

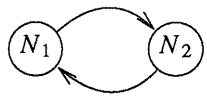
$N_{1C} \rightarrow N_{2S}$
$N_{2C} \rightarrow N_{1S}$
$N_{1C} \rightarrow N_{1B}$
$N_{1S} \rightarrow N_{1B}$
$N_{2C} \rightarrow N_{2B}$
$N_{2S} \rightarrow N_{2B}$

This construction is illustrated in Figure 1. Figure 1b shows the how the original computation shown in Figure 1a is decomposed using the formal construction described above. In Figure 1c, we have rotated the figure somewhat to show the logical structure of the computation more clearly.

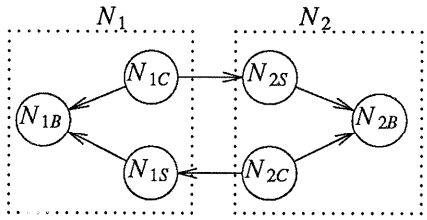
We now label N_{1C} and N_{2C} as clients, and N_{1S}, N_{2S}, N_{1B}, and N_{2B} as servers. The communication correspondence is established as follows. Wherever N_1 would have initiated communication with N_2 in the original computation, in the corresponding client/server paradigm computation N_{1C} calls N_{2S}. Any state shared between the component of N_1 that initiates contact with N_2 and the component of N_2 that initiates contact with N_1 is represented as explicit calls by N_{1C} to N_{1B} to store the state, and by N_{1S} to N_{1B} to retrieve it, or vice versa. A symmetrical construction holds for wherever N_2 would have initiated communication with N_1. This completes the construction and the proof.

We have shown that given any role asymmetric computation one can construct a functionally equivalent computation that uses the client/server computation. Since the client/server paradigm is role asymmetric by definition, this proof shows that the class of role asymmetric computations is the same as the class of computations that can be implemented by the client/server paradigm. Combining this result with our definition of the role symmetric paradigm completes the partioning of the computation space. We state this as a Corollary:

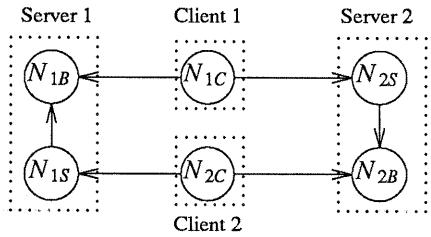
COROLLARY: *The condition of role symmetry partitions the universe of distributed computations into two classes corresponding to the client/server paradigm and the role symmetric paradigm, with the following correspondences: a computation can be implemented by the client/server paradigm if and only if it is role asymmetric; and a computation can be implemented by the role symmetric paradigm if and only if it is role symmetric.*



a. Original Computation



b. Decomposition of N_1 and N_2



c. Constructed Equivalent Computation

Figure 1: DAG Construction for Recursive Components

This result is illustrated in Figure 2.

4. The Basis of Role Symmetry; Composite Computations

In the previous section we showed that the condition of role symmetry can be used to characterize the set of computations that can be implemented with each of the paradigms examined in this paper. However, designers of distributed computations do not typically think about their tasks in terms of role symmetry. Instead, a computation's role symmetry/asymmetry normally derives from constraints placed on the computation. In this section we analyze the basis of role symmetry, and show how one can view various practical problems in distribution in terms of composing subcomputations in different ways according to role symmetry.

To analyze the basis of role symmetry, we begin by observing that in each of the role symmetric computations considered in Section 2, an important goal was to avoid centralization, for example to avoid single points of failure, performance bottlenecks, or the need to designate a globally agreed upon administrative authority. In fact, a role symmetric computation cannot centralize any of its state or control. If it could, the process responsible for the centralized component would respond to some requests differently than the other processes in that computation, violating the definition of role equivalence. Thus, if a computation is role symmetric, it must be distributed.

If the converse of this statement were true, we would have a complete characterization of the basis of role symmetry. However, many role asymmetric (client/server) computations support various degrees of distribution

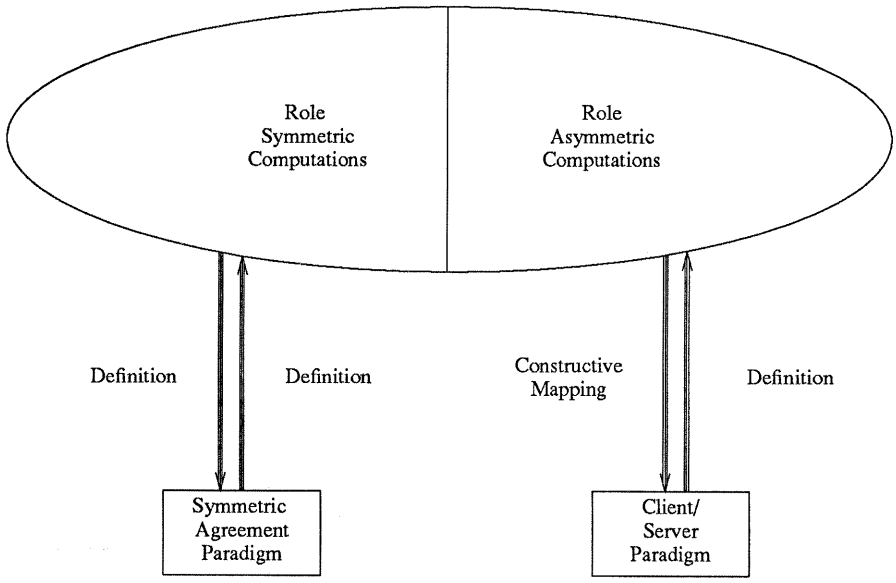


Figure 2: Computation Space Partitioning

(cf. Section 2). Therefore, the question to ask is what types of distribution require role symmetry. By definition, the answer is those computations for which it is required that any process will produce the same output messages for any given set of input messages. For example, a fully replicated database requires role symmetry, since otherwise some parts of the database would not be available from all processes.† We call this type of distribution *role symmetric distribution.*

While the client/server paradigm is role asymmetric, there are many client/server computations that require role symmetric distribution. In the database example, for instance, a convenient user interface could be provided in a client that accessed the (role symmetric) database across a network. This suggests an interesting way to look at such a client/server computation, namely, as a composite computation. At the top level is a client/server computation, where the server appears to be *logically* centralized from the client's perspective (i.e., the client is not concerned with the distributed nature of the server role). Below that is a conglomerate server, which is a distributed computation in its own right. This subcomputation can then be analyzed according to the role symmetry condition of Section 3. Services that need to support role symmetric distribution can be organized into role symmetric subcomputations, as illustrated in Figure 3.

On the other hand, the goal of distributing the server role may be to allow parts of a global resource to reside at distinct nodes (e.g., because the resource is very large or crosses administrative boundaries). We call this type of distribution *role asymmetric distribution.* This type of distribution requires the use of a role asymmetric server subcomputation. In such a case, the server role is often organized hierarchically, for example to facilitate scalability or the delegation of administrative authority. This organization is illustrated in Figure 4.

Of course, a computation could need both role symmetric and role asymmetric distribution. For example, the Domain Naming System makes use of a hierarchy of servers for dividing administrative responsibility, and server replication to provide a measure of fault tolerance and load distribution. One can view this system as a three level deep computation. At the top level is a simple client/server computation, where clients simply contact any server, without being aware of the distribution of the naming information. Below that, the abstract name service is a hierarchically organized, role asymmetric collection of services, each of which is authoritative over a particular part ("domain") of the naming tree. At the bottom level, each of these authoritative services is

†Many databases of a practicably large size would be infeasible to replicate fully. A common practice is to replicate parts of the database in such a fashion that any particular part can be found at several nodes, yet no part can be found at all nodes. We discuss how our model handles such situations shortly.

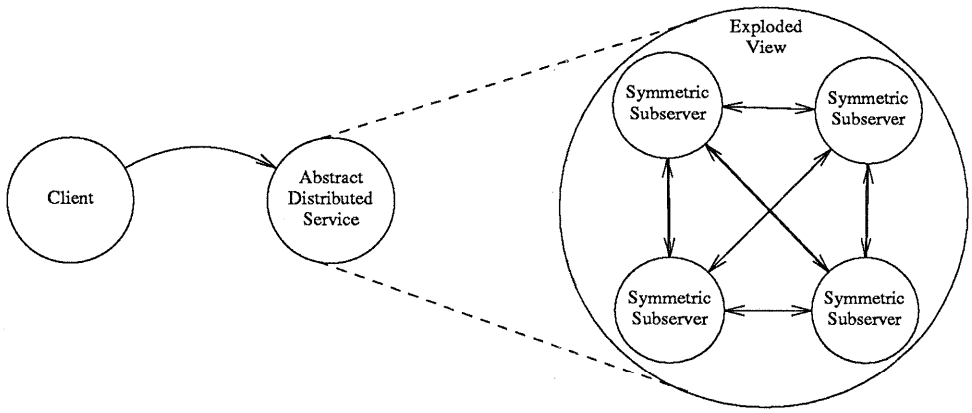


Figure 3: Client/Server Paradigm with Role Symmetric Server Subcomputation

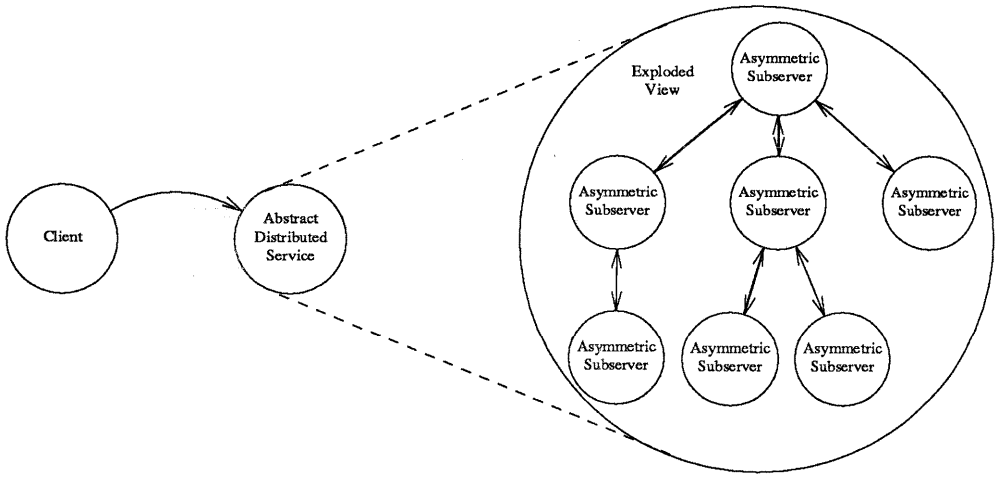


Figure 4: Client/Server Paradigm with Role Asymmetric Server Subcomputation

a role symmetric, replicated collection of servers. This decomposition is illustrated in Figure 5.†

The analysis done in this section shows how various practical problems in distributed computing can be viewed in terms of the composition of simpler subcomputations, each of which can be analyzed according to the structure of Section 3. In general, a computation could be composed of an arbitrarily deep nesting of subcompu-

†Actually, the Domain Naming System uses a less sophisticated form of replication than is pictured here, involving a primary site and some secondary replicas [Mockapetris & Dunlap 1988]. We have omitted this detail from the figure because it does not further illustrate the ideas behind the composite representation we have described.

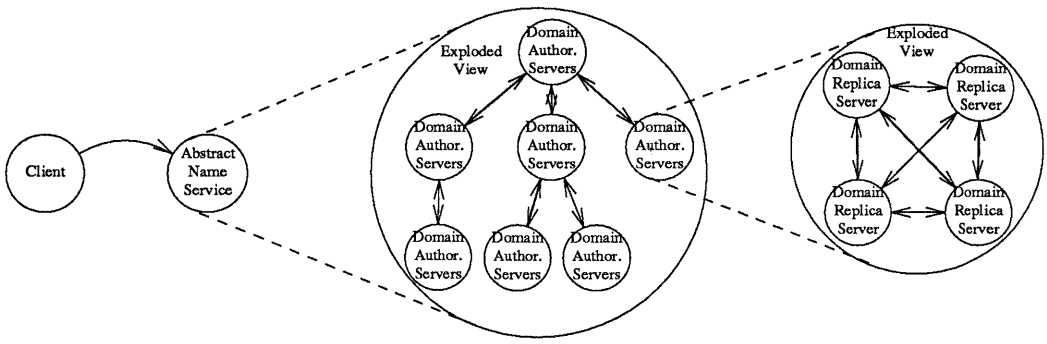


Figure 5: (Simplified) Composite Representation of Domain Naming System

tations using the basic constructions discussed in this section, as needed to solve the problem at hand.

5. Strengths and Weaknesses of the Paradigms

In this section we examine the strengths and weaknesses of each the paradigms considered in this paper. We begin with a brief example.

An often cited goal of distributed systems is network transparency, to improve the ease with which resources are shared [Tanenbaum & van Ranesse 1985, Walker et al. 1983]. For example, a pool of printers could be shared conveniently by placing them together in a room and attaching them to a computer running a server that load shares printing requests from clients around a local area network. Doing this would be easier, for instance, than forcing users to check queue lengths explicitly at each printer before specifying which printer to use.

A similarly coherent view can be provided using the role symmetric paradigm, by building a pool of printer control processes (one per printer) that elect a "service representative" each time a print request is received. Doing so would have the advantage that no one process' failure would disrupt access to the printers. However, to make this pool of processes easy to use, it would make sense to provide a client/server interface. The result is a computation structure like that of Figure 3.

This example illustrates the basic strengths of both paradigms. First, in an important sense, role symmetric distribution is a more "thorough" form of distribution than role asymmetric distribution, because in the former case no control or state can be centralized. Therein lies the strength of the role symmetric paradigm: when role symmetric distribution is required, only the role symmetric paradigm will suffice. Yet, from a user's perspective, the client/server paradigm is easier to use. Moreover, it seems easier to build and reason about computations that use the client/server paradigm than it is with computations that use the role symmetric paradigm. (Consider, for example, the volume of theoretical work that has been devoted to role symmetric paradigms, particularly in Byzantine agreement algorithms [Lamport, Shostak & Pease 1982], voting algorithms [Bloch, Daniels & Spector 1987], and common knowledge [Halpern & Moses 1984].)

To understand why the client/server paradigm might be easier to build, reason about, and use than the role symmetric paradigm, we begin with the observation that an important step in designing and building distributed computations is constructing appropriate abstractions. The client/server paradigm supports the development of abstractions by providing a powerful structuring mechanism for implementing abstractions, namely, *separation of concerns* between the client and server. While a server concerns itself with supporting an abstraction, a client need only consider how to make requests of the server. The role symmetric paradigm lacks this structuring tool.

In addition to supporting the construction of abstractions, the client/server paradigm also allows more general organizational structures than the role symmetric paradigm. By definition, all role symmetric computations must exhibit a fully connected ("flat") communication structure. Because it is role asymmetric, the client/server paradigm allows arbitrary organizations. Of particular importance is the ability to support hierarchy, which is

often used to enhance scalability.

The client/server paradigm's support for arbitrary organizational structures is also useful for the class of problems that have a recursively defined nature in which one process must "prime" the computation by solving an initial instance of the problem, and the remainder of the problem must be solved by another process. The most common example is an object location service. Before locating an object using an object location service, the client must first locate an object location server (e.g., by reading a local file containing the host name on which a server resides). Another example is an authentication service. Before using an authentication service to gain trust in the identity of some entity in the network, the client must first trust the identity of the authentication service (e.g., by trusting that the service runs on machines that reside in physically secure areas, and that network communication is secure). Some traditionally centralized computations also exhibit this character. For example, before a compiler can produce code, some initial instance of the code for the compiler must be produced without the use of a compiler.

The class of computations that exhibit this type of recursion is probably not very large compared with the set of all distributed computations. Yet, the issue deserves discussion because some very useful, general purpose computations fit this scenario. Moreover, the solution to these problems using the client/server paradigm has an interesting character: it is analogous to the construction of a proof by induction. The simple subcase corresponds to proving the base case of an induction proof. The base case of a computation (proof) is typically implemented (proven) very simply, and forms the basis for the rest of the computation (proof). The service's responsibility in the computation corresponds to the inductive step in the proof. The server's (inductive step's) implementation typically requires some sophistication, and its usefulness (correctness) depends on the base case.

While the need to solve a base case of the problem in the client is in conflict with the goal of separation of concerns, this is not a fault with the client/server paradigm. Rather, it is a characteristic of the problem at hand. In fact, only the client/server paradigm can usefully support this class of computations. Using the role symmetric paradigm one would be forced to build both the base case and the inductive step into each process. Usually it would not make sense to do this in the case of a general purpose computation, since that would not allow the common infrastructure to be shared. (For example, it would mean that every process were capable of authenticating all network entities.)

In summary, the role symmetric paradigm provides the sole means of supporting role symmetric distributed computations, but it is relatively difficult to build, reason about, and use computations based on this paradigm. The client/server paradigm cannot support this class of computations, but eases the process of building a distributed computation by virtue of role separation. In addition, it is a more flexibly organized paradigm, in that it can support arbitrary organizations across processes, including, in particular, hierarchy and recursion. The features of both paradigms can be obtained by building composite computations, as illustrated in Section 4.

6. Conclusions

In this paper we discussed two popular distributed computing paradigms: the client/server paradigm, and the role symmetric paradigm. We established a simple role symmetry condition that partitions the space of distributed computations into two parts corresponding to these two paradigms. We then showed how to construct a functionally equivalent computation that uses one of these paradigms, given any distributed computation. Because programmers do not typically think explicitly about role symmetry when designing computations, we explored the basis of this type of symmetry in distributed computations, showing the situations where each paradigm is required, based on the constraints placed on a computation. In doing so, we showed how practical computations can be viewed as compositions of role symmetric and role asymmetric subcomputations. Finally, we analyzed the strengths and weaknesses of the two paradigms. We showed that the role symmetric paradigm is required whenever a powerful type of distribution that we call *role symmetric distribution* is needed in a computation. In contrast, the client/server paradigm supports the development of abstractions through separation of concerns. It also allows arbitrary organization of the processes comprising the computation, whereas the role symmetric paradigm only allows "flat" organization. Of particular interest are hierarchical organizations, which support scalability; and recursive organizations, which are used in some very common practical computations, such as name and authentication services.

In summary, any distributed computation can be viewed as a composition of one or more subcomputations, each of which uses one of the two paradigms. Each of these paradigms has certain strengths and weaknesses for

building and understanding distributed computations, which are inherited by any subcomputation that uses that paradigm. Therefore, dividing the universe of distributed computations into these two paradigms is useful, because a programmer developing a distributed computation can be aware of the properties each paradigm possesses in solving a specific problem, and thereby more clearly design each piece of the computation.

Acknowledgements

We would like to express our appreciation for many interesting discussions about this work with Calton Pu, and for the helpful comments we received from Richard Ladner, Gary Nutt, and John Zahorjan, who provided feedback on an earlier draft of this paper. We are also grateful to Sharon Smith and Mark Squillante, whose feedback helped us in the preparation of the final version of this paper.

7. References

[Accetta et al. 1987]
M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian and M. Young. Mach: A New Kernel Foundation for UNIX Development. *Proc. 33rd IEEE Comput. Society Int. Conf.*, Feb. 1987.

[Almes et al. 1985]
G. T. Almes, A. P. Black, E. D. Lazowska and J. D. Noe. The Eden System: A Technical Review. *IEEE Trans. Software Eng.*, SE-11(1), pp. 43-59, Jan. 1985.

[Alon, Barak & Manber 1987]
N. Alon, A. Barak and U. Manber. On Disseminating Information Reliably Without Broadcasting. *Proc. 7th IEEE Int. Conf. Distrib. Comput. Syst.*, pp. 74-81, Sep. 1987.

[Birrell et al. 1986]
A. D. Birrell, B. W. Lampson, R. M. Needham and M. D. Schroeder. A Global Authentication Service Without Global Trust. *Proc. IEEE Symp. on Security and privacy*, pp. 223-230, Apr. 1986.

[Bloch, Daniels & Spector 1987]
J. J. Bloch, D. S. Daniels and A. Z. Spector. A Weighted Voting Algorithm for Replicated Directories. *J. ACM*, 34(4), pp. 859-909, Oct. 1987.

[Chandy & Misra 1986]
K. M. Chandy and J. Misra. How Processes Learn. *Distributed Computing*, 1(1), pp. 40-52, Jan. 1986. Presented at the Fifth ACM Symp. Principles Distr. Comput., Alberta, Canada, Aug. 1986.

[Chang & Maxemchuk 1984]
J. M. Chang and N. F. Maxemchuk. Reliable Broadcast Protocols. *ACM Trans. Comput. Syst.*, 2(3), pp. 251-273, Aug. 1984.

[Cheriton & Mann 1988]
D. R. Cheriton and T. P. Mann. Decentralizing a Global Naming Service for Improved Performance and Fault Tolerance. *ACM Trans. Comput. Syst.*, 7(2), pp. 147-183, May 1988.

[Gifford 1979]
D. K. Gifford. Weighted Voting for Replicated Data. *Proc. 7th ACM Symp. Operating Syst. Prin.*, Dec. 1979.

[Gordon 1988]
M. Gordon. Probabilistic and Genetic Algorithms in Document Retrieval. *Commun. ACM*, 31(10), pp. 1208-1218, Oct. 1988.

[Gusella & Zatti 1986]
R. Gusella and S. Zatti. An Election Algorithm for a Distributed Clock Synchronization Program. *Proc. 6th IEEE Int. Conf. Distrib. Comput. Syst.*, pp. 364-371, May 1986.

[Hagmann 1986]
R. Hagmann. Process Server: Sharing Processing Power in a Workstation Environment. *Proc. 6th IEEE Int. Conf. Distrib. Comput. Syst.*, pp. 260-267, May 1986.

[Halpern & Moses 1984]
J. Y. Halpern and Y. Moses. Knowledge and Common Knowledge in a Distributed Environment. *Proc. 3rd ACM Symp. Principles Distr. Comput.*, pp. 50-61, Aug. 1984.

[Harrenstien, Stahl & Feinler 1985]
K. Harrenstien, M. Stahl and E. Feinler. NICName/Whois. Req. For Com. 954, Oct. 1985.

[Howard et al. 1988]
J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham and M. West. Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, 6(1), pp. 51-81, Feb. 1988. Presented at the Eleventh ACM Symp. Operating Syst. Prin., Austin, TX, Nov. 1987.

[Hudson & King 1988]
S. E. Hudson and R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. Tech. Rep. 88-20, Dept. Comput. Sci., Univ. Arizona, Tucson, AZ, 1988.

[Lamport, Shostak & Pease 1982]
L. Lamport, R. Shostak and M. Pease. The Byzantine Generals Problem. *ACM Trans. Prog. Lang. Syst.*, 4(3), pp. 382-401, July 1982.

[Martin, Bergan & Russ 1987]
B. Martin, C. Bergan and B. Russ. PARPC: A System for Parallel Procedure Calls. *Proc. IEEE Int. Conf. Parallel Processing*, pp. 449-452, Aug. 1987.

[McQuillan, Richer & Rosen 1980]
J. M. McQuillan, I. Richer and E. C. Rosen. The New Routing Algorithm for the ARPANET. *IEEE Trans. Commun.*, COM-28(5), pp. 711-719, May 1980.

[Metcalfe & Boggs 1976]
R. M. Metcalfe and D. R. Boggs. Ethernet: Distributed Packet Switching for Local Computer Networks. *Commun. ACM*, 19(7), pp. 395-404, July 1976.

[Mockapetris & Dunlap 1988]
P. Mockapetris and K. J. Dunlap. Development of the Domain Name System. *Proc. ACM SIGCOMM Symp.*, pp. 123-133, Stanford, CA, Aug. 1988.

[Mueller, Moore & Popek 1983]
E. T. Mueller, J. D. Moore and G. J. Popek. A Nested Transaction Mechanism for LOCUS. *Proc. 9th ACM Symp. Operating Syst. Prin.*, pp. 71-89, Oct. 1983.

[Nelson & Snyder 1987]
P. A. Nelson and L. Snyder. Programming Paradigms for Nonshared Memory Parallel Computers. In L. H. Jamieson, D. B. Gannon and R. J. Douglas, editors, *The Characteristics of Parallel Algorithms*, MIT Press, Cambridge, MA, 1987.

[Nelson, Welch & Ousterhout 1988]
M. Nelson, B. Welch and J. Ousterhout. Caching in the Sprite Network File System. *ACM Trans. Comput. Syst.*, 6(1), pp. 134-154, Feb. 1988. Presented at the Eleventh ACM Symp. Operating Syst. Prin., Austin, TX, Nov. 1987.

[Notkin et al. 1987]
D. Notkin, N. Hutchinson, J. Sanislo and M. Schwartz. Heterogeneous Computing Environments: Report on the ACM SIGOPS Workshop on Accommodating Heterogeneity. *Commun. ACM*, 30(2), pp. 132-140, Feb. 1987.

[Salton 1986]
G. Salton. Another Look at Automatic Text-Retrieval Systems. *Commun. ACM*, 29(7), pp. 648-656, July 1986.

[Saltzer 1982]
J. H. Saltzer. On the Naming and Binding of Network Destinations. In P. C. Ravasio, G. Hopkins and N. Naffah, editors, *Local Computer Networks*, pp. 311-318, North-Holland, New York, NY, 1982.

[Sandberg et al. 1985]
R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh and B. Lyon. Design and Implementation of the Sun Network File System. *Proc. USENIX Summer Conf.*, pp. 119-130, June 1985.

[Scheifler & Gettys 1986]
R. W. Scheifler and J. Gettys. The X Window System. *ACM Trans. Graphics*, 5(2), pp. 79-109, Apr. 1986.

[Schwartz, Zahorjan & Notkin 1987]
M. F. Schwartz, J. Zahorjan and D. Notkin. A Name Service for Evolving, Heterogeneous Systems. *Proc. 11th ACM Symp. Operating Syst. Prin.*, pp. 52-62, Nov. 1987.

[Schwartz 1989]
M. F. Schwartz. The Networked Resource Discovery Project. *IFIP XI World Congress*, pp. 827-832, San Francisco, CA, Aug. 1989.

[Spector 1982]
A. Spector. Performing Remote Operations Efficiently on a Local Computer Network. *Commun. ACM*, 25(4),

pp. 246-260, Apr. 1982. Presented at the Eighth ACM Symp. Operating Syst. Prin., Pacific Grove, CA, Dec. 1981.

[Steiner, Neuman & Schiller 1988]
J. G. Steiner, C. Neuman and J. I. Schiller. Kerberos: An Authentication Service for Open Network Systems. Project Athena Report, MIT Lab. for Comput. Sci., Mar. 1988.

[Sun Microsystems 1985]
Sun Microsystems. *Remote Procedure Call Programming Guide*. Sun Microsystems, Inc., Mountain View, CA, Jan. 1985.

[Tanenbaum & van Ranesse 1985]
A. S. Tanenbaum and R. van Ranesse. Distributed Operating Systems. *ACM Comput. Surv.*, 17(4), pp. 419-470, Dec. 1985.

[Tanenbaum 1987]
A. S. Tanenbaum. A UNIX Clone with Source Code for Operating Systems Courses. *Operating Syst. Review*, 21(1), pp. 20-29, Jan. 1987.

[Walker et al. 1983]
B. Walker, G. Popek, R. English, C. Kline and G. Thiel. The LOCUS Distributed Operating System. *Proc. 9th ACM Symp. Operating Syst. Prin.*, pp. 49-70, Oct. 1983.

[Wang & Morris 1985]
Y. T. Wang and R. J. T. Morris. Load Sharing in Distributed Systems. *IEEE Trans. Comput.*, C-34(3), pp. 204-217, Mar. 1985.