A Service Acquisition Mechanism for Server-Based Heterogeneous Distributed Systems

Rong N. Chang, *Member, IEEE*, and Chinya V. Ravishankar, *Member, IEEE*

***Abstract*—This paper presents a mechanism that facilitates and enhances the use of independently administered remote network servers in the presence of server interface heterogeneity. The mechanism is designed under the *client-service* model, which extends the client-server model with an abstraction of *service* to decouple abstract server capabilities from concrete server interface specifics such as server interface binding protocols and the interface operation invocation protocols. The mechanism selects servers, accommodates server interface heterogeneity, and handles server access failures as per the abstract server capabilities desired by the client. It could return the identity of the server used for each service access invocation to facilitate billing, refining service specifications, and reporting server-specific errors.**

This paper also illustrates a C library interface to this mechanism, and describes a language veneer over the C programming language demonstrating how a typed procedural language could be extended by a few language constructs to support the mechanism under the client-service model. In this language, server capabilities are referenced by abstract data type (ADT) objects, and are accessed by invoking the objects' interface operations using a call-by-value-result paradigm. This language veneer also makes it easier to port the client software across to systems that use different service specification schemes.

Our work suggests that this mechanism facilitates the development, use, and maintenance of client and server software in large heterogeneous distributed systems comprising many autonomous servers. It also shows that the overhead of invoking remote server operations via the mechanism can be quite low.

***Index Terms*—Heterogeneous distributed systems, client-server model, client-service model, service acquisition mechanism, attribute-based naming, remote procedure call, server interface directory service, agent process, fault tolerance, object-oriented programming, language veneer.**

I. INTRODUCTION

THIS paper presents the design and implementation of a mechanism that facilitates and enhances the use of independently created and administered remote network servers in the presence of server interface heterogeneity. This mechanism [5] is designed under an extended client-server model called the *client-service* model. In this model, an abstraction of *service* is introduced to decouple abstract server capabilities from concrete server interface specifics such as server interface binding protocols and the interface operation invocation protocols. The mechanism selects servers, accommodates server interface heterogeneity, and handles server access failures as per the abstract server capabilities desired by the client. It could return the identity of the server used for each service access invocation to facilitate billing, refining service specifications, and reporting server-specific errors.

This paper also illustrates a C library interface to this mechanism, and describes a language veneer over the C programming language [4] demonstrating how a typed procedural language may be extended by a few language constructs to support the mechanism under the extended client-server model. In this language, server capabilities are referenced by *Abstract Data Type* (ADT) objects, and are accessed by invoking the objects' interface operations using a call-by-value-result paradigm, independent of the interfaces exported by the servers in use.

This language veneer also facilitates the development of ADT-like libraries to model available network services. Such libraries would greatly simplify the task of specifying services abstractly. Parametrized objects from the libraries may simply be included in the client software, making it easier to port the client software across to systems that use different service specification schemes.

Our work suggests that this mechanism facilitate the development, use, and maintenance of client and server software in large heterogeneous distributed systems comprising many autonomous servers. It also shows that the overhead of invoking remote server operations via the mechanism can be quite low.

A. Client-Server Computing Issues

Before distributed computing came into vogue, resource management functions were provided solely by monolithic operating system kernels [16]. When many of these functions migrated out of the kernels into user-level processes to improve system maintainability, extensibility, scalability, and cost-performance ratios, user-level resource managers became known as *servers* and the *service* notion arose as a convenient abstraction of server capability [25], [26]. As centralized computing models became obsolete, the service notion evolved into a major abstraction for managing and using networked resources. The *client-server* model, which many current distributed computing systems use [22], represents the prevalent implementation of this approach.

When the number of servers grows, however, the client-server model gives rise to an unsatisfactory paradigm for client applications to acquire server capabilities or computational services. Since server capabilities cannot be cleanly decoupled from server interfaces in the client-server model,

Manuscript received November 12, 1991; revised December 15, 1992. This work was supported in part by a grant from Bell Northern Research, Inc.

R. N. Chang was with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109 and Bell Communications Research, Morristown, NJ 07962. He is now with IBM, White Plains, NY 10605.

C. V. Ravishankar is with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, MI 48109.

IEEE Log Number 9214465.

1045–9219/94$04.00 © 1994 IEEE

they must be tied to implementation specifics like server interface binding protocols, interface operation signatures, and operation invocation protocols. Thus, client applications must choose server interfaces matching the desired capabilities and confront server-specific implementation details. They must also implement server-dependent fault-tolerant algorithms to enhance their reliability when a desired server capability can be provided by several servers.

For example, in the distributed computing environment (DCE) [20] promoted by the Open Software Foundation (OSF), the interface to a DCE *Remote Procedure Call (RPC)* server is a set of typed operations and may be defined via an interface definition language. A specialized distributed database registers and advertises the associations between servers and their RPC interfaces. A client could query the database to search for appropriate servers and to obtain necessary binding information. It must use the DCE RPC package to make client-server bindings and to invoke server interface operations. It also is in charge of handling incomplete invocations of the operations, should the client-server binding be broken unexpectedly because of network partition or server failure. In the client-server model, the RPC run-time is not obliged to automatically reconfigure the client-server bindings on behalf of the client.

In order to overcome such deficiencies in using network servers, we must address the issues concerning 1) how to specify server capabilities (or services) so that the association between clients and servers can be changed dynamically without disturbing the clients, 2) how to accommodate server interface heterogeneity to support such a specification scheme and maximize the utilization of the servers, and 3) how to reconfigure the bindings between server capabilities and servers when appropriate, without interfering with the client.

These issues are key to the development of robust client applications, to extending the life cycle of client software when changes to networking technology are inevitable, to better using server applications, and to encouraging the exploration of new server access protocols for improved or specialized high-performance servers [6].

B. Partial Solutions

Although mechanisms have been developed to address some hard issues in distributed systems [17], none of them provide an integrated solution to the issues we have outlined. For instance, as a partial solution to the specification issue, generic names are sometimes assigned to stateless server interfaces as service identities, so that service-server bindings can be changed between accesses to the servers. This happens in the run-time support for remote program execution services in Marionette [23], which forwards client requests to a new server, should the current one be unreachable. Most solutions to accommodating server interface heterogeneity are limited in the networking protocols considered. For example, the HCS RPC run-time [1] is able to emulate Sun RPC and Xerox Courier RPC. Servers speaking these RPC's may be accessed by HCS RPC clients with no modifications, though the clients must configure the emulation library operations correctly for each client-server binding. Similarly, fault-tolerance support for accessing server capabilities is usually enhanced by improving client-server communication protocols. For example, in the ISIS [2] distributed system, an access request can be multicast to a group of servers so that access fails only when all servers are inaccessible.

- **Client/Server Model**
  - **Client requests client-server bindings.**
  - **Client accesses services via server-dependent protocols.**
  - **Client releases client-server bindings.**
  - **Server identity is determined when client-server binding is established.**
  - **Client handles broken client-server bindings.**
- **Client/Service Model**
  - **Client requests client-service bindings.**
  - **Client could access services via server-independent protocols.**
  - **Client releases client-service bindings.**
  - **Server identity is determined when service-server binding is established.**
  - **Client handles broken client-service bindings and need not to deal with broken service-server bindings.**
  - **A service access invocation could return the identity of the server used to facilitate billing, refining service specifications, and reporting server-specific errors.**

Fig. 1. Client-server model versus client-service model.

C. The Client-Service Model

This paper presents a service acquisition mechanism that provides a framework for an integrated solution to the three issues mentioned above. This mechanism is designed under the *client-service* model [4], which extends the client-server model with the abstraction of *service*[1] to decouple abstract server capabilities from concrete server interfaces. A service abstraction in the model is a description of the processing capabilities that some server may provide. Fig. 1 provides a comparison of key features of these two models. In contrast to the client-server model, a client in the client-service model uses uniform mechanisms to establish or release client-service bindings and to obtain services from servers. The client need not deal with server interface heterogeneity or handle server access failure. In comparison with (generic) server interface names, the specification of such a service (or a server capability) represents 1) a set of server selection criteria, 2) a set of service access operations[2] that can be supported by a single server, and/or 3) a set of service-server reconfiguration constraints. To facilitate billing and reporting server-specific errors, a service access invocation could return the identity of the server used as a standard output argument.

D. The Cygnus Service Acquisition Mechanism

A prototype implementation of the mechanism has been evaluated in a server-based heterogeneous distributed system at the University of Michigan at Ann Arbor. This system, named the Cygnus Distributed System, contains several au-

[1] In the client-service model, the terms *services* and *server capabilities* are interchangeable.

[2] These operations *may* have to be translated into the interface operations exported by a selected server if the two are different. Further details are given in Sections III-A and III-C.

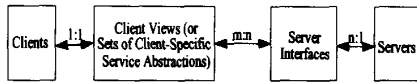


Fig. 2. The extended client-server model in Cygnus.

tonomous network servers that may be accessed through various intermachine interprocess communication (IPC) mechanisms, such as Sun RPC, NCS RPC, and BSD UNIX sockets. The servers export their interfaces through various *server interface directory servers* such as Sun's Network Information System and RPC port mappers, NCS location brokers, Oracle databases, and Profile name servers.

This paper also illustrates a C library interface to the mechanism, and presents the design and implementation rationale of an experimental language, named CygnusC [4], to demonstrate how a C-like typed procedural language can be extended with a few language constructs to support the mechanism. In this language, server capabilities are referenced as abstract data type (ADT) objects that are instantiated at run-time via special templates. These templates permit the compiler to type check service acquisition operations, and enable the language run-time to invoke user code for validating service specifications and handling service access failures. A call-by-value-result paradigm is used to invoke the objects' methods (or interface operations). Each operation performed on the objects is mapped transparently to one or more remote server invocations.

E. Organization

This paper is organized as follows. Section II elaborates on the computational model supported by the Cygnus service acquisition mechanism. Section III illustrates the design and implementation of the Cygnus service acquisition mechanism. Section IV explains how the Cygnus service acquisition operations can be used to access a network service. Section V demonstrates how the mechanism can be supported in typed procedural languages like C. Section VI analyzes the cost of using the Cygnus service acquisition mechanism to access local or remote servers in our prototype. Section VII describes several typical client-service applications that we have built. Section VIII discusses some of the lessons we learned from the development of several applications. Section IX concludes the paper.

II. The Cygnus Computational Model

Fig. 2 shows the extended client-server model supported by the Cygnus service acquisition mechanism. In the Cygnus distributed system, a client or a server is a computing entity (e.g., a UNIX process) that is developed, installed, and maintained as a unit. A Cygnus client sees the network as a collection of server-based service abstractions, and every service access operation returns the identity of the server used as a standard output argument. Each service abstraction is bound to exactly one server interface. (See Section I-C.) Coordinated access to shared servers must be handled either by the servers themselves or by a coordinator, which is itself a server.

The set of service abstractions visible to each client is called its *view* of the network. A one-to-one relation exists between clients and views, because each client-generated abstract service specification must be interpreted in a client-specific context.

Server interfaces are shown in Fig. 2 because they represent the sets of operations that the servers are willing to support. A many-to-one relation exists between server interfaces and servers, because a server may have several interfaces, but each interface belongs to only one server. As an example, a server running on the Internet may support a set of operations either through the connection-oriented transport protocol, TCP, or through the unreliable datagram protocol, UDP.

The relation between views and server interfaces is many-to-many, because an abstract service may represent some abstract functionality common to several concrete server interfaces, and each server interface may be associated with several abstract services. For example, a service designated abstractly as a text message delivery service may be provided concretely through the interface of an electronic mail server, a fax server, or an alphanumeric paging server, whichever happens to be most appropriate. In addition, an electronic mail server interface may be used to realize multimedia document transfer services as well.

A. Cygnus Service Specifications

In the Cygnus distributed system, the abstract services in client views are specified by sets of name-value pairs or attributes. As an example, a Cygnus client may use the following set of name-value pairs to acquire a personal messaging capability that delivers a text message from Bob to Allen over the particular communicators (e.g., telephones, fax machines, or portable computers) that Allen may be using at the moment of delivery:

```
((CONTEXT, messager), (SENDER, Bob),
 (RECEIVER, Allen), (ACCESS_INTERFACE, send)).
```

A principal advantage of dealing with this service as a high-level abstraction is to free Bob from the onus of knowing where Allen is or from the identities of the communicators that Allen may be using to receive information. Although the information must be sent by using the specified server-independent operational interface `send`, it may be transformed from one medium to another (e.g., from text to voice) by a server, depending on the chosen receiving communicator. The attribute `CONTEXT` indicates the context for interpreting the other name-value pairs, and is used to simplify the interpretation of Cygnus service specifications. (See Section III-D.)

A Cygnus service specification may contain attributes (e.g., MIPS for processor services, lines per second for compile service, and points per character for print services) that are orthogonal to the functional specification of the associated service access interface. We think that this feature is important because it encourages the clients (and the end users who run them) to exploit their knowledge about the computing environment. In other words, this feature allows the cost-

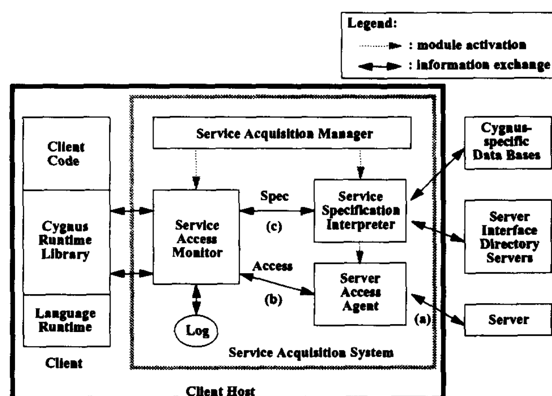


Fig. 3. Cygnus distributed system architecture.

performance ratio of a service operation to be reduced easily at run-time, with no need to change the client code.

We could have chosen to specify Cygnus services with character strings (like UNIX file path names) or through formal languages, without compromising the generality of the Cygnus computational model. Considering the combinations of the possible criteria for classifying services, however, string names would quickly lead to unmanageably large name spaces, though they might require simpler name resolution algorithms. On the other hand, formal specification methods would impose too much overhead in encoding and/or decoding service specifications, though they might help eliminate ambiguous specifications.

In contrast to the name-value pairs used in other attribute-based name/directory systems [18], [19], [21], the Cygnus service specifications are used to facilitate choosing servers providing services (or server capabilities) that the client desires, and for reconfiguring the bindings between (abstract) services and concrete server interfaces. Instead of getting back a set of object identities or object property lists, Cygnus clients obtain a reference to a client-service binding for each service specification that is honored by the service acquisition mechanism. We illustrate the service specification interpretation scheme in Section III-D and describe our experience with it in Section VIII.

III. The Service Acquisition Mechanism

Fig. 3 sketches the software architecture of the Cygnus distributed system. The servers are autonomous computing entities, and export their interfaces through various server interface directory servers like the ANSA/ISA *trader* [11]. The clients have no jurisdiction over the servers, because the mechanism is designed to use independently developed remote servers. (See Section I-A.) The Service Acquisition System hides the service realization details from the clients and is central to our realization of the Cygnus model. The Cygnus run-time library insulates the client code from the implementation details of the Service Acquisition System and promotes the portability of the client code. The clients trust the Service Acquisition System. (See Section VIII.)

The Service Acquisition System resides on the client host for two reasons. First, we want its links to clients to remain intact, even upon network failure or partition. Second, we treat servers as autonomous entities that may not be willing to run additional software. The system is composed of a *Service Acquisition Manager* and three kinds of user-level[3] processes: *Service Specification Interpreters*, *Service Access Monitors*, and *Server Access Agents*. The Service Acquisition Manager responds to client requests by creating a Service Specification Interpreter and a Service Access Monitor per service acquisition session. Service Specification Interpreters analyze service specifications, determine service-server bindings, and activate Server Access Agents locally. They also help the Service Access Monitors handle server access failures on behalf of the clients by using server-independent[4] algorithms. Server Access Agents accommodate server interface heterogeneities and implement server-dependent fault-tolerance algorithms, so that existing servers need not be modified to make them accessible to Cygnus clients.

The Cygnus run-time library includes a set of Cygnus-specific operations for use by the client code to communicate with the trusted Service Acquisition System running on the same machine using the RPC paradigm [3]. These operations are compiler-dependent, because the transformation between local service acquisition invocations into interprocess communication (IPC) messages depends on the language run-time associated with the compiler. Parameters are passed to these operations by using a call-by-value-result paradigm.

A. Service Acquisition Phases

We now briefly describe the working principles of the service acquisition mechanism. Further design and implementation details will be given in the following sections.

Service Request Phase: At the start of each service acquisition session, the client first contacts the local Service Acquisition Manager to get a service request port, which is a communication endpoint supported by the Cygnus internal IPC facility. It then composes a service request message, ships out the request through the service request port, and waits for an acknowledgment.

When the Service Acquisition Manager gets the client's request for a service request port, it first creates a Service Access Monitor and a Service Specification Interpreter. It then passes one of the Service Access Monitor's internal IPC ports back to the client so that the client can establish a link to the monitor. It then prepares to serve the next request.

The Service Access Monitor first makes connections to the associated Service Specification Interpreter and client. It then accepts the client's service specification message, and forwards the message verbatim to the interpreter. It also saves the message internally to implement service-dependent, server-

[3] The system was implemented by user-level processes because the interfaces between them were considered more important than the implementation performance. See [4] for further design rationale of the Cygnus distributed system.

[4] The algorithm is *server*-independent because the Service Access Monitor records service access requests and results in *service*-specific format. The algorithm is sketched in Section III-A, explained in detail in Section III-E, evaluated in Section VI-B, and reviewed in Section VIII.

independent service-server reconfiguration algorithms. After a Server Access Agent is activated by the Service Specification Interpreter for the requested service, the monitor establishes a link to it on behalf of the client. Finally, it creates a new service access port and returns it to the client.

Service Access Phase: After the client-service binding request is honored, the client invokes service operations through the service access port as necessary and awaits results.

When the Service Access Monitor receives an invocation request, it forwards the request to the Server Access Agent, which translates the request into one or more invocations on the associated remote server. It also receives the execution results from the Server Access Agent and returns them verbatim to the client. The monitor may save the access request and the execution results into a log buffer according to a logging-and-replay descriptor given by the Service Specification Interpreter in the service specification phase. This descriptor instructs the Service Access Monitor on how to correctly replay logged service operations, should the link between the monitor and the Server Access Agent be broken abnormally.

Service Reconfiguration Phase: The Cygnus Service Acquisition System uses two kinds of failure recovery algorithms to make clients more resilient to network or server failure. The first of these depends on the server in use. For transaction-based servers, for example, the Server Access Agent may stop its execution until the server machine is up again, so that the server state can be restored correctly. The second uses a server-independent operation logging-and-replay algorithm so that in the event of server (or Server Access Agent) failure, the Service Access Monitor can send all of the logged service access requests to another server through a new Server Access Agent. Should the new server interface be different from the old one, the new Server Access Agent reconverts the logged service access requests to server access requests for the new server. Such reconversion is possible because Server Access Agents are designed to process server access requests arriving in server-independent format.

Fig. 3 labels three links as (a), (b), and (c). Any one of these links may be broken during the service access phase. For example, link (a) may be broken because of server failure. The Server Access Agent may close link (b) when it does not get the execution results from the server in time. Link (c) may be cut by the Service Specification Interpreter because the number of requests for generating server access agents exceeds a predefined limit.

To make the client's service access link resilient to such faults, the Service Access Monitor always asks the Service Specification Interpreter for a new Server Access Agent when it finds the current one unavailable. If a new Server Access Agent can be created, the monitor replays the logged operations. It also ensures the correctness of this replay procedure by comparing the new execution results with the old ones. The new Server Access Agent may be instructed by the Service Specification Interpreter to eliminate some side effects (e.g., removing temporary work files) caused by the old server(s) and Server Access Agent(s) when it starts.

If link (c) in the figure had been broken when the current Server Access Agent died, the Service Access Monitor asks the Service Acquisition Manager for another Service Specification Interpreter. It then sends the saved service specification to the new interpreter, and reinvokes all of the logged service operations after link (b) is successfully restored. If the Service Acquisition Manager is unable to create the required interpreter because, for example, the kernel has run out of process table entries, the monitor informs the client that the service was interrupted unexpectedly.

Service Termination Phase: A service acquisition session is terminated when the client invokes the service termination operation in the Cygnus run-time library. After the Service Access Monitor receives the message from the client, it 1) forwards the message to the Server Access Agent, 2) releases its link to the Service Specification Interpreter if that link is still active, and 3) terminates itself after performing some other housekeeping routines like cleaning up log buffers. The Server Access Agent terminates after notifying the remote server(s) in use. The Service Specification Interpreter terminates after the monitor and Server Access Agent exit.

B. The Cygnus Run-Time Library

The Cygnus run-time library includes a set of compiler-dependent service acquisition primitives and hides the implementation details of the Service Acquisition System. The required set of OS-dependent IPC routines, which realize the IPC links, are also included in the library. In this section, we describe the *Application Programming Interface* (API) provided by the library. The use of the API is exemplified in Sections IV and V.

Cygnus IPC Operations: To facilitate the service-server reconfiguration support (see Section III-A), Cygnus IPC links are designed as reliable two-way communication channels and support atomic send and receive operations. They are implemented as follows. Each Cygnus IPC link is associated with a shared memory segment and two (System V) named pipes or FIFO's. To send a message, the sender places the message in the shared memory segment and writes a one-byte control token through the sender-to-receiver FIFO. The receiver determines whether a message is available through a read operation on the same FIFO. Because the file descriptors allocated to FIFO's are closed automatically when their owners terminate, the write operation returns an error code if the receiver dies unexpectedly. No special exception-handling or time-out mechanism is required.

The message-passing control mechanism could have been implemented instead by using semaphores, UNIX-domain stream sockets, or Internet-domain stream sockets. FIFO's were chosen because they appeared to perform better under normal loading conditions on our client host [4].

We have exploited the shared memory mechanism to further reduce message-processing overhead. Cygnus clients are required to initialize (or format) the shared memory segments that they acquire for accessing Cygnus services. This format is carefully designed to allow the processes involved to compose and decompose messages efficiently through data structures resident in the shared memory.

A set of name-based data types is defined to facilitate the communication among Cygnus clients and the service acquisition agents. The representation scheme for these data types depends on the IPC facility in use so that these processes can encode and decode messages efficiently. This data representation approach is different from structure-based ones, such as Sun's XDR.

These internal data types are useful to the implementation of the Cygnus service acquisition mechanism, because the mechanism must accommodate three kinds of heterogeneity. First, since the Service Specification Interpreters may need to access several different database servers to analyze service specifications, they must accommodate database query protocol heterogeneity. For example, most relational database servers support SQL queries, whereas most name servers like DEC's distributed name service [14] have their own query protocols to meet functional requirements such as access and/or update performance.

Second, different servers may use different data representation protocols. Therefore, the Server Access Agents must accommodate such protocol heterogeneity in converting service operations to server access requests. For example, Sun RPC servers use Sun's XDR representation scheme, whereas DEC HDS [8] servers understand Network Command Language (NCL) data types only.

Third, since different client language run-times may support different sets of data types, using a single internal data type representation scheme facilitates the development of library routines for each language run-time.

Service Acquisition Primitives: `RequestService`, `AccessService`, and `TerminateService` are the three service acquisition primitives provided by a Cygnus library. These operations hide the implementation details of the service acquisition mechanism by using keyword arguments and a call-by-value-result paradigm, and by forcing the client code to refer to client-service bindings through specialized opaque pointer structures called *service handles.*

A service handle must be initialized to hold service request messages, and must be bound to a service before it can carry service access requests. The Cygnus library includes `ShNew` to create and initialize unbound service handles and `RequestService` to make bound service handles. To support the call-by-value-result parameter-passing paradigm, the argument buffer of a bound service handle must be initialized by `ShClean` for each service access request.

The Cygnus library also contains a routine called `service_errno`, which the client may call to get an informative error code when a service acquisition operation could not be executed successfully. The routine `service_errno_set` permits the client code to save user-defined error codes into the service handles. To facilitate the implementation of the library by using OS- and/or language-supported lightweight processes, the error codes are not provided as global variables and cannot be accessed directly from within the client code.

The library contains two routines to reset and shut down the service acquisition run-time support: `Service Runtime_start` and `Service Runtime_stop`. These two operations are provided mainly to permit the client code to reclaim resources (like file descriptors) held by the Cygnus run-time.

```
#include "my_header_files"
main(argc, argv) int argc; char *argv[];
{
  initialize_program_variables(argc, argv);
  establish_link_to_service_requester();
  establish_link_to_server();

  /* process service access requests */
  for (;;){
    wait_for_access_request();
    if (terminate()) break;
    extract_access_arguments();
    invoke_server_operations();
    compose_return_message();
    send_execution_result_to_service_requester();
  }
  housekeeping_routines();
}
```

Fig. 4. Skeleton of a Server Access Agent coded in C.

C. The Server Access Agent

Pursuant to the working principles of the Cygnus service acquisition mechanism described in Section III-A, Server Access Agents 1) convert service operations into server access requests, 2) accommodate server protocol heterogeneity, and 3) implement server-dependent error recovery algorithms. This section elaborates on how they work and how they can be implemented easily based on the programming structure sketched in Fig. 4. Performance overhead imposed by the Server Access Agents is analyzed in Section VI-A.

Service Request Phase: During the service request phase, a Server Access Agent normally first initializes itself in accordance with a set of configuration parameters (e.g., the network and transport address of a server interface) set by the service specification interpreter that activates it.

The Server Access Agent then establishes a link to the service requester. The requester is normally the Service Access Monitor, but may be a Cygnus client when the fault-tolerance support provided by the monitor either is not desired by the client or is not applicable to the desired service. From the viewpoint of the requester, the Server Access Agent is a local server[5] that speaks the Cygnus internal IPC protocol.

The Server Access Agent may also try to establish a link to the associated server during the service request phase. If the link cannot be set up successfully, the Server Access Agent shuts down its link to the requester and terminates itself. If the requester is not a Cygnus client, the Service Access Monitor either contacts the Service Specification Interpreter for a new Server Access Agent or returns an error message to the client.

Service Access Phase: A Server Access Agent falls into a loop during the service access phase. The code segment that invokes server operations is usually a multiway branch statement on the names of the supported service operation.

[5] According to the Cygnus model (see Section II), a Server Access Agent that integrates the functions of several servers or need not be bound to a server to support the desired service operations should be identified as a server and should not be considered as a component of the service acquisition mechanism.

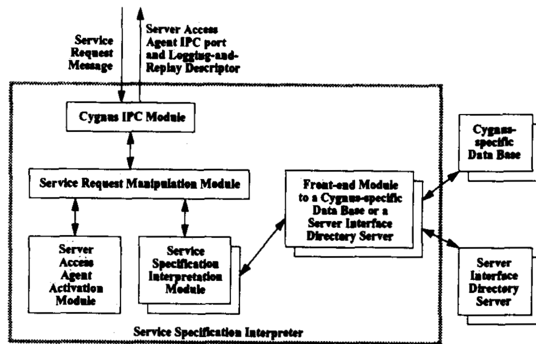


Fig. 5. Service Specification Interpreter structure.

For each operation, the agent first extracts the input-only tagged arguments from the service access request message, and transforms those arguments into a form that the server expects. The service operation is then implemented by invoking one or more server interface operations. Finally, the execution results are converted into a Cygnus IPC message and sent back to the requester.

Fault-Tolerance Support: The Server Access Agent may also support various server-dependent fault-tolerance mechanisms. This resilience support complements the server-independent fault-tolerance support provided by the Service Access Monitor, especially when the associated server can be shared by other processes to manipulate common data objects. For example, an optimistic message-logging and process-checkpointing mechanism can be used by the Server Access Agents and Servers to make the services in use resilient to machine crashes [13].

D. The Service Specification Interpreter

The Service Specification Interpreters assume the tasks of 1) analyzing service specifications in client-specific contexts, 2) selecting appropriate servers for use by the clients based on the results of the analysis, and 3) activating Server Access Agents for the servers that do not support the specified service access operations directly using the Cygnus IPC mechanism.

Since different interpretation contexts may require different interpreters and access to different information management software (such as personal profile managers, user-location servers [27], and server interface directory servers), a Service Specification Interpreter must be coded as a collection of extensible, cooperative computing entities to ensure its quality. Thus, instead of presenting the implementation details of the Service Specification Interpreters, this section focuses on how the Interpreters are structured to perform the tasks well and to meet the extensibility requirement.

Fig. 5 shows that the Service Specification Interpreter is composed of a Cygnus IPC module, a Service Request Manipulation Module, a Server Access Agent Activation Module, several Service Specification Interpretation Modules, and various front-end modules to the databases or directory servers in the system.

The Service Request Manipulation Module interacts with the service requester through the Cygnus IPC Module, forwards service specifications to Service Specification Interpretation Modules, and controls the activation of Server Access Agents. The interpretation module for a service specification is now chosen based on the value of the standard attribute CONTEXT (see Section II-A). The module keeps its link to the service requester active during the service access phase if the requested service can be supported by several server interfaces.[6]

Each Service Specification Interpretation Module analyzes the given service specification in a certain context. There are no restrictions on what resources (e.g., databases) can be used by a Service Specification Interpretation Module, nor are there any restrictions on how the resources can be used. As an example, to analyze the personal messaging service specification given in Section II-A, the module may access a user-location server and a cross-domain directory system to find out what communicators Allen may use at his current location [7]. The location server and the directory system may not be required by other interpretation modules.

After a Service Specification Interpretation Module analyzes a service specification, it returns to the Service Request Manipulation Module the file path name of a Server Access Agent program, which will interact with a server providing the server capability desired by the client. It also returns a list of arguments that could be passed to the Server Access Agent as command line arguments and be used to initialize the agent. Finally, it returns a logging-and-replay descriptor that could be used by the Service Access Monitor to correctly reconfigure service-server bindings should the Server Access Agent become inaccessible unexpectedly. (See Section III-E.)

Server Access Agent programs could either be maintained in a database or be generated on demand via a program synthesizer [12]. When the Service Specification Interpretation Module does not establish a link to the server on behalf of the Server Access Agent, it provides the agent with sufficient information on the server interface, depending on the Server Interface Directory Servers that the agent may use. (See Section III-C.)

The Server Access Agent Activation Module creates new Server Access Agents upon request by using the `fork` and `execl` system calls. It also reclaims the resources held by the agents when they terminate by the `signal` and `wait3` system calls.

E. The Service Access Monitor

The Service Access Monitor uses a logging and replay mechanism to make its link to the client resilient to server access failure. (See Section III-A.) This fault-tolerance mechanism is server-independent, because the monitor records service access requests and results in service-specific format. It records neither server interface operations nor server execution states. The applicability of this mechanism depends on the service in use, because not all server access failures can be recovered by simply replaying the access requests made by the client. (See Section III-A.)

To guide the monitor in performing these logging and replay operations correctly and efficiently, Service Specification

[6]In accordance with the Cygnus model, each server interface belongs to only one server. (See Section II.)

```
01 #include <stdio.h>
02 #include <cygnus/cygnus.h>
03 #define S_to_cS(x) (x)
04 main()
05 {
06   ServiceHandle sh;
07   if (ServiceRuntime_start() < 0) exit(1);
08   if (ShNew(&sh) < 0) exit(1);
09   ArgIn_cString(sh, "CONTEXT", S_to_cS("display"));
10   RequestService(sh);
11   if (service_errno(sh) != 0) exit(1);
12   ShClean(sh);
13   ArgIn_cString(sh, "MSG", S_to_cS("hello"));
14   ArgIn_Op(sh, "display");
15   if (AccessService(sh) < 0) exit(1);
16   TerminateService(&sh);
17   ServiceRuntime_stop();
18 }  /* main() */
```

Fig. 6. A simple Cygnus client.

Interpreters always provide a logging-and-replay descriptor for each service request message. A logging-and-replay descriptor presently contains a *log code* and an optional set of *replay records*. The log code indicates what logging and replay scheme is required. Each replay record includes a service operation name and a replay code that indicates how to replay the service operation. When the log code is `NoLog`, the monitor lets the client communicate with the Server Access Agent directly.

The Service Access Monitor interprets the value in the replay record only when it is instructed to do selective logging and replay. A service access request is not logged when the replay code is `NoReplay`. If the replay code equals `ReplaySend`, execution results of an access request are not logged. Both the request and result messages are logged when the replay code is `ReplayAll`.

During the service reconfiguration phase, the monitor replays the logged service access requests. To ensure transparent recovery, it also compares the values returned by the new Server Access Agent with those in the log. A match indicates functional equivalency between the old and new servers over the period that the client has accessed the service. The monitor, however, validates the new execution results only for the operations whose replay codes are `ReplayAll`. The server identities returned by the Server Access Agents are not compared at all.

The monitor always asks for a new Server Access Agent when it finds the current one unavailable during the service access or reconfiguration phase. It also does so when logged service access requests cannot be replayed correctly during the service reconfiguration phase. The number of attempts to recover from a server access failure is currently bounded by a configuration parameter set by the Service Acquisition Manager.

The performance overhead imposed by the Service Access Monitor is analyzed in Section VI. Our experience with the server-independent service-server reconfiguration mechanism is given in Section VIII.

IV. An Example

Fig. 6 shows a simple C program that sends the string "hello" to a display server. The server returns an acknowledgment message after displaying the string on its output device. The numbers along the left margin are provided for ease of reference and are not part of the code.

The program starts its execution at line 7, which initializes the Cygnus service acquisition run-time. At line 8, the client code initializes an unbound service handle. It then saves attribute `CONTEXT` with value `"display"` into that service handle at line 9 and proceeds to request the service at line 10. If the service requested can be honored, the `RequestService` routine stores necessary information about the allocated Service Access Monitor or Server Access Agent into the unbound service handle; otherwise, it sets an error code in the service handle.

Lines 12–15 show how to invoke a Cygnus service operation. At line 12, the client first cleans up the bound service handle. It then stores the input-only keyword argument `MSG` with value `"hello"` and the service operation name `display` into the service handle at lines 13 and 14. The service access primitive `AccessService` at line 15 returns a nonnegative number if the request can be processed successfully.

Lines 16–17 cancel the requested service and terminate the Cygnus run-time, respectively. These two statements are not mandatory, though they are recommended.

V. Programming Language Support for the Mechanism

There are two issues that arise when using Cygnus in this fashion. First, a user may not know exactly which keywords to use in specifying a desired service. Such keywords may even be system- or domain-dependent. Also, since we expect new services to be created regularly, it is inappropriate to require that users be sufficiently aware of their details to be able to infer specification keywords.

Second, the simple example given in Fig. 6 shows that it can be error-prone in invoking Cygnus run-time library routines directly. First, service specification attributes and service operation arguments are expressed in two types of systems: data types supported by C and the Cygnus IPC facility. (see Section III-B.) Second, the type signatures of service operations can be obscured. Finally, the programmer must understand the Cygnus model well and be familiar with the Cygnus run-time library, because each invocation of a service operation usually takes several statements.

We have therefore developed a language veneer over the C programming language that supports the Cygnus computational model. The extended language, called CygnusC, allows programmers to view the client-service bindings as references to ADT objects that are instantiated at run-time via special templates. These templates permit the compiler to type-check service acquisition operations and enable the language run-time to invoke user code for validating service specifications and handling service access failures. The ADT object instantiating and disposal operations in the language establish and release client-service bindings. Each operation performed on the objects is mapped transparently to one or more remote server invocations.

The CygnusC compiler ensures that the service operations are invoked as they are declared, but does not insert code for run-time type-checking. It relies on the service acquisition system to detect run-time type errors, because the clients can compose service specifications at run-time and because different services may be associated with different sets of operations.

CygnusC is carefully designed so that compilers for it can be implemented easily via native C compilers. For example, on our client machine, the compiler is implemented by pipelining the native C preprocessor `cpp` , a CygnusC precompiler, and the native C compiler `compile`. The precompiler [4] transforms CygnusC constructs into C statements with no macros. A big advantage of this approach is that it permits the programmers to develop CygnusC software by using existing C libraries and programming tools.

We note that one of the CygnusC design goals is to experiment with a few language constructs that facilitate the development of client applications in server-based heterogeneous distributed systems. These constructs may also be established on top of languages other than C, in the manner of Linda [9] or Concert [29]. Thus, CygnusC must not be viewed as a full distributed programming language, but as an instance of our language veneer resulting in extensions to C. As such, CygnusC lacks many of the features and trappings of a full distributed programming language.

In the remainder of this section, we first present a simple CygnusC program to outline the extensions that we have made to the C programming language. We then give the rationale for these extensions. The C code generated by the CygnusC precompiler for each of the service acquisition constructs is shown to 1) explain the semantics of the new language constructs, 2) exhibit the C features used by the precompiler, and 3) illustrate a good way of structuring Cygnus library routines.

A. A Message Dispatcher Coded in CygnusC

Fig. 7 shows a simple CygnusC program that calls the attention of the Department Computing Organization (DCO) to problems with the computing environment. Only one response from the DCO staff is required for each execution of the program. The staff member who acknowledges a message may be contacted by a server through a pager, by electronic mail, or by another device. Before terminating its execution successfully, this program prints an acknowledgment code and an identification string for the notification server used on a default output device. The client need not know beforehand how the dispatched message will be sent to the DCO staff or how the message will be acknowledged.

This sample program is organized as follows. Lines 1–3 import the interfaces from the required run-time libraries. Lines 4–19 declare a *service specification template* with name `spec` to define a family of bound service handles (see Section III-B) for accessing Cygnus services. The `Attribute` component (lines 6–8) of this template specifies the set of attributes associated with those service handles. The `Constraint` component (lines 9–13) contains user code for validating the service specifications composed at run-time. The `Operation` component (lines 14–15) declares the type signatures of the required service access operations so that the compiler can ensure that they are invoked as declared. The `OnError` component (lines 16–19) includes code for handling service access errors at run-time. These components are dealt with in detail in Section V-B.

```
01 #include <stdio.h>
02 #include <cygnus/cygnus.h>
03 #include "userlib.h"
04   ServiceHandle spec  /* service specification template */
05   {
06     Attribute  /* specify attribute names and their types */
07       CONTEXT(cString)String="notify";
08       LIST(cString)String="dco";
09     Constraint /* user code for validating specification */
10       if ((0!=strcmp(CONTEXT,"sos"))&&(0!=strcmp(CONTEXT,"notify"))){
11         printf("Invalid attribute CONTEXT: %s\n",CONTEXT);
12         return(1);
13       }
14     Operation  /* specify service operations */
15       dispatch[MSG(cString)String]->[ACK(cInt)int,SERVER(cString)String];
16     OnError  /* error handler for service access operations */
17       printf("access errno = %d\n",service_errno(spec));
18       exit(1);
19   }  /* spec */
20 main(argc, argv) int argc; char *argv[];
21 {
22     char *id; int ack; ServiceHandle spec sh;
23   if (ServiceRuntime_start()<0) exit(1);
24       if (argc<3) sh=ServiceHandle spec[];
25         else sh=ServiceHandle spec[CONTEXT=argv[2]];
26       if (service_errno(sh)!=0) exit(1);
27   [ack=ACK,id=SERVER]=sh:dispatch[MSG=argv[1]];
28   printf("(Server ID, Acknowledge code) = (%s, %d)\n",id,ack);
29       TerminateService(&sh);
30   ServiceRuntime_stop();
31 }  /* main() */
```

Fig. 7. A message dispatcher coded in CygnusC.

This program starts its execution from the statement at line 23, which initializes the Cygnus library routines. The service request statement at line 24 is executed with the default service attribute values defined in the `Attribute` part of `spec` when the value of attribute `CONTEXT` is unavailable as an input argument. The service handle returned by this invocation is saved in the service handle variable `sh` declared at line 22. Arguments for the request statement at line 25 are DCO for attribute `LIST`, and the second command line input argument for `CONTEXT`.

The service operation `dispatch` is invoked at line 27 if the service handle returned was successfully bound to the requested service. In accordance with our call-by-value-result service invocation paradigm, the input-only keyword argument `MSG` is first set to the message to be shipped out. The `AccessService` routine in the Cygnus library is then invoked with service handle `sh`, operation name `dispatch`, and an input argument with tag `MSG`. When the invocation completes, the value of output-only keyword argument `ACK` is assigned to the integer variable `ack`, and the value of `SERVER` to pointer variable `id`.

The `TerminateService` operation at line 29 is invoked to reclaim the resources allocated to service handle `sh`. Necessary housekeeping tasks would also be performed on the associated server host. Finally, the client shuts down the Cygnus service run-time gracefully at line 30. These last two statements are not mandatory, but recommended.

B. The Service Specification Template

From the viewpoint of the client code, a service specification template defines a set of ADT objects for accessing remote

services. Executing a service request statement instantiates an ADT object for a specific service at run-time. Invocations of the associated service operations are similar to those on an ADT object: The caller knows only the type signature of the invoked operation, and is ignorant of the implementation details. In the remainder of this section, we present the design rationale of this template.

Attribute: The CygnusC data type[7] and Cygnus IPC data type for every attribute must be declared in the `Attribute` component to enable the compiler to type-check the operations on attribute values and determine necessary type conversion routines between those two types.

The Cygnus run-time library contains type conversion routines between primitive CygnusC (or C) and Cygnus IPC data types. For example, two routines `int_to_cInt` and `cInt_to_int` are incorporated in the library to convert C `int` values to Cygnus `cInt` values, and vice versa. When the language data type of a service attribute is a composite or user-defined data type, the programmer must provide the required conversion routines for that data type. As Sun XDR library routines are to the user-provided XDR routines, so are the Cygnus library routines to the user-provided type conversion routines.

`Constraint`*:* The `Constraint` component is used for screening service specifications. The syntax of its body is the same as that of a C function without enclosing braces. The formal parameters of this function-like component are the attributes declared in the `Attribute` component. It must be written to return a nonzero integer value as a user-defined service request error code when the service specification given fails to pass the test.

We expect the `Constraint` component to be very useful in large heterogeneous systems, where the number of servers tends to be large. In such an environment, the cost of invoking a service request operation increases with the cost of locating a server interface.

Fig. 8 shows how our CygnusC precompiler translates the `Constraint` component of `dco.ccc` into C code. The declaration of the specification checking function shown in Fig. 8(b) is emitted at the point where the service specification template is declared in the source code. This function is named by appending "`__constraint`" to the name of the associated service specification template, i.e., "`spec`." Two underline characters are used to reduce the chance of redefining an existing function. Statements in Fig. 8(c) constitute the function's definition, and are emitted by the precompiler only if the program starting routine `main()` is defined in the source file.

`Operation`*:* The programmer must define the type signatures of service access operations in the `Operation` component to allow compile-time type checks on service operation invocations. For example, line 15 in Fig. 7 declares that operation `dispatch` needs the input argument `MSG` and two output arguments: `ACK` and `SERVER`. The Cygnus IPC data type of `MSG` is `cString`, and the language data type is `String`. Similarly, the Cygnus IPC data type of `ACK` is `cInt` and language data type `int`. When the compiler encounters the service access statement at line 27, it validates the number of arguments and the related assignments. For example, if variable `ack` is declared as a pointer to an integer, the compiler would issue a type error message at this line.

[7] The CygnusC and C programming languages share a common type system.

```
09 Constraint /* user code for validating specification */
10   if ((0!=strcmp(CONTEXT,"sos"))&&(0!=strcmp(CONTEXT,"notify"))){
11     printf("Invalid attribute CONTEXT: %s\n",CONTEXT);
12     return(1);
13   }
```

(a)

```
01 int spec__constraint();
```

(b)

```
01 int spec__constraint(CONTEXT, LIST) String CONTEXT; String LIST;
02 {
03   if ((0!=strcmp(CONTEXT,"sos"))&&(0!=strcmp(CONTEXT,"notify"))){
04     printf("Invalid attribute CONTEXT: %s\n",CONTEXT);
05     return(1);
06   }
07 }
```

(c)

Fig. 8. Precompiler output for the `Constraint` component.

```
16 OnError
17   printf("access errno = %d\n",service_errno(spec));
18   exit(1);
```

(a)

```
01 void spec__onerror();
```

(b)

```
01 void spec__onerror(spec) ServiceHandle spec;
02 {
03   printf("access errno = %d\n",service_errno(spec));
04   exit(1);
05 }
```

(c)

Fig. 9. Precompiler output for the `OnError` component.

The input and output arguments of each service access operation are explicitly identified, because their positions are not linked to the properties of the desired server capabilities. The service acquisition run-time is in charge of marshaling the keyword arguments for the servers in use.

`OnError`*:* The syntax of the `OnError` component is the same as that of the `Constraint` component. The `OnError` component, however, is used to include user code for handling service access errors. When a service operation invocation fails, the service acquisition run-time sets an error code in the service handle and passes control to this component via the language's procedure call mechanism. This component can then remedy the error based on the error code or abort the program's execution as it does in `dco.ccc`. When this component returns, control is passed back to the statement following the service access statement being executed.

Fig. 9 shows the precompiler output for this component in `dco.ccc`. The declaration of function `spec__onerror` is emitted immediately after the service specification template `spec` is completely parsed. The precompiler does not emit this function's definition unless the starting routine `main()` is defined in the source file.

```
24 if (argc<3) sh=ServiceHandle spec[];
25    else sh=ServiceHandle spec[CONTEXT=argv[2]];
```

(a)

```
01 if (argc<3) sh=spec__init("notify","dco");
02    else sh=spec__init(argv[2],"dco");
```

(b)

```
01 ServiceHandle spec__init();
```

(c)

```
01 ServiceHandle spec__init(CONTEXT,LIST) String CONTEXT; String LIST;
02 {
03    int ret; ServiceHandle sh;
04    if (ShNew(&sh)<0) exit(1);
05    if ((ret = spec__constraint(CONTEXT,LIST))==0){
06      ArgIn_cString(sh,"CONTEXT",String_to_cString(CONTEXT));
07      ArgIn_cString(sh,"LIST",String_to_cString(LIST));
08      RequestService(sh);
09    } else service_errno_set(sh,ret);
10    return(sh);
11 }
```

(d)

Fig. 10. Precompiler output for service request statements.

```
27 [ack=ACK,id=SERVER]=sh:dispatch[MSG=argv[1]];
```

(a)

```
01 {ServiceHandle  tmp__sh;
02    tmp__sh=sh;
03    ShClean(tmp__sh);
04    ArgIn_cString(tmp__sh, "MSG", String_to_cString(argv[1]));
05    ArgIn_Key(tmp__sh, "ACK");
06    ArgIn_Key(tmp__sh, "SERVER");
07    ArgIn_Op(tmp__sh, "dispatch");
08    if (AccessService(tmp__sh)<0){
09      spec__onerror(tmp__sh);
10    } else {
11      ack = cInt_to_Int(ArgOut_cInt(tmp__sh,"ACK"));
12      id = cString_to_string(ArgOut_cString(tmp__sh,"SERVER"));
13    }
14 }
```

(b)

Fig. 11. Precompiler output for service access statements.

C. Service Request Statement

A service request statement is executed as follows. The service attributes are first initialized to the defaults declared in the associated service specification template. These defaults may be overridden by new pairs of attribute name and value in the argument list of the service request statement. This specification is then validated by the constraint code defined in the associated `Constraint` component. Finally, the service acquisition run-time returns a bound service handle after establishing a client-service link. An unbound service handle is returned with an error code if the link could not be set up successfully. This error code could be set by the service acquisition run-time or by the user-defined constraint code. The service handle returned always appears to the client code as a pointer to an opaque data structure. In Fig. 7, service handle variable `sh` is declared at line 22, and is initialized at line 24 or 25.

Fig. 10 shows how the precompiler translates such service request statements into C statements. Fig. 10(a) lists the `if` statement that contains two service request statements in our sample CygnusC program. The C code shown in Fig. 10(b) is emitted when the precompiler encounters the `if` statement. The associated specification validation function is named by appending "`__init`" to the name of the associated service specification template. A declaration for this function is emitted when the specification template has been completely parsed. To prevent duplicate definitions, this function will not be defined in the precompiler output file if the routine `main()` is undefined.

Fig. 10(d) shows that the service specification screening routine generated by the precompiler always initializes an unbound service handle first by invoking routine `ShNew` provided by the Cygnus run-time library. (See Section III-B.) If the set of attributes provided fails to satisfy the constraints given in the `Constraint` component, a user-defined error code is stored into the unbound service handle via the `service_errno_set` routine in the Cygnus library.

After the validated attributes are copied into the unbound service handle, routine `RequestService` is invoked to bind the service handle with the requested service. If the binding process fails, the service handle remains unbound and holds an error code set by the service acquisition run-time.

D. Service Access Statement

From the viewpoint of the client code, a service access statement is a function invocation with a bound service handle, a service operation name, and a set of input-only keyword arguments as parameters. The service handle argument can be specified by a simple variable, such as `sh`, in the sample program, or by a more complicated postfix expression such as `ShArray[i++]`. The output values are assigned to a set of tagged address references under our call-by-value-result parameter-passing paradigm.

Fig. 11 shows how a CygnusC service access statement is translated into C code by our precompiler. The service handle expression in the statement, i.e., `sh`, is first evaluated and assigned to a temporary service handle variable (see Fig. 11(b)) to ensure that the expression will be executed only once in the generated C code. After the service handle is initialized, input arguments to the service access primitive `AccessService` are copied into the handle through various Cygnus library routines. If `AccessService` fails, the service acquisition subsystem sets an error code and passes control to the associated `OnError` component as a C procedure call. If the access invocation succeeds, the output values are extracted and copied into the corresponding memory locations. In order to reduce the overhead in extracting the invocation results, the output argument labels are made available to the service acquisition run-time via `ArgIn_Key`. The overhead is presently a linear function of the number of the tagged values returned because a linear search algorithm is used in the Cygnus library to locate the value associated with a specific tag.

VI. Performance

We developed a Sun RPC client-server pair to estimate the performance overhead that Cygnus clients may incur in using the service acquisition mechanism to access local or remote

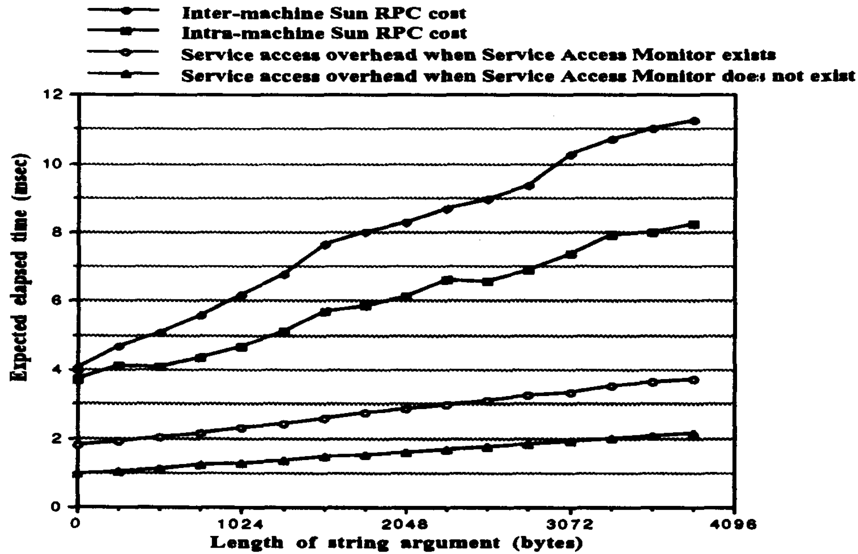


Fig. 12. Performance of Cygnus service access mechanism.

servers. The client stub, server stub, the main program of the server, and the required C header files were generated by Sun's `rpcgen`. The RPC Language (RPCL) code specifies only one `void` function with one input argument of type `string`. The server implements that function with a dummy routine.

We measured the cost of a Sun RPC call as the expected elapsed time in executing the `clnt_call` statement in the `rpcgen`-emitted client stub. With reference to Fig. 6, it is the expected elapsed time in executing the statements at lines 12–15. Thus, the overhead is the difference between these two times. The computing environment was under very light load conditions when the performance data were collected. For intermachine calls, the server ran on another Sun 4/60 workstation sitting on the same ethernet and with the same configuration as the client host. We view the performance of intramachine Sun RPC as a baseline data for the cost of invoking server operations.

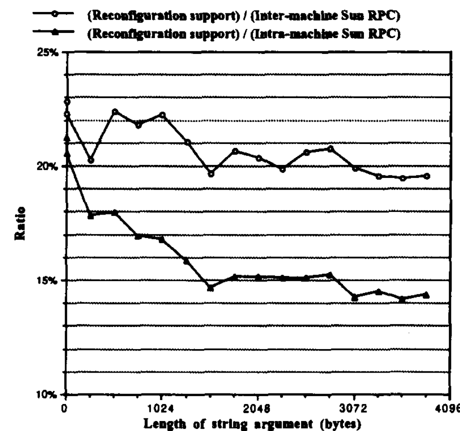


Fig. 13. Cygnus reconfiguration support overhead.

A. The Service Access Mechanism

Fig. 12 shows that the overhead is small in absolute terms and acceptable in relative terms. When the message length is less than 512 bytes, the Cygnus IPC cost is less than 2 ms when the Service Access Monitor runs between the client and Server Access Agent, and less than 1.2 ms when the monitor does not exist. These numbers are independent of the network load, but depend on the processor load on the client machine. The overhead is small because under normal load conditions, it usually takes tens to hundreds of milliseconds to send a 1024-byte string remotely via Sun RPC.

We are satisfied with the performance of the current Cygnus service access mechanism, because the mechanism is currently implemented by heavyweight processes. We have observed that process scheduling delay is the bottleneck in our present implementation of the Cygnus service access mechanism. We estimate that scheduling delay contributes more than 97% of the Cygnus IPC cost for a one-byte message, and more than 30% for a 3840-byte message. Moreover, the percentages increase with the system's load.

B. The Reconfiguration Support

Fig. 13 depicts the reconfiguration support overhead in comparison with the local and remote Sun RPC costs. The reconfiguration support cost is the Cygnus IPC cost when the Service Access Monitor exists, minus the IPC cost when the logger does not exist. (See Fig. 12.) For both cases, the ratios never exceed 25%, and tend to decline when the message size increases.

The logging and replay mechanism could be implemented very efficiently for three reasons. First, the logs were not

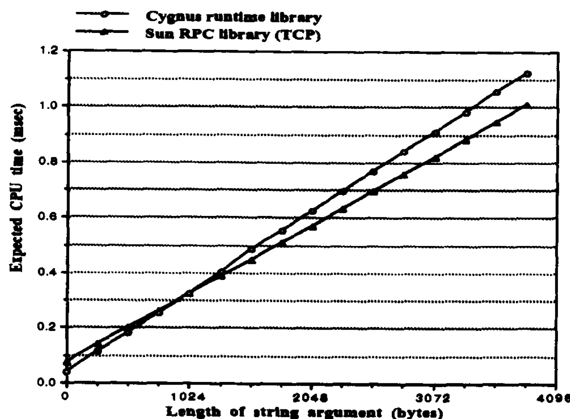


Fig. 14. Performance of Cygnus run-time library.

stored on stable storage, because crash recovery support for the client was not available. Second, the service-server reconfiguration mechanism requires that no more than one server be available at any one time. Unlike other replication-based fault-tolerance mechanisms, like that in ISIS [2], this reconfiguration mechanism does not incur the overhead of synchronizing the executions of a group of functionally identical servers running on different hosts. Finally, since the logging and replay algorithm is applied on the basis of nonshared client-service bindings, it is far less complicated than those used in transaction-based systems such as Quicksilver [10]. The corresponding mechanisms in those systems are designed to optimize the throughput of updating shared persistent objects, and must be coupled with checkpointing and rollback mechanisms.

C. The Cygnus Run-Time Library

We have also compared the performance of the Cygnus run-time library with the Sun RPC library. The performance of the Sun RPC library was measured as the expected CPU time spent in executing the `clnt_call` statement with no BSD socket invocations. The FIFO system calls were commented out when we measured the performance of the Cygnus run-time library. It turns out that the Cygnus run-time library even consumes less CPU cycles when the size of the string argument is less than 1024 bytes as depicted in Fig. 14. Fig. 14 shows that the slope of the curve for the Cygnus run-time library is about 0.28 ms per byte, and that the Sun RPC library 0.24 ms per byte. Since these two libraries use the same routines to move string arguments into message buffers, we think the difference is caused by the system overhead for shared memory support. In any event, it shows that the cost of processing IPC messages is much less than the cost of making interprocess invocations.

VII. Other Typical Applications of the Mechanism

To date, the service acquisition mechanism has been implemented on Sun 3 workstations running Mach or SunOS 4.x, Sun 4 workstations running SunOS 4.x, IBM RT running BSD 4.3, and IBM PS/2 running OS/2 1.2. Besides the examples given in the previous sections (i.e., the personal messaging service in Section II-A, the text display service in Section IV, and the message dispatcher in Section V-A), three other very different distributed applications have also been developed to investigate the usefulness of the mechanism: a dictionary service, a snake game service, and a computational vision service.

The dictionary service enables the clients to look up words in *Webster's* dictionary (7th ed.) with no need to handle server access failures. The servers accessed may run on other Internet nodes over which we have no control. The source code for a client program named `webster.c` is available in the public domain. The Cygnus client contains mainly the user-interface code in `webster.c`, and the Server Access Agent incorporates the code for accessing the server.

The computer game `snake` is a display-based chase game, and was written as monolithic software to help people familiarize themselves with text editor `vi`. To develop a distributed version of this game, we first split the original source code into two parts: one contains mainly user-interface routines, and the other implements the rules of the game. The service operations are defined in light of the interactions between these two modules. The snake client contains the user-interface routines and invokes the service operations through CygnusC service acquisition facilities. Two snake servers with different interfaces were developed: one of them exports its interface through Sun RPC/XDR protocol, and the other uses the NCS RPC/NDR protocol. Two corresponding Server Access Agent programs also exist. The only differences between those two agent programs are that they use different binding protocols to establish links to the servers and convert service operations to different remote calls. Their interfaces to Cygnus clients are the same. The logging-and-replay descriptor for this service instructs the logger not to replay the operations for refreshing the screen.

The computational vision service is designed to analyze two-dimensional or three-dimensional images by using the generate-and-test (or hypothesize-and-test) approach. The service operations are presented to the client code as a set of re-entrant library routines. The computations, however, are performed by remote servers that may run on Sun 4 workstations, Apollo DN4000 workstations, or Alliant FX-8 compute servers. To avoid running a specialized daemon process on each of the server machines for this service, the servers are created by the Server Access Agents dynamically via BSD UNIX `rexec` system call. Only one Server Access Agent program was written for this application. The Service Access Monitor replays only the last incomplete service access request when a server access fails. The service access requests and associated execution results are not logged, because each service operation represents an atomic computation. There is no state information needed to be restored for a broken service-server link.

To the best of our knowledge, none of the existing distributed systems provides a comparably uniform approach to supporting these services for the client applications. For example, most of the contemporary distributed systems have

problems in supporting fault tolerance and in accommodating server protocol heterogeneity. Specialized service acquisition systems like Marionette [23] and RM [24] cannot enhance the resilience of service access links as our service acquisition mechanism does. Language-based distributed systems like Argus [15], Emerald, and DEC HDS requires an instance of the language run-time to run on each of the participating hosts, whereas the Cygnus service acquisition system needs to be installed only on the client hosts. Command-interpreter-based approaches like the Profile shell [21] and Wills shell [28] have difficulty in providing application routines with services like our computational vision service.

VIII. Experience and Lessons Learned

Below we outline some of our experiences in implementing and using the service acquisition mechanism, the CygnusC programming language, and the applications.

- **Generality is useful, but domain structure must be used to regulate generality.** This work was motivated by the issues arising from the increasing prevalence of independently developed network servers, and the resulting possibilities of making a wide variety of services available to clients. The client-server model offers a clean and simple framework for addressing these issues. The Cygnus service request primitive is general enough to enable the client to specify whatever server capabilities and service access interfaces it desires. The Cygnus service access primitive can be used to access many network services under an RPC paradigm.

However, we found unmitigated generality to be a hindrance to disturbed computing. In their raw form, the Cygnus primitives can be very general and powerful, but they are not always useful to application creators. Programmers often just need to know what service specifications a client is allowed to compose, and what service access operations a certain service specification defines in some programming language. This experience lead us to the work on programming language support for the Cygnus service acquisition mechanism (see Section V).

Similarly, at one point, we thought that we needed a Smalltalk-like object-oriented model to address the issues raised in Section I-A, but soon realized that such a model was far too general to provide a useful perspective on the issues. It became apparent that the use of a computational model should be justified only in the context of a well-defined problem domain. Generality does not automatically help at all levels.

- **Descriptive naming is a good approach to realizing the client-service model when used properly**. Our experience with the applications that we have built verifies that expressing server capabilities via name-value pairs is a good approach to realizing the client-service model (see Section II-A) when the flexibility can be controlled well. Completely unstructured descriptive names can be messy, and it is important to introduce rules or conventions to restrict the mechanism's power. For example, a Cygnus service specification is now analyzed by first extracting the CONTEXT field from the specification to determine which routine to invoke to analyze the remaining name-value pairs. This scheme can be used recursively; successive routines may analyze remaining name-value pairs partially and pass the rest to other routines to finish the analysis. This service specification analysis scheme permits us to provide new services for the client by introducing new service attributes and new Service Specification Interpretation Modules (see Section III-D), without worrying about the possibility that a partial change in semantics would have global effects. It also facilitates easy modification of current service specifications and the maintenance of interpretation modules.

- **The server-independent service-server reconfiguration mechanism is useful, but may complicate the task of interpreting service specifications.** Although the implementation of the logging-and-replay mechanism used by the Service Access Monitor is not fancy, it performs well when the state of the client-service link can be restored by replaying the logged service access operations on the new Server Access Agent. When the Server Access Agents for a service support a common set of service state checkpointing and reset operations, the Service Access Monitor can reduce the reconfiguration cost further by intelligently recording the state from time to time transparently to the client. Thus, like table-driven syntax analyzers, the Service Access Monitor becomes a generic descriptor-driven fault-tolerance mechanism that can be used or shared by many applications.

We recognize that server-dependent fault-tolerance mechanisms are still necessary because the logging-and-replay mechanism is not applicable to all kinds of services (see Section III-A), especially when the server state is part of the client-service state and cannot be recorded by the Service Access Monitor. These mechanisms may also be preferred over the server-independent ones when they incur less performance overhead. Since the logging-and-replay descriptor and Server Access Agents for a service acquisition session are determined by a Service Specification Interpretation Module, the task of interpreting service specifications can be complicated when the choice between server-dependent and server-independent mechanisms is not straightforward.

- **Security issues in Cygnus must be addressed by mechanisms native to the client host.** We thought it was unnecessary to incorporate a security mechanism directly into our prototype implementation of the Service Acquisition System (see Fig. 3) because the security mechanisms local to the client should take precedence over any that the Service Acquisition System might use. The Cygnus Service Acquisition System must be modified when its execution could break some security rule enforced on the local host. Thus, it is important to realize that more efficient implementations of the Service Acquisition System must be done in conformance with the local trust model and security mechanisms.

As an example, implementing the Cygnus Service Acquisition System components by a dynamic linking facility might improve the performance of accessing remote servers in high trust systems, but might violate several local security rules in widely used multitasking computing environments.

On our client hosts, the Cygnus IPC mechanism is supposed to be secure. The Service Acquisition System is assumed to be trusted enough to access data and acquire server capabilities on behalf of the client. In order to support this trust model on our multiuser Unix workstations, the Service Acquisition System components belong to the super user `root`. They use the `setuid()` system call to change their real and effective ID when it needs to access personal files or to execute a program (which may be a shell script) on behalf of the client. From the viewpoint of the server, the Service Acquisition System is part of the client software. As an RPC, run-time is to the RPC client module, so is the Service Acquisition System to the client. Although this scheme is vulnerable to many kinds of threats, it seems to work pretty well in academic or industrial research computing environments.

IX. Conclusion

The increasing prevalence of networked servers has resulted in a great demand to increase the use of the servers and to reduce the development and maintenance cost of robust client applications. The Cygnus service acquisition mechanism provides the client a clean, simple view of server capabilities and a uniform, reliable service access interface that is independent of the interface exported by the ultimate service provider. The mechanism could be easily supported by C-like typed procedural languages. Several typical and useful applications have been developed to ensure the quality of the mechanism. Our work suggests that the mechanism facilitates the development, use, and maintenance of client and server software in large heterogeneous distributed systems comprising many autonomous servers. It also shows that the overhead of invoking remote server operations via the mechanism can be insignificant.

Acknowledgment

We are grateful to P. Honeyman for his numerous constructive suggestions on our work. We would like to thank D. Ballou, H. Bussy, S. Mohan, S. Sechrest, T. Teorey, S. Weinstein, R. Wolff, and our reviewers for their comments on earlier drafts of this paper. We appreciate the assistance provided by the Center for Information Technology Integration (CITI) and the EECS DCO staff at the University of Michigan, and the computer staff of the Information Networking Research Laboratory at Bellcore. Finally, we acknowledge the work done by other students in the Cygnus group: Nigel Hinds developed the snake servers on Sun 3, Sun 4, and HP Apollo workstations; Hsiu-ying Hsu ported our service acquisition mechanism on IBM PS/2 under OS/2 1.2; Yen-min Huang measured the performance of Sun RPC library and ported the service acquisition mechanism on Mach 2.5 machines; Shih-ping Liou developed the computational vision service; and Aruna Victor developed several servers and service agents for the *Webster's* and snake service by using Sun RPC and NCS RPC.

References

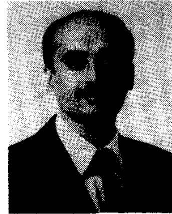
[1] B. Bershad, D. Ching, E. Lazowska, J. Sanislo, and M. Schwartz, "A remote procedure call facility for interconnecting heterogeneous computer systems," *IEEE Trans. Software Eng.*, vol. SE-13, no. 8, pp. 880–894 Aug. 1987.
[2] K. Birman and T. Joseph, "Reliable communication in an unreliable environment," *ACM Trans. Comput. Syst.* vol. 5, no. 1, pp. 47–76, Feb. 1987.
[3] A. Birrell and B. Nelson, "Implementing remote procedure calls," *ACM Trans. Comput. Syst.* , vol. 2, no. 1, pp. 39–59, Feb. 1984.
[4] R. Chang, "A network service acquisition mechanism for the client/service model," Ph.D. dissertation, Dept. of Electrical Eng. Comput. Sci., University of Michigan, 1990.
[5] R. Chang and C. Ravishankar, "Service acquisition mechanism for the client-service model in Cygnus," in *Proc. 11th Int. Conf. Distrib. Computing Syst.*, May 1991, pp. 90–97.
[6] R. Chang and S. Mohan, "Realizing the client-service model in the information networking (INA)," Tech. Memo. TM-ARH-021800, Bellcore, Oct. 1992.
[7] R. Chang, S. Mohan, and R. Wolff, "SISAS: A server-independent service acquisition system for distributed personal communications applications," in *Proc. Int. Conf. Communications ICC '93*, May 1993, pp. 307–312.
[8] J. Falcone, "A programmable interface language for heterogeneous distributed systems," *ACM Trans. Comput. Syst.*, vol. 5, no. 4, pp. 80–112, Nov. 1987.
[9] D. Gelernter, "Generative communication in Linda," *ACM Trans. Programming Languages Syst.* vol. 7, no. 1, pp. 80–112, Jan. 1985.
[10] R. Haskin, Y. Malachi, W. Sawdon, and G. Chan, "Recovery management in QuickSilver," *ACM Trans. Comput. Syst.*, vol. 6, pp. 82–108, Feb. 1988.
[11] A. Herbert, "The computational projection of ANSA," in *Distributed Systems*, S. Mullender, Ed. Reading, MA: Addison-Wesley, 1989.
[12] Y.-M. Huang and C. Ravishankar, "Accommodating RPC heterogeneities using automatic agent synthesis," Tech. Rep. CSE-TR-131–92, Dept. Electrical Eng. and Comput. Sci., Univ. of Michigan, 1992.
[13] D. Johnson and W. Zwaenepoel, "Recovery in distributed systems using optimistic message logging and checkpointing," *ACM Symp. Principles Distrib. Computing*, 1988, pp. 171–181.
[14] B. W. Lampson, "Designing a global name service," in *ACM 5th Symp. Principles Distrib. Computing*, 1986, pp. 1–10.
[15] B. Liskov and R. Scheiffer, "Guardians and actions: Linguistic support for robust, distributed programs," *ACM Trans. Programming Languages Syst.*, vol. 5, no. 3, July 1983, pp. 381–404.
[16] S. Madnick and J. Donovan, *Operating Systems*. New York: McGraw-Hill, 1974
[17] S. Mullender, Ed., *Distributed Systems*. Reading, MA: Addison-Wesley, 1989.
[18] G. Neufeld, "Descriptive names in X.500," in *Proc. SIGCOMM '89 Symp. Commun. Architectures Protocols*, 1989, pp. 64–71.
[19] D. Oppen and Y. Dalal, "The clearinghouse: A decentralized agent for locating named objects in a distributed environment," *ACM Trans. Office Inform. Syst.* , vol. 1, no. 3, pp. 230–253, July 1983.
[20] OSF, *Introduction to OSF DCE*. Englewood Cliffs, NJ: Prentice-Hall, 1992.
[21] L. Peterson, "The profile naming service," *ACM Trans. Comput. Syst.* vol. 6, pp. 341–364, Nov. 1988.
[22] A. Sinha, "Client-server computing: Current technology review," *Commun. ACM* vol. 35, pp. 77–98, July 1992.
[23] M. Sullivan and D. Anderson, "Marionette: A system for parallel distributed programming using a master/slave model," in *Proc. 9th Int. Conf. Distrib. Computing Syst.*, 1989, pp. 181–188.
[24] R. Summers, "A resource sharing system for personal computers in a LAN: Concepts, design, and experience," *IEEE Trans. Software Eng.* vol. SE-13, pp. 895–904, Aug. 1987.
[25] L. Svobodova, "File servers for network-based distributed systems," *ACM Computing Surveys* vol. 16, pp. 353–398, Dec. 1984.
[26] A. Tanenbaum and R. Renesse, "Distributed operating systems," *ACM Computing Surveys*, vol. 17, no. 4, pp. 419–470, Dec. 1985.
[27] R. Want, A. Hopper, V. Falcao, and J. Gibbons, "The active badge location system," *ACM Trans. Inform. Syst.* vol. 10, pp. 91–102, Jan. 1992.
[28] C. E. Wills, "A service execution mechanism for a distributed environment," in *Proc. 9th Int. Conf. Distrib. Computing Syst.*, 1989, pp. 326–334.

[29] S. A. Yemini, G. S. Goldszmidt, A. D. Stoyenko, Y.-H. Wei, and L. W. Beeck, "Concert: A high-level-language approach to heterogeneous distributed systems," in *Proc. 9th Int. Conf. on Distributed Computing Systems*, 1989, pp. 162–171.

R. N. Chang (S'86–M'90) received the B.S. degree in computer engineering with honors from the National Chiao Tung University, Taiwan, in 1982, and the M.S. and Ph.D. degrees in computer science and engineering from the University of Michigan, Ann Arbor, in 1989 and 1990, respectively.

He is currently Manager of Server Systems in the Internetworking and Multimedia Services Division of IBM. From 1990 to 1993, he was a member of the Technical Staff in the Applied Research Area of Bell Communications Research. From 1982 to 1984, he served in the Chinese Army Communications School, Taiwan, as a Lecturer in computer science. His research interests include multimedia communications and computing, personal mobile/nomadic computing, distributed systems, operating systems, and software engineering.

Dr. Chang is a member of the IEEE Computer Society and the IEEE Communications Society, the Association of Computing Machinery, the USENIX Association, Eta Kappa Nu, and Tau Beta Pi.



C. V. Ravishankar (S'82–M'86) received the B.Tech. degree in chemical engineering from the Indian Institute of Technology, Bombay, India, in 1975, and the M.S. and Ph.D. degrees in computer sciences from the University of Wisconsin, Madison, in 1986 and 1987, respectively.

He has been with the Department of Electrical Engineering and Computer Science, University of Michigan, Ann Arbor, since 1986. He is also a member of the Software Systems Research Laboratory and the Real-Time Computer Laboratory at the University of Michigan. His teaching and research at the University of Michigan have been in the areas of distributed systems and programming languages. His present research interests include large-scale distribution, heterogeneity, protocol synthesis, real-time systems, and database systems.

Dr. Ravishankar is a member of the IEEE Computer Society and the Association for Computing Machinery.