A Service Acquisition Mechanism for the Client/Service Model in Cygnus*

Rong N. Chang
Bell Communications Research
rong@thumper.bellcore.com

Chinya V. Ravishankar
University of Michigan—Ann Arbor
ravi@eecs.umich.edu

Abstract

Three of the most important issues in exploiting network servers concern (1) how to specify services so that service-server bindings can be changed dynamically without disturbing clients, (2) how to make clients resilient to network or server failure, and (3) how to accommodate server protocol heterogeneity to provide a single system view to the clients. This paper presents a service acquisition mechanism for solving these issues. This mechanism is designed under a *client/service* model in which the abstraction of service is a first-class entity. The mechanism comprises (1) a service request mechanism for establishing client-service (and service-server) bindings, (2) a service access mechanism for invoking server interface operations, (3) a service reconfiguration mechanism for making the service access operations resilient to network or server failure, and (4) a service cancellation mechanism for terminating the services in use gracefully. Our preliminary results show that the mechanism can be implemented efficiently.

1 Introduction

Most current distributed systems use the client/server model as their structuring paradigm. The abstraction of *service* in this model is a second-class entity, and is usually viewed as an abstraction of a set of (operating system) kernel calls, of a set of operations exported by language-level entities, or of a message exchange protocol between programs in execution (or active computing objects).

When the service is an abstraction of a set of kernel calls, the servers are operating system kernels. Clients access services by invoking kernel calls. For example, Locus kernels [Popek85] running on different machines cooperatively support a Unix-compatible distributed file system and a location-transparent remote tasking facility. This approach permits better control and usage of networked resources. However, it requires kernel changes to support new type of resources and new ways of utilizing existing resources. In light of the rising degree of heterogeneity of networked resources, this approach could result in very complex operating systems that are costly to maintain.

*This research was supported in part by a grant from Bell Northern Research, Inc.

When the service is an abstraction of a set of operations exported by language-level entities, both clients and servers are language-specific objects and clients obtain services by invoking server interface operations. For example, in the Emerald [Black87] distributed language, an *abstract type* represents a service interface and every object is inherently the implementation of a set of services. The compiler type checks service access invocations, and the distributed language runtime supports the invocation mechanism in a location-transparent fashion.

Distributed systems with language support provide much better programming and run-time environments than do systems with kernel-level support alone. However, this approach requires existing software to be rewritten in the chosen distributed language. The language runtime must also be available on all the nodes in the system. Consequently, requiring language homogeneity on diverse networked computers to make use of various resources is not a preferred system design approach.

When the service is an abstraction of a message-exchange protocol between programs in execution, both clients and servers are usually user-level processes running on different machines. Services in this category include the ISO/CCITT X.500 directory service and the Internet DNS. The biggest advantage of this approach is that it requires neither changes to the underlying operating systems to provide new services, nor extensive distributed language runtime support. However, considering the variety of client-server binding and message-exchange protocols that may be in use, finding a uniform way of utilizing the servers could be a formidable task.

Many leading computer companies have agreed on a vendor-neutral distributed computing environment (DCE) architecture proposed by the Open Software Foundation [OSF90]. This architecture is designed under the client/server model, and requires the interactions between its components follow the RPC paradigm.

Although the DCE architecture helps reduce the heterogeneity of server access protocols, three important issues are still outstanding: service specification, fault tolerance support, and system integration. The *service specification issue* concerns how to specify services so that service-server bindings can be changed dynamically without disturbing clients. This issue is still outstanding because the RPC subsystem [Kong90, Lamp-

son86] binds clients to servers directly. Once a client-service-server binding is established, the server cannot be changed transparently to the client.

The *fault tolerance issue* concerns how to make clients resilient to network or server failure. Clients of the RPC package in DCE are not protected well from such failures because the RPC subsystem is ignorant of the features of the services (and servers) in use. Tedious server-dependent reconfiguration algorithms must be implemented by the clients to exploit the case where more than one server is willing to provide the desired service.

The *system integration issue* concerns how to accommodate server protocol heterogeneity to provide a uniform system view to the clients. Although the DCE architecture provides a framework for developing distributed applications, it does not suggest a practical means of integrating existing server access protocols so that the clients can request, access, and cancel services through a uniform *service acquisition mechanism.*

These three issues are important because they hinder the development of robust client software, shorten the life cycle of client software when changes to networking technologies are inevitable, and discourage the exploration of new server access protocols for specialized, high-performance servers.

1.1 Client/Service Model

The above problems are not unique to the DCE architecture, and arise in all systems using the client/server model. Since the abstraction of service is a second-class entity in the model, a service cannot be realized in a server-independent fashion. When this dependency exists, the problems are formidable.

This paper presents a service acquisition mechanism for solving these problems. This mechanism is designed under a *client/service* model in which the abstraction of service is a first-class entity. This model permits the realization of the *service entity* to (1) support the client's view of service operations, (2) accommodate server access protocol heterogeneity, and (3) handle server access failure on behalf of client.

The service acquisition mechanism is composed of (1) a service request mechanism for establishing client-service (and service-server) bindings, (2) a service access mechanism for invoking server interface operations, (3) a service reconfiguration mechanism for making the service access operations resilient to network or server failure, and (4) a service cancellation mechanism for terminating the services in use gracefully.

1.2 Organization

In the remainder of this paper, Section 2 introduces the *Cygnus model*, a service acquisition model in the Cygnus distributed system. This system has been built at the University of Michigan—Ann Arbor. Section 3 presents the software architecture of the Cygnus distributed system in which our service acquisition mechanism is evaluated. Section 4 illustrates the design and implementation of the Cygnus service acquisition mechanism. Section 5 explains how the service acquisition operations can be composed to access a network service. Section 6 analyzes the cost of using the Cygnus service acquisition mechanism to access local or remote servers. Finally, a conclusion is drawn in Section 7.

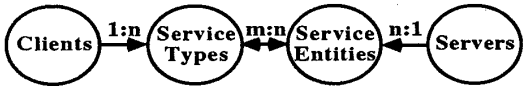

Figure 1: The Cygnus model.

2 Cygnus Model

The Cygnus model is an instance of the client/service model, and uses four classes of abstractions: servers, *service entities*, *service types*, and clients. The servers export sets of related operations to the clients. The service entities represent server interfaces. The service types are abstractions of service (see Figure 1).

2.1 Service Types

A service type is identified by a set of *attributes*. Examples of attributes are: a system-wide category for the service operations, the identity of a server administration domain, a version number, a service operation name, a performance criterion, and so on.

From the viewpoint of the clients, identifications of the service types are also service specifications. For example, the attribute list

((`category`, *compile*), (`source`, *c*), (`target`, *mc68020*), (`compile`, *yes*), (`link`, *no*), (`go`, *no*))

may signify, in Lisp-like syntax, a compile service which translates a C program into a series of MC 68020 machine instructions. The attribute list ((`compile`, *yes*), (`link`, *no*), (`go`, *no*)) requires the selected server to provide at least compile operations. The client will send neither link nor compile-and-go requests. As long as the server has the features specified, it may run on an IBM 3090 main frame, a Sun 3 workstation, an Apple Mac II personal computer, or any other machine.

We have made a commitment to attribute-based descriptive naming for service types. Such a commitment preserves the generality of the model, and the naming technology fits well with the model. We regard attribute-based naming as technically feasible because it has been used in several distributed name and/or directory services to help clients discover objects in the system [Oppen83, Peterson88, Neufeld89]. However, this naming technology is used differently in the Cygnus model. We use it to let the clients specify the services they want to access and to facilitate the establishment of bindings between service types and service entities. Instead of getting back a list of values, Cygnus clients obtain references to client-service bindings.

2.2 Service Entities

Service entities are used to capture the precise nature of service-server bindings, because a server may be willing to export more than one interface to clients through

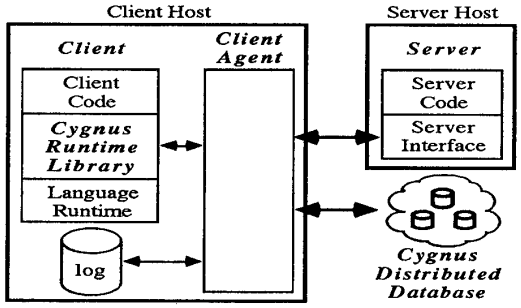

Figure 2: Cygnus distributed system architecture.

several different access mechanisms. For example, in the Mach/Mig environment [Jones86], a server may possess several kernel-protected communications channels to export different interfaces.

Figure 1 showed the relationships between the abstract entities in the Cygnus model. A one-to-many relation exists between clients and service types because a client can acquire more than one service type. The relation between service types and service entities is many-to-many because a service type may represent several service entities and a service entity many be associated with more than one service type. Similarly, a many-to-one relation exists between service entities and servers because a service entity represents a server interface and a server may export several interfaces.

3 Cygnus Distributed System

The major components of the Cygnus distributed system are *server processes* which realize the servers in the Cygnus model, a *distributed database* for maintaining service interfaces and server information, a series of compiler-dependent runtime libraries, and a number of *client agents* (see Figure 2).

The Cygnus distributed database maintains the relationships between service types, service entities, and servers. Similar to the *trader* in the ANSA/ISA architecture [Herbert89], this database is queried to discover adequate servers during the service request and reconfiguration phases.

A Cygnus runtime library includes a set of Cygnus-specific operations for use by the client code to communicate with the client agents running on the same machine. These operations are compiler-dependent because the transformation between local service acquisition invocations into inter-process communication (IPC) messages depends on the language runtime associated with the compiler. They are also OS-dependent because the required IPC mechanism must be supported by the local operating system.

The client agents are central to our realization of the Cygnus model. They mediate service-server bindings based on service specifications, accommodate server access protocol heterogeneity, and handle server access failures on behalf of the clients.

One client agent is, in principle, activated for every service type needed by a client. We run the client agent on the client host for two reasons. First, we want its link to the client to remain intact even upon network failure or partition. Second, we assume the client has no jurisdiction over other hosts. The implementation of client agent depends on such factors as: (1) the client host's hardware and operating system, (2) the network protocols available on the client host, and (3) the trust model enforced on the client host. For example, on a Sun 4/60 running SunOS 4.1, a client agent may be a single heavy-weight process or a set of cooperating ones. Local messages may be passed through BSD UNIX sockets or hardware-supported shared memory. Servers on the XNS network would not be reachable directly if the client host sits only on the DARPA Internet. The client agent would not be able to access the Kerberos ticket granting service on behalf of its client if it is not privileged enough to hold the client owner's password.

The client agent provides two kinds of failure recovery mechanisms to make its client more resilient to network or server failure. The first of these depends on the server protocol in use. For transaction-based servers, for example, the client agent would stop its execution until the server machine is up again so that the server state can be restored correctly. The other is server-independent and is a service-server (or ST-SE in the Cygnus model) reconfiguration mechanism. A simple logging and replay algorithm is used in this mechanism such that, in the event of server access failure, the client agent could replay all the logged service access requests to the new server. If the new server access protocol is different from the old one, the client agent reconverts the logged service access requests to server access requests for the new server.

This logging and replay mechanism can be implemented very efficiently for three reasons. First, the logs need not be stored on stable storage when crash recovery support for the client is not available. Second, the service-server reconfiguration mechanism requires no more than one server to be available at any one time. Unlike other replication-based fault tolerance mechanisms like that in the ISIS [Birman87] distributed system, 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 non-shared bindings between clients and service types, it is far less complicated than those used in transaction-based systems such as Quicksilver [Haskin88]. The corresponding mechanisms in those systems are designed to optimize the throughput of updating shared persistent objects, and must be coupled with check pointing and rollback mechanisms.

4 Service Acquisition Mechanism

The Cygnus service acquisition mechanism is a set of facilities used by the client agents to (1) make client-service (or Client-ST in the Cygnus model) bindings, (2) convert each service request operation to one or more server-specific remote invocations, (3) change service-

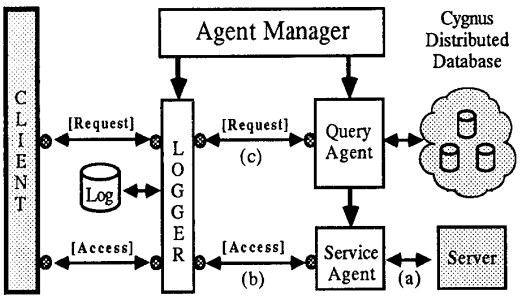

Figure 3: Cygnus client agent architecture.

server (or ST-SE in the Cygnus model) bindings whenever necessary, and (4) shut down client-service links.

Our prototype implementation of the Cygnus client agent is composed of four kinds of processes: *agent managers*, *query agents*, *service agents*, and *loggers* (see Figure 3). The agent managers receive service requests from clients and create query agents and loggers on demand for each service acquisition session. Query agents interpret service specifications, determine service-server bindings, and create server-dependent service agents locally. Service agents hide server protocol heterogeneity so that existing servers need not be modified to make them accessible to Cygnus clients. Loggers handle server access failures on behalf of the clients.

The client agent was implemented by user-level processes because we were more interested in the interfaces between them than in optimizing performance. Only one agent manager exists on each host. An instance of each of the other components may be created locally for each service requested.

4.1 Service Acquisition Phases

We now briefly describe how these component processes interact with each other during service request, access, reconfiguration, and cancellation phases. The rationale for the design and implementation of these processes will be given in the following subsections.

4.1.1 Service request phase

For each service acquisition session, the client first contacts the local agent manager to get a service request port, which is a communication endpoint for the Cygnus IPC facilities. It then composes a service request message, ships out the request through the service request port, and waits for an acknowledgment.

When the agent manager gets the client's request for a service request port, it first creates a logger and a query agent. It then passes one of the logger's Cygnus IPC ports back to the client so that the client can establish a link to the logger. It then prepares to serve the next request.

The logger first makes connections to the associated query agent and client. It then accepts the client's service specification message, and forwards the message in verbatim to the query agent. It also saves the specification message internally to support our service-server reconfiguration mechanism. After a service agent is created by the query agent for the requested service, the logger establishes a link to that service agent on behalf of the client. Finally, the logger creates a new service access port and returns it to the client.

4.1.2 Service access phase

For each service access request, the client composes a service request message and sends it to the service access port. It then blocks on the service access port until the execution results are returned.

The logger receives the access message from the client and forwards it to the service agent. The logger may also save the access request and the execution results into a log buffer as per the properties specified in the Cygnus distributed database for the client-logger link (or the Client-ST binding in the Cygnus model). Logging must be performed if the access request must be replayed when the logger-to-service agent link is broken. The logger also receives the execution results from the service agent and returns them in verbatim to the client.

4.1.3 Service reconfiguration phase

Figure 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 or network failure. The service agent may close link (b) if it cannot get the execution results from the server in time. Link (c) may be cut by the query agent because the number of queries exceeds a predefined limit.

To make the client's service access link resilient to such faults, the logger always asks the associated query agent for a new service agent when it finds the current one unavailable. If a new service agent can be created, the logger replays the logged operations. It also compares the new execution results with the old ones to ensure the correctness of the replay procedure. When the new service agent starts, it may be required to eliminate some side-effects caused by the old one(s).

If link (c) in the figure had been broken when the current service agent dies, the logger asks the local agent manager for another query agent. It then resends the saved service specification to the new query agent, and replays all the logged operations after link (b) is successfully restored. If the agent manager is unable to create the required query agent because, for example, the kernel has run out of process table entries, the logger sends the client a message saying that the service was interrupted unexpectedly.

4.1.4 Service cancellation phase

The service acquisition session is terminated when the client invokes the service cancellation operation in the Cygnus runtime library. After the logger receives the service cancellation message from the client, it (1) forwards that message to the service agent, (2) shuts down its link to the query agent if that link is still active, and

(3) terminates itself after performing some other housekeeping routines like deleting work files. The service agent terminates after notifying the remote server(s) in use. The query agent terminates after the service agent exits.

4.2 Cygnus Runtime Library

The Cygnus runtime library includes a set of compiler-dependent service acquisition primitives. These primitives are coded as per the abstractions of Cygnus IPC links. The required set of OS-dependent IPC routines, which realize the IPC links, are also included in the library.

4.2.1 Cygnus IPC operations

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) FIFOs. 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 FIFOs 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 using semaphores, Unix-domain stream sockets, or Internet-domain stream sockets. FIFOs were chosen because they appear to perform better under normal loading conditions on our client host [Chang90].

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

A set of name-based data types are defined to facilitate the communication among Cygnus clients and the client agent component processes. 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 like Sun's XDR.

These internal data types are useful to the implementation of the Cygnus service acquisition mechanism because the client agent component processes must accommodate three kinds of heterogeneity. First, since the Cygnus distributed database may comprise many specialized database servers, the query agents may have to accommodate query protocol heterogeneity. For example, most relational database servers like support SQL queries, while most name servers like DEC's distributed name service [Lampson86] have their own query protocols to meet functional requirements such as access and/or update performance.

Second, since different servers may use different data representation protocols, the service agents must accommodate such protocol heterogeneity in converting service access operations to server operations. For example, Sun RPC servers use Sun's XDR representation scheme, while DEC HDS [Falcone87] servers understand NCL (Network Command Language) data types only.

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

4.2.2 Service acquisition primitives

`RequestService`, `AccessService`, and `CancelService` are the three service acquisition primitives provided by a Cygnus library. These three operations hide the implementation details of the service acquisition mechanism by forcing the caller 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 access request messages. 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 access request.

The Cygnus library also contains a routine called `service_errno` which the client calls to get an informative error code when a service acquisition operation cannot 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 using OS- and/or language-supported light-weight 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 runtime support: `ServiceRuntime_start` and `ServiceRuntime_stop`. These two operations are provided mainly to permit the client code to reclaim resources (like file descriptors) held by the Cygnus runtime.

4.3 Service Agent

Service agents convert service operations into server operations. They are used mainly to accommodate server protocol heterogeneity. They may also be used to integrate the functions of several existing servers and to support various server-dependent recovery mechanisms.

4.3.1 Service request phase

During the service request phase, a service agent normally first initializes itself in accordance with a group of configuration parameters set by the query agent which activates the service agent.

The service agent then establishes a link to the service requester. The requester is a Cygnus client when the fault tolerance support provided by the logger is not desired by the client or cannot be applied to the desired service. From the viewpoint of the requester, the service agent is a server which speaks the IPC protocol in use.

The service agent may also try to establish a link to the associated server. If the service agent integrates the functions of several servers, it establishes links to all the supporting servers. If the link(s) cannot be set up successfully, the service agent shuts down its link to the requester and terminates itself. If the requester is a Cygnus logger, it either contacts a local query agent for a new service agent, or returns an error message to the Cygnus client.

4.3.2 Service access phase

A service agent falls into a conditional loop during the service access phase. The code segment which invokes server operations is usually a multi-way branch statement on the service operation names supported. For each operation, the service agent first extracts the input arguments from the client's access request message, and transforms those arguments into a form the server expects. The abstract service operations are then implemented by invoking one or more server operations. Finally, the remote invocation results are converted into Cygnus format and sent back to the requester.

4.3.3 Fault tolerance support

The service agent may also support various server-dependent fault-tolerant mechanisms. This resilience support complements our service access logging and replay mechanism, especially when the associated server(s) can be shared by other processes for manipulating common data objects. For example, an optimistic message logging and process checkpointing mechanism can be used by the service agents and servers to make the Cygnus services in use resilient to machine crashes [Johnson88].

4.4 Query Agent

A query agent assumes several responsibilities. It searches the Cygnus distributed database for appropriate service entities, activates service agents on demand as per the data encoded in the service entities, and returns necessary IPC information about the service agents it activates to the service requester.

Each service entity in the Cygnus distributed database contains a *server descriptor*, a *logging-and-replay descriptor*, and a list of machine-dependent *service agent descriptors*. The server descriptor records information about the supporting server. The logging-and-replay descriptor carries information on how to log and replay service access requests selectively. The service agent descriptors are used for activating local service agents.

Figure 4 depicts the structure of the query agent code. Since the Cygnus distributed database is still under development, different query modules are used as front end routines to the various databases in the system.

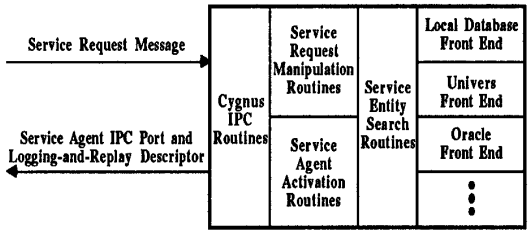

Figure 4: Cygnus query agent structure.

These query modules are integrated by a generic service entity search module through a standard interface so that the query agent code can be easily maintained and extended. This interface comprises three operations to allow the generic service entity search module to initialize, query, and terminate the front end modules.

The service request manipulation routines accept service request messages, invoke service entity search routines, control the creation of service agents, and send acknowledgment messages to the service requester. The query agent keeps its link to the service requester active during the service access phase if the requested service type can be supported by more than one service entity.

The service agent activation routines create new service agents upon request using the `fork` and `execl` system calls. It also reclaims the resources held by the service agents when they terminate by the `signal` and `wait3` system calls.

4.5 Logger

The logger uses a logging and replay mechanism to make its link to the client resilient to server access failure. This fault tolerance mechanism is server-independent because the logger records service access requests and results, not server operations and 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.

To guide loggers in performing the logging and replay operations correctly and efficiently, query agents always return 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 comprises a service operation name and a replay code which indicates how to replay the service operation. When the log code is `LogNo`, the logger lets the client communicate with the service agent directly.

During the service access phase, the logger interprets the value of the replay record only when it is instructed to do selective logging and replay. An access request is not logged when the replay code is `ReplayNo`. 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`.

```
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, "CATEGORY", 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   CancelService(&sh);
17   ServiceRuntime_stop();
18 }  /* main() */
```

Figure 5: A simple Cygnus client.

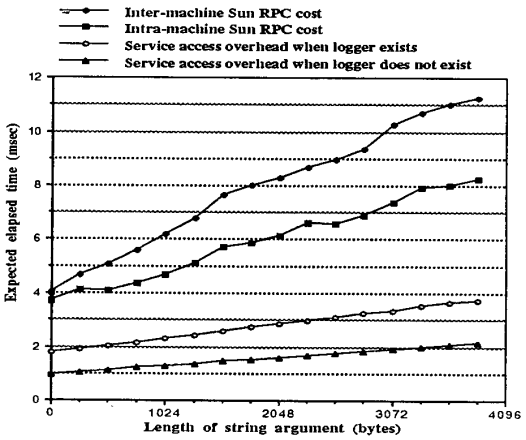

Figure 6: Performance of Cygnus service access mechanism.

During the service reconfiguration phase, the logger replays the logged access requests. In addition, a comparison-based validation scheme is adopted to ensure that the service in use would not be disrupted by accessing the new service agent. The logger, however, validates the new execution results only for the operations whose replay codes are `ReplayAll`.

The logger always asks for a new service agent when it detects the unavailability of the current one during the service access phase. It also does so when logged access requests cannot be replayed correctly during the service reconfiguration phase. The number of attempts to recover from a server access failure is currently bound by a configuration parameter set by the local agent manager.

5 An Example

Figure 5 shows a simple C program which 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 runtime. At line 8, the client code initializes an unbound service handle. It then saves attribute `CATEGORY` 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 IPC information about the allocated logger or service 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 routine `AccessService` at line 15 returns a non-negative number if the request can be processed successfully.

Lines 16–17 cancel the requested service and terminate the Cygnus runtime, respectively. These two statements are not mandatory, though they are preferred.

6 Performance Analysis

We developed a pair of Sun RPC client and server to estimate the performance overhead that Cygnus clients may incur in using the service acquisition mechanism to access local or remote 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 Rpcl (RPC Language) code specifies only one `void` function with one input argument of type `string`. The server implements that function by 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 Figure 5, it is the expected elapsed time in executing the statements at lines 12–15. Thus, the overhead is the difference between these two times.

Figure 6 shows that the overhead is small in absolute terms and acceptable in absolute terms. The computing environment was under very light load conditions when the performance data were collected. For inter-machine calls, the server ran on another Sun 4/60 workstation sitting on the same ethernet and with the same configuration as the client host.

The overhead is small because, under normal load conditions, it usually takes tens to hundreds 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 client agent is implemented by heavy-weight processes. We have estimated that scheduling delays contribute about 97% of the overhead for one-byte string.

We have also compared the performance of the Cygnus

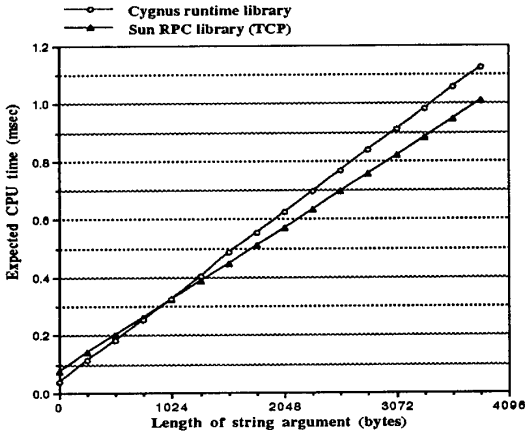

Figure 7: Performance of Cygnus runtime library.

runtime 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 runtime library. It turns out the Cygnus runtime library even consumes less CPU cycles when the size of the string argument is less than 1024 bytes (see Figure 7).

7 Conclusions and Future Work

The service specification, fault tolerance, and system integration issues are presently three of the most important and different issues in exploiting network servers. Our work shows that the client/service model provides a perspective on these issues than the client/server model, though the realization of this model can still be viewed as a client/server system at some level of abstraction. Our prototype implementation of the Cygnus service acquisition mechanism demonstrates that these issues can be solved effectively and efficiently under the Cygnus model.

We have constructed several dramatically different distributed applications to explore the limitations of the Cygnus service acquisition mechanism. We have also developed a language veneer over C to embed the Cygnus model in the extended language [Chang90]. Future work in progress includes performance enhancement, particularly for the client agent.

Acknowledgements

We would like to thank David Ballou, Peter Honeyman, Stuart Sechrest, Toby Teorey, our reviewers, and the members of the Cygnus project for their comments on earlier versions of this work.

References

[Birman87] Birman, K. and T. Joseph, "Reliable Communication in an Unreliable Environment," *ACM Transactions on Computer Systems* 5:1 (February 1987), pp. 47–76.

[Black87] Black, A., N. Hutchinson, E. Jul, H. Levy, and L. Carter, "Distribution and Abstract Types in Emerald," *IEEE Transactions on Software Engineering* SE-13:1 (January 1987), pp. 65–76.

[Chang90] Chang, Rong, "A Network Service Acquisition Mechanism for the Client/Service Model," Ph.D. Thesis, Department of Electrical Engineering and Computer Science, University of Michigan, 1990.

[Falcone87] Falcone, J., "A Programmable Interface Language for Heterogeneous Distributed Systems," *ACM Transactions on Computer Systems* 5:4 (November 1987), pp. 330–351.

[Haskin88] Haskin, R., Y. Malachi, W. Sawdon, and G. Chan, "Recovery Management in QuickSilver," *ACM Transactions on Computer Systems* 6:1 (February 1988), pp. 82–108.

[Herbert89] Herbert, Andrew, "The Computational Projection of ANSA," in *Distributed Systems*, ed. Sape Mullender, Addison-Wesley, 1989.

[Johnson88] Johnson, D. and W. Zwaenepoel, "Recovery in Distributed Systems Using Optimistic Message Logging and Checkpointing," pp. 171–181 in *ACM Symposium on Principles of Distributed Computing*, August 1988.

[Jones86] Jones, M. and R. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems," pp. 67–77 in *Proc. OOPSLA'86*, September 1986.

[Kong90] Kong, M., T. Dineen, P. Leach, E. Martin, N. Mishkin, J. Pato, and G. Wyant, *Network Computing System Reference Manual*, Prentice-Hall, 1990.

[Lampson86] Lampson, B. W., "Designing a Global Name Service," pp. 1–10 in *ACM 5th Symposium on Principles of Distributed Computing*, August 1986.

[Neufeld89] Neufeld, G., "Descriptive Names in X.500," pp. 64–71 in *Proc. SIGCOMM'89 Symposium on Communications Architectures and Protocols*, 1989.

[Oppen83] Oppen, D. and Y. Dalal, "The Clearinghouse: A Decentralized Agent for Locating Named Objects in a Distributed Environment," *ACM Transactions on Office Information Systems* 1:3 (July 1983), pp. 230–253.

[OSF90] OSF, *OSF Distributed Computing Environment Rationale*, Open Software Foundation, May 1990.

[Peterson88] Peterson, Larry, "The Profile Naming Service," *ACM Transactions on Computer Systems* 6:4 (November 1988), pp. 341–364.

[Popek85] Popek, G. and B. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.