On process aliases in distributed kernel design

K. Ravindran & Samuel T. Chanson

Technical Report 85-5
April 1985

Dept. of Computer Science,
University of British Columbia.
Vancouver, B.C., Canada V6T 1W5

ABSTRACT

As distributed computing systems become popular because of their functional, economics and reliability characteristics, a new class of problems has emerged. These problems are characterized by the fact that the resources being used by a process as well as the system state is distributed. The management of the processes and resources in such an environment present a challenge that cannot be satisfactorily met by the traditional procedural-based methods which often assume the existence of shared memories. This paper presents a multiagent structure consisting of corporate processes and their associated aliases as an efficient and systematic solution to this class of problems. The model and the kernel primitives necessary to implement the model together with some design considerations are outlined. An example described in terms of the model is also given.

On process aliases in distributed kernel design

1.0 Introduction

As distributed computing systems become popular because of their functional, economics and reliability characteristics, a new class of problems has emerged. These problems arise from the fact that the resources used by a process may be distributed and reside on different machines. Furthermore, the system state is also distributed. The management of the processes and resources in such an environment presents a challenge that cannot be satisfactorily met by the traditional procedure-based methods which often assume the existence of shared memory. This paper presents a message-based model using process aliases (which we call multiagent structure) as an efficient and systematic solution to this class of problems. An alias is a light-weight agent that performs a small and well-defined task on behalf of a process. It derives its identity from the process and does not exist by itself. The alias executes asynchronously to its associated process and may reside on the same machine or on a different machine. Thus aliases are tools by which a process may be simultaneously present at different sites in the distributed system performing different activities. We show how the multiagent structure provides a clean and uniform framework whereby previously hard-to-solve problems can be solved systematically, and in some cases easily. The model and the kernel primitives necessary to implement the model together with some design considerations are outlined. An example code description in terms of the model is also given.

2.0 Some issues in distributed kernel design

We outline some of the issues that are either characteristic of a distributed system or need to be reexamined in the context of a distributed environment.

2.1 Resource reclamation

Resource reclamation can be quite difficult in a distributed system. This is because the system state is distributed across machines. Lack of adequate support from the kernel in maintaining a consistent view of the global state at each machine may result in loss of resources.

One scenario is that of a client-server interaction across one or more gateways in an interconnected network where the kernel commits some resources at the gateways for the interaction. If the client/server dies during the interaction, the kernel abstraction should enable the reclaiming of not only the resources committed by the client/server, but also the resources that are committed at the gateways. Reclaiming the resources at the gateways requires kernel support in the form of death notification to the appropiate machines in some systematic fashion.

The problem is more severe for a connection-oriented client-server interaction consisting of a sequence of transactions (for example, a file access) than that of a connection-less interaction (for example, a single transaction of a time request to the time server). Apart from the amount of resources committed at both ends to maintain the connections (this is in addition to those required for each transaction), resources may be tied up for a long time (or even lost) if either the client or the server dies *in between* transactions without the knowledge of the other. Another problem is that if the client dies after requesting a transaction, the server should complete the requested operation in order to satisfy the atomicity of the transaction to maintain the resource in a consistent state. Thus the kernel should allow the ongoing transaction to complete even though the client has died.

2.2 Piping data between servers

We use the term piping to mean data transfer from one server to another that is essentially unidirectional. Typically, the data are not subjected to client process level interpretation but may need simple preprocessing. Examples of such transfers are file transfers and logging of telemetry data in a spacecraft application. The conventional

interprocess communication (IPC) models do not support low level stream-transfer of data. Typically in these models, the client process reads data from a source, say a disk file, into its address space and writes data to the sink, say another file. Since read/write requests are processed by the system, a single read/write cycle takes 4 context switches, two data copying and two system calls to move data from one point to another in the kernel on the same machine. This overhead is about 8.0 msec for 1 Kbyte of data in UNIX 4.2 BSD running on a SUN workstation, constituting about 50% of the system time spent in such transfers. Similar overhead has been observed in [4]. Measurements done elsewhere on a PDP-11/44 running version 7 UNIX [3] show that this overhead can be as high as 70%.

In a distributed environment, the servers and the client could potentially reside across machines adding a new dimension of inefficiency. From the measurement results on the V-system [5], we estimate that the data spend about 40% of the time in getting across from one server to the client to be piped to another server across machines. Since data are not subject to client level interpretation, the large overhead may be curtailed by providing a low level path from the source directly to the sink without routing through the client process. This eliminates unnecessary data movement, network transfers and process activations. Such application-dependent performance optimisations require efficient kernel tools on which relevant structures may be built.

2.3 Asynchronous resource access

Asynchronous resource access enables a client process to initiate multiple service requests which execute concurrently at different servers in the distributed system. The traditional shared-variable type mechanisms used in a single machine environment are too restrictive and not viable in a distributed system [1] since resources can be on different machines which do not share memory. The abstraction of an asynchronous resource access in a distributed environment requires a message-based multiprocess structure. In such a structure, the abstraction conceivably requires three processes P1 (client process), P2 and P3 with P3 blocked on the server access, and P2 providing an

intermediary function between P1 and P3. The function of P2 is to receive a message from P1 (service query) or P3 (service completion message) and reply appropriately.

2.4 Process interactions across gateways

When a process interaction extends across one or more gateways, there are network-specific protocols that have to be handled at the intermediate gateways, and the IPC operations initiated by the sender/receiver must be translated into these network-specific protocols. These intricacies should be transparent to the sender/receiver. In other words, the image activations that are initiated at these gateways should be transparent to the sender/receiver. Furthermore, as we have seen in section 2.1, the associated resource reclamation issues require some form of binding between these image activations and the sender/receiver.

2.5 Process migration

Process migration is a characteristic requirement of a distributed kernel from the point of view of performance, modularity and robustness. Such a kernel feature enables a high level process manager to schedule processes across machines during execution to effect global load control, and automatic reconfiguration in the event of site failures [6]. However this raises a number of issues to be solved at the low level. One obvious problem is the handling of messages arriving at the original site for the process that has migrated elsewhere. This requires some form of kernel-supported message forwarding mechanism at the original site.

2.6 Object search

A typical example is the location of servers in a distributed system. If objects are allowed to migrate, this issue assumes a different dimension. In such an environment, an

IPC may potentially initiate an object search. Though caching techniques may be used to contain some of these searches to the local site, a significant number of object searches will be across the network. So every object needs to have some form of answering mechanism for these search protocols.

2.7 Handling application-level emergencies

Emergency conditions are frequent phenomena in real time systems where each emergency should be treated in bounded time [7]. A typical scenario is that when a sensor exceeds a threshold indicating an emergency, an emergency message is sent to the client for immediate corrective action which could range from turning a valve ON or OFF to making a certain amount of complex decision to handle the particular emergency. The source of the emergency, the client and the sink for the corrective measures could potentially be on different machines adding a new dimension to the problem.

3.0 Solution approach

Having outlined a broad spectrum of issues that either are characteristic of a distributed system or need more efficient solutions in the context of a distributed environment, let us examine how they may be solved. One way is to provide a different solution for each of the individual problems; this typically takes the form of creating a kernel function that provides a mechanism to solve a particular problem. This approach has the following deficiencies:

(i) The solution technique is adhoc and is not extensible. Secondly, this increases the kernel complexity and size. This is evidenced in the evolution of UNIX which started with a well-knit kernel abstraction and turning out to be more and more complex as functions are added [4].

(ii) Since the issues have varying degrees of interaction with one another, individually solving the issues results in interactions between the individual solutions

which will turn out to be complex and difficult to understand.

Thus a unified approach is sought which provides solutions to a majority of the issues. Our solution method using process aliases discussed in the following sections evolved from this approach.

4.0 Process aliases

We propose ***process aliases*** as a tool to solve the class of issues outlined in section 2.0. A process alias is an ancillary process that executes on behalf of a main process to carry out a ***limited, well defined*** function. These aliases may be created by the process itself or by the kernel. In the later case, they are transparent to the process and are known as ***invisible aliases*** . The alias executes asynchronously to its associated process. The process on whose behalf the aliases execute is known as a ***corporate process*** . For simplicity, we will refer to a corporate process simply as a process in the following sections. The meaning should be clear from the context. Also the term "agents" and "aliases" are used interchangeably. An aliase uses the identity of the associated process in all its interactions with other processes/aliases. In other words, aliases are abstract tools by which a process may be simultaneously present at different sites in the distributed system performing different activities. It is different from a process in the following respects:

(i) A process is an independent entity whereas aliases do not exist by themselves, i.e., as soon as a process dies, its aliases cease to exist.

(ii) Aliases do not have distinct, high level identity, and they are not subject to high level scheduling, i.e., they are executed indivisibly until they exit or are blocked. There is no notion of priority associated with these aliases.

(iii) Aliases are only support tools provided to perform limited, well defined functions. They are small in size, and have short execution time. They may be created in a predefined state. Typical examples of alias functions are to receive a message and buffer it, read data from a message channel and write it onto

another.

It is different from a procedure invocation in that the alias executes asynchronously to the process while procedure execution is synchronous. Remote Procedure Call (RPC) [8, 9] too is different from remote aliases in the above respect in addition to the fact that RPC is a high level abstraction built on top of low level message-passing.

The rationale behind proposing aliases, rather than full-blown processes for solving the distributed system issues is the following:

(i) Because of the limited functionality of aliases, the descriptor information (as well as the state information) associated with them is usually small. This results in reduced overhead in terms of storage space, execution time and alias management.

(ii) Aliases could potentially execute within the kernel owing to their limited functionality. For such kernel-resident aliases, the kernel message facilities may be accessed much more efficiently than the case of a process accessing them via a kernel trap with the associated overhead. The creation of such aliases may be done cheaply since they do not require a separate address space, i.e., they share the kernel space.

(iii) Since aliases cease to exist as soon as the associated process dies, the problem of orphans [8] and resource reclamation common in distributed systems is minimised.

The properties of aliases are different from those used in RIG [10, 11]. The RIG aliases are independent user-level processes transparent to the kernel, and the process has to explicitly manage its aliases. Furthermore, accidental deaths of the original processes might leave the alias processes orphans. The same differences apply to alias ports used in Accent [12]. Our aliases are also different from the ghost processes used in COCANET [13]. In addition to the differences with respect to RIG aliases, the ghost processes in COCANET are essentially used to fill up the lacuna created at the local site when the local process performs a remote execution. The function of this ghost process is to anchor the environment of the remotely executing process to the local site.

5.0 Process alias based solutions

In this section, we discuss solutions to some of the problems outlined in section 2.0 based on the process alias concept.

5.1 Piping data between servers

As pointed out in section 2.2, the extent of the client's participation in the data flow is very minimal, ranging from splicing the data from one server connection to another to some simple preprocessing of data. Process aliases may be used to provide a low overhead interkernel stream transfer model in such applications.

Essentially, the client's role is distributed between two aliases each colocated with a server. A typical application process anchors one alias each to the source and sink servers in a well-connected state. These aliases explicitly control the stream transfer from source to sink ***without*** the intervention of the process. The aliases stop the stream on detection of an ***end-of-stream*** or an explicit stream abort message from the application process. These aliases could potentially execute at a low level in the kernel. Preprocessed streams may be supported by dispatching ***intelligent aliases*** to these server sites. Since the aliases execute at the server sites, the stream model effectively eliminates one hop in the network path. When the model is applied to the degenerate case of a single host system, the savings are in terms of data copying, context switchings and system call processing.

5.2 Resource reclamation

The key requirement in resource reclamation pertaining to some of the situations outlined in section 2.1 is the detection of the death of a process by another process. Conceivably, the kernel could guarantee that the process death is broadcast to all nodes. However, reliable broadcast entails heavy overhead [14]. A solution used in RIG [10] is

that the concerned processes register with the kernel that they be notified by emergency messages on a particular process' death. This scheme is practical only if the number of clients for a server is small. THOTH [2] uses a vulture-based scheme whereby a server creates a vulture that is receive-blocked on the client. On the client's death, the vulture is unblocked and notifies the server enabling resource reclamation from the server end. The death of the server destroys the vulture due to the ancestral relationship between them. A limitation in extending this scheme to distributed kernels like V and PORT [5, 15] is that these kernels do not support remote creation/destruction of processes and the associated death notifications across machines. Let us see the implications of this limitation. Suppose the server creates a vulture on the local machine M1 which becomes receive-blocked on the client residing on a different machine M2 (see Figure 5.1.a). The death of the server automatically destroys its creation, the vulture. But the death of the client is not sent to M1. Thus the vulture is unaware of the client's death, nullifying its purpose. On the other hand, if the server arranges to create the vulture on M2 by indirect means (using a process server on M2), the client's death can be detected by the vulture which notifies the server. However the server's death does not destroy the vulture since the server is not the ancestor of the vulture, thereby losing the resource being used by the vulture (see Figure 5.1.b).

A kernel model supporting both local and remote aliases overcomes the limitation in these systems. The server dispatches an alias (whose function is essentially that of the vulture used in THOTH) that is receive-blocked on the client. When the server dies, the kernel destroys the vulture alias because of its abstraction supporting the alias properties (that they do not have self-existence).

5.3 Asynchronous resource access

Basically, the process creates two aliases to handle the interactions with the server. This is illustrated in Figure 5.2. The process creates two aliases A1 and A2 with A1 blocked on the server access and forwarding the reply messages to A2. The function of A2 is to buffer the server replies and interact with the process. Such a set up enables

the process to pick up the reply messages later. A1 and A2 have functions similar to those of P3 and P2 discussed in section 2.3. This provides a more efficient solution to the problem as the aliases can be managed more efficiently. These agent functions could be placed inside the kernel to improve performance.

5.4 Handling application level emergencies

With proper support from the kernel, process aliases may be used to handle application level emergencies with a tight upper bound on response time. The occurrence of an emergency may be likened to a hardware interrupt that is serviced immediately. A similar interrupt-like abstraction at the application level may be realised as follows: the application process (client) dispatches an alias A1 to the site where the emergency condition is likely to occur and another alias A2 (which can be local or remote) to handle the emergency (see Figure 5.3). The execution thread of A2 extends into the application level interrupt handler. When A1 senses an emergency (in the form of a message from a local process, say, a sensor preprocessor), it dispatches an emergency message to A2. The kernel forces a context switch and executes A2 to handle the emergency. Simple corrective measures could be applied by the aliases themselves without getting back to the client if they are equipped with sufficient information when created. In such a situation, the remote aliases use the same model of communication discussed in section 5.1.

6.0 Invisible process aliases

These are aliases created by the kernel without explicit request from the process. The invisible aliases exhibit the same properties as discussed in section 4.0 except that they operate at a lower level than those explicitly created. They provide a good solution to problems such as those arising from IPC interaction across gateways and process migration. When the sender and the receiver are on different interconnected networks, location transparency requires that the sender be unaware of the location of the receiver, let alone the intervening gateways. The gateway implements different types of protocols

matching the network characteristics. These protocols may be encapsulated in the alias functions. In this context, these aliases may be thought of as abstract objects provided by the kernel to activate these protocols.

Consider the process migration scenario. The message forwarding function may be encapsulated in an invisible alias and installed in the original site by the kernel. Since process migration is usually an involuntary (as well as transparent) activity initiated by the kernel, the process is unaware of its alias residing on the original site. If the process migrates further, an identical alias is created on the second machine to relay the forwarded messages. When the process dies, it is the responsibility of the kernel to destroy such alias activations on both the local and remote sites.

7.0 Implementation considerations for aliases

The identifier of an alias may be derived from the corporate process itself. This allows the binding between the alias and the process to be made efficiently. This also enables the kernel to quickly authenticate the requests made by an activation (process or alias). If an alias is frequently used and its functions are well-defined, it could be made kernel-resident. These kernel-resident aliases provide an efficient mechanism to handle the various issues in the distributed kernel since their creation, destruction and the message-passing activities could be efficiently managed inside the kernel. Thus if we liken the kernel to a government and the corporate process to an individual, then this mechanism is analogous to the individual hiring various contractors owned by the government as his agents where each contractor is capable of doing a limited, well-defined function, and has access to all the internal machinery of the government.

8.0 The abstract model of a kernel supporting aliases

Our model of the process structure in a distributed environment is that of a corporate process served by a heterogeneous set of agents. The execution of this *multiagent*

structure results in the execution of the entire program in the distributed system. Some of the agents could be on remote machines executing on behalf of the corporate process; some could be kernel-resident for performance considerations. The multiagent structure can be likened to a company located at some site with a large number of agents (visible and invisible aliases) working for the company at various sites. The relationship between the process and its aliases are governed by the properties discussed in section 4.0. Thus our multiagent structure is different from the multiprocess structure as used in V and PORT in two respects:

(i) There is support for local and remote aliases.

(ii) The relationship between the process and its aliases is asymmetric.

Another difference is that V-kernel team members reside in a single address space on the local machine. More importantly, remote processes are not supported in these kernels. The process itself has to arrange for remote creations indirectly through the remote IPC supported by the kernel. This lack of kernel support for remote processes puts a severe limitation in resource reclamation discussed earlier.

8.1 Objects provided by the kernel

Having introduced the notion of a multiagent structure, we now describe our model of the kernel that supports such a structure. The kernel provides ***processes, aliases*** and ***ports*** as the primitive objects. The former two are active entities and the latter is a passive one. Besides queueing messages, ports are also used to anchor aliases onto the process. They provide an abstract mechanism to create, maintain and destroy aliases by the kernel. Ports are ***tightly coupled*** with the process in that they can be manipulated only by the process that created them. Ports are automatically destroyed when the associated process dies. The abstraction of tight-coupling and non-self-existing property of the ports is introduced to satisfy the alias properties outlined in section 4.0. Furthermore, we introduce the notion of ***typing*** the ports with each port type allowing only specific operations so as to anchor application-specific protocols as alias functions. Ports

are typed as follows:

Port_type	Permissible operations
CLIENT_PORT	Send service requests and receive service replies.
SERVICE_PORT	Receive service requests and send service replies.
SYMMETRIC_PORT	Send and receive data/control messages.

The SYMMETRIC_PORT is subtyped as SYNC, ASYNC, CONCURRENT and EMERGENCY. They are characterised as follows:

Symmetric_port_type	Permissible receive operations
SYNC_PORT	Receive a message and send a reply message as soon as the recepient picks up the message.
ASYNC_PORT	Receive a message and send a reply without waiting for the recepient to pick up the message.
EMERGENCY_PORT	Receives an emergency message, forces an immediate entry in to the appropriate emergency handler, and then replies.
CONCURRENT_PORT	Receives the appropriate typed message. The reply message is sent after the message-initiated activation is over.

The send operation on a particular port is to send an appropriately typed message and wait until a reply message arrives from the destination port. Our notion of ports is different from that of Accent [12] and UNIX IPC [16] in three respects:

(i) Accent ports are free objects whereas our ports are tightly coupled to the process.

(ii) Accent ports are untyped while ours are typed objects.

(iii) Our ports provide a mechanism for message-triggered concurrent executions within the program (EMERGENCY and CONCURRENT) while Accent ports

are exclusively used for IPC operations.

The differences (ii) and (iii) apply also to RIG ports [10, 11].

Aliases are anchored onto these ports to implement the specific port types and functions specific to these port types. Since every IPC message is to be initiated through one of these ports which might result in the creation of invisible aliases, the ports may also be used by the kernel to anchor information about these aliases. A typical skeleton of a port descriptor (in a C-like syntax; appropriate types are assumed for the variables) is given below:

```
struct port_dsp  /* Port descriptor */
    {
        port_id;
        port_type;
        struct alias_dsp *local_alias_dsp;
        struct rem_alias *remote_alias_dsp; /* Both are managed
            by the kernel on request by the process */
        struct inv_alias *invisible_alias_dsp; /* Managed by
            the kernel transparent to the process */
    }

struct alias_dsp  /* Alias descriptor */
    {
        owner_process_id;
        owner_port_id;
        owner_port_type;
        alias_type;              /* Local/remote */
        alias_func_code;
        alias_state;             /* Alias attributes */
        alias_entry_point;
        msg_in_q;
        msg_out_q;          /* Message queues */
```

```
    }

struct rem_alias
    /* Information for the remote alias as available on the
     port descriptor at the local site */
    {
        dstn_process_id;
        dstn_port_id;
        remote-site_id;
        remote_alias_state;
        remote_alias_func_code;
    }

struct inv_alias
    /* Information about the invisible alias as available on
     the port descriptor at the local site */
    {
        remote_site_id;
        remote_alias_state;
        remote_alias_func_code;
    }
```

8.2 Notion of a logical network

The distributed system is abstracted as a set of processes (with or without aliases), each executing on a logical machine, and accessed by others through a ***logical network***. In this model, each process communicates with another via the logical network, even though the processes involved reside on the same physical machine. This logical network may map onto to a physical network (when processes are on different machines) or onto a simple software loop-back (when processes are on the same machine). Thus the notion of *remote* and *local* is only a logical one. This notion has the following

advantages:

(i) It avoids the necessity of having two sets of protocols, one for local and one for remote interactions as done in the V-kernel [5] and SHOSHIN [19].

(ii) It facilitates easy and transparent relocation of objects. When an object migrates to a different machine, only the mapping of the logical network needs to be changed.

(iii) The placement of local and remote aliases is well-defined and uniform.

8.3 Object identification

An activation is identified by the quartet

<activation_id> = <reincarnation_id, host_id, local_pid, alias_id>

based purely on locally available information. The <reincarnation_id> uniquely identifies the system instance across failures, the <host_id> uniquely identifies the host in the network and the <local_pid> identifies the corporate process uniquely within the local host. The <alias_id> is a small integer that identifies the alias activation for the process; a value of zero implies that it is the id of the root activation which is the hub of all activations at both local and remote sites. A non-zero value implies that it is an alias. <alias_id> inherits its value from the <port_id>. Each port is assigned a process-wide unique integer value as its identifier.

Each host joining the network broadcasts a message looking for the highest <host_id> in the network. Each host maintains a pair of variables <highest_host_id, local_host_id>. The host whose <local_host_id> equals the <highest_host_id> in the network responds with the id. The requestor then uses the next higher id and broadcasts this to other hosts.

8.4 Kernel primitives

The services requested by the client are sent as messages <msg> to the kernel. The kernel interprets these messages to perform the appropriate services. In this sense, the messages are strongly typed. We describe the *procedural interface* to the kernel services in this section. All high level operations are realised in terms of this procedural interface. Each of the primitives returns a structured value containing the success code and other primitive-dependent data. The list is by no means exhaustive.

1. process_id = *create_process (site_id, process_attributes)*.

   The kernel creates a process on the specified site. The process_id of the created process is returned. The local kernel interacts with its peer on the specified machine to create the process. The <process_attributes> specify the initial state of the process, the initial entry point, the port that is created along with the process, the message that is to be delivered to the process on its invocation, etc.

2. err_code = *destroy_process (process_id)*.

   The kernel destroys the process specified by <process_id>. All resources associated with the process are reclaimed. All aliases for the process existing in the system are destroyed.

3. err_code = *create_port (port_type, port_id)*.

   The kernel creates a port of the specified type and id for the process. This is a local operation. <port_id> is unique within the process.

4. err_code = *destroy_port (port_id)*.

   The kernel destroys the specified port. The resources allocated to the port are reclaimed. Also it destroys all aliases (both local and remote) associated with the port.

5. <process_id, port_id> = *locate_server (server_name)*.

   The kernel locates the server specified by server_name. The kernel makes a local search of its internal tables to locate the server. If this fails, a broadcast-based

networkwide search is made. The alias for the server process replies to the message with the pair <process_id, port_id>. The kernel also caches the mapping in its local tables for future references subject to table size constraints.

6. err_code = *create_alias (alias_type, local_port_id, remote_process_id, remote_port_id, function)*.

The kernel creates an alias to implement the specified function. There is no distinct identity for the alias; it is identified by the associated port_id itself.

7. err_code = *destroy_alias (alias_type, local_port_id, remote_process_id, remote_port_id)*.

The kernel destroys the specified alias. The kernel handles the orphaned service requests that may arise due to such a destruction. The resources used by the alias are reclaimed.

8. err_code = *send_msg_to_alias (alias_type, local_port_id, remote_process_id, remote_port_id, msg)*.

The kernel sends the message <msg> to the alias associated with the port_id. The alias performs the required activities bracketed by *receive_from_process* and *reply_to_process* operations. The structure of <msg> is dependent on the alias function.

9. err_code = *send_msg (src_port_id, dstn_process_id, dstn_port_id, msg)*.

Sends a message <msg> through src_port to the specified destination port to be received by the specified process. Non-specific recipients are supported. The blocking property of the primitive and the underlying kernel activities depend on the port type which specifies the message that unblocks the sender. Note that sending a message through a port of type SERVICE_PORT is equivalent to sending an application level reply message.

10. err_code = *recv_msg (port_id, source_qualifier, src_process_id, src_port_id, timeout, msg)*.

    Receive a message <msg> on the port specified by <port_id>. The message originator is qualified by <source_qualifier> which can be SOURCE_ANY, SOURCE_SPECIFIC. The primitive can be made blocking, non-blocking and blocking with timeout by the <timeout> parameter.

11. err_code = *exit_from_port_handler ()*.

    The alias anchored on the particular port (of type CONCURRENT or EMERGENCY) returns control to the kernel, and gets initialised to a *message_wait* state. An appropriate reply message is sent.

An example of a high level operation (file serve access) using our model of the kernel is given in Appendix-A.

9.0 Conclusions

We have discussed the use of process aliases to solve a class of problems for distributed systems. Some design considerations and kernel primitives supporting the multiagent structure have also been presented. Besides lending an efficient solution, the alias approach also provides a clean and uniform framework whereby previously hard-to-solve problems can be solved systematically, and in some cases easily. The notion of aliases have been used indirectly in some systems for network IPC. However none has explored the potentials of the multiagent structure as presented in this paper. Indeed the solutions require our aliases to assume some different properties than those used in other systems. In RIG [10], the kernel does not support the notion of aliases, and the process has to explicitly create an alias for itself. In the Accent kernel, which uses free ports for IPC, the alias ports must also be explicitly created and managed by the process [12]. In both these systems, the abstract properties mentioned in section 4.0 cannot be provided by the kernel. Thus for example, the accidental death of the original process could orphan

its aliases. V-kernel supports the notion of invisible aliases in its internet communication [17]. However, they are identified separately from their processes. It uses a special bit in the process identifier to distinguish an alias from a process. There is no relationship maintained in the alias identifier with respect to its process. In such a scheme, the kernel has to make an extensive arrangement in binding such aliases with a process to, for example, locate the aliases when a process dies.

We believe the ideas presented in this paper constitute a sound foundation for further research in distributed kernel design. A system based on the model described is being implemented on a network of SUN workstations connected by an Ethernet.

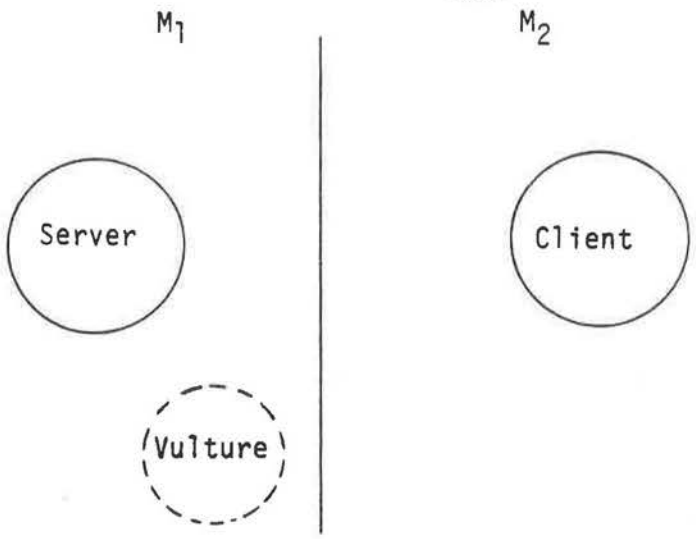


Figure 5.1.a. Vulture on the same machine as the server

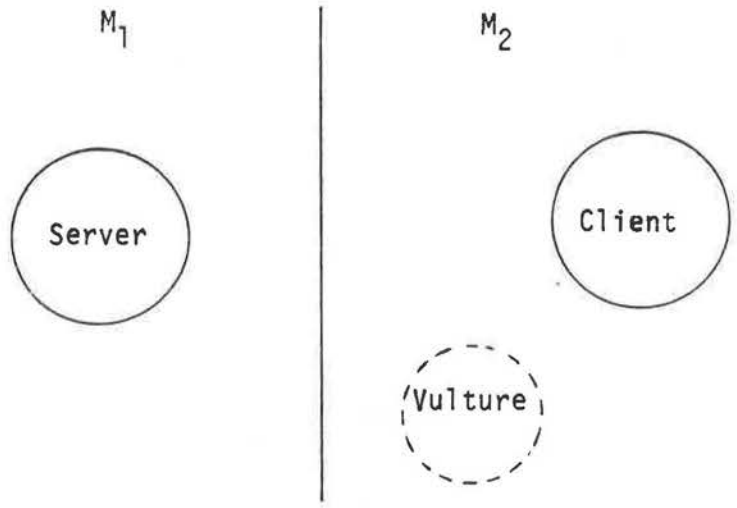


Figure 5.1.b. Vulture on a different machine from the server

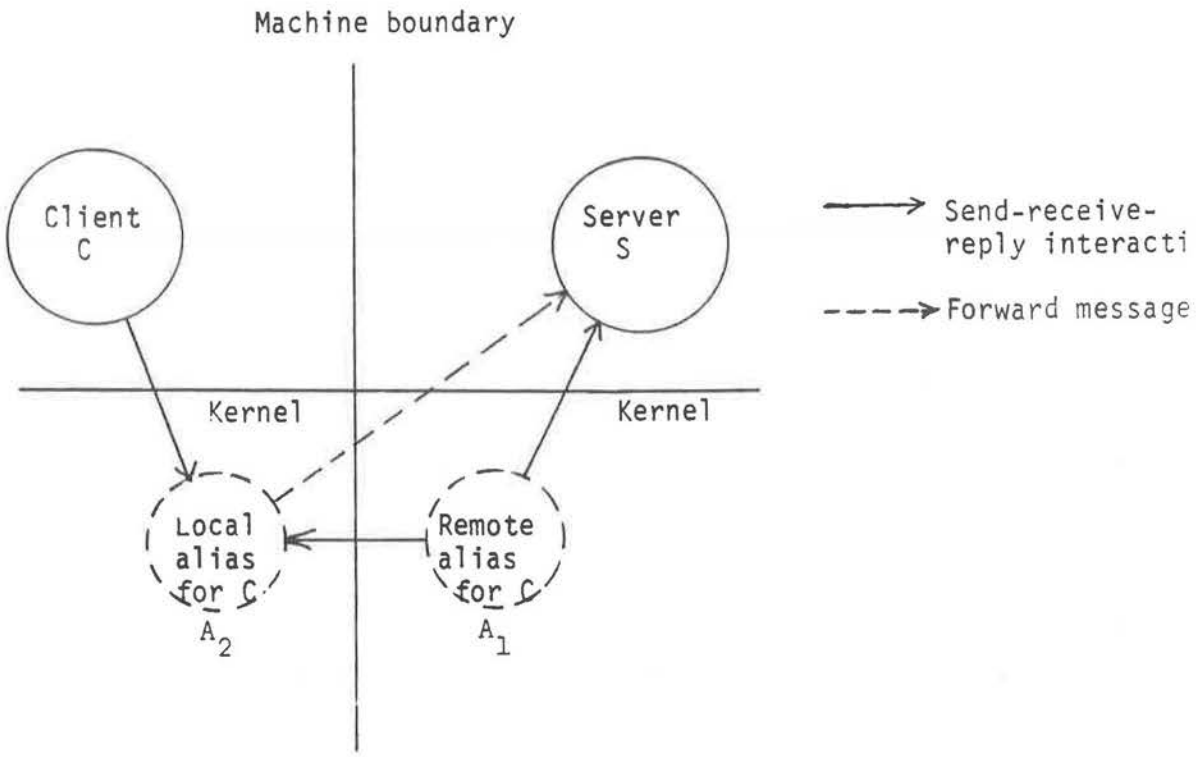
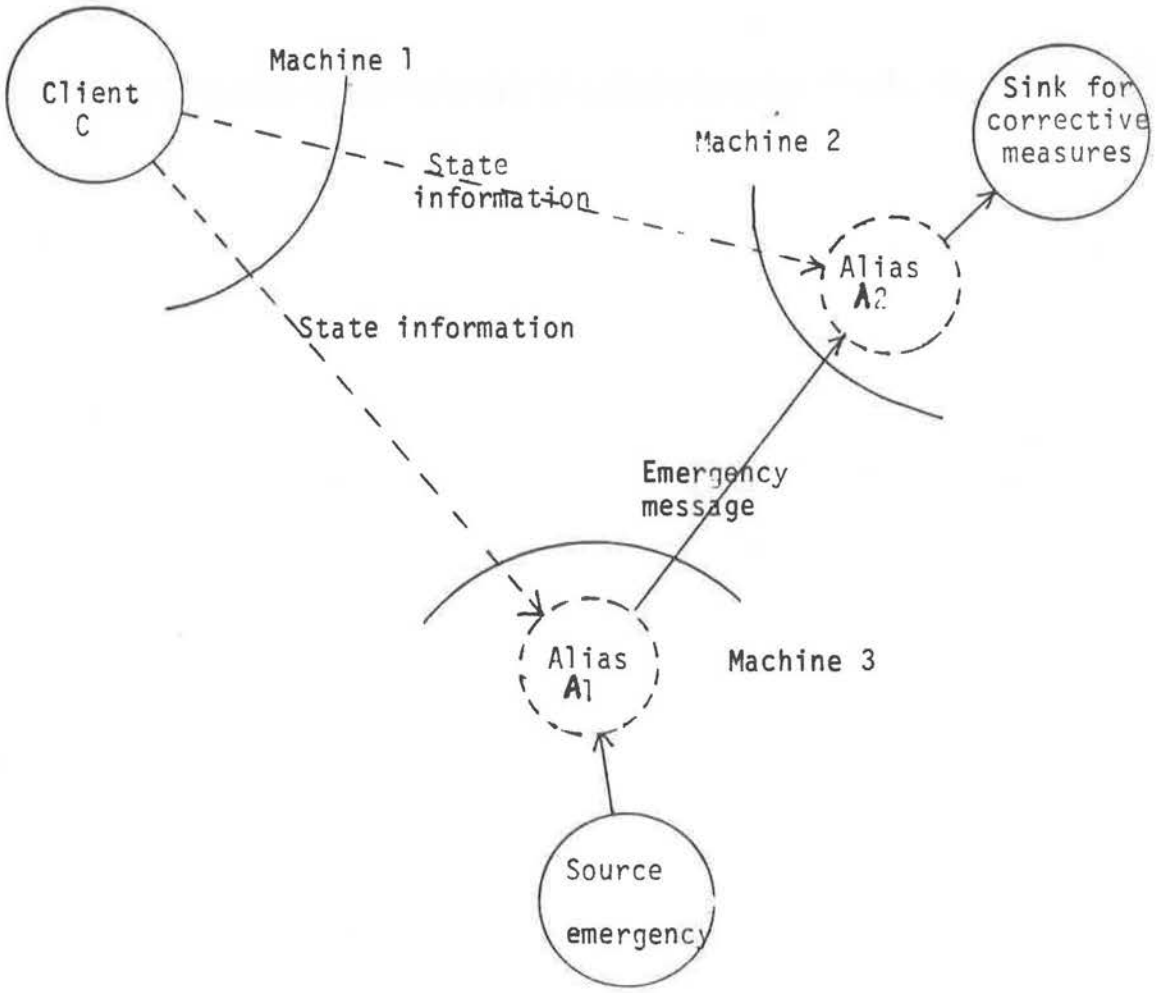


Figure 5.2. Asynchronous resource access using process aliases.



References

1. J.A.Stankovic, *Software communication mechanisms: procedure calls versus messages,* IEEE Computer, vol.15, no.4, April '82, pp.19-25.

2. D.R.Cheriton, *The Thoth system: Multiprocess structuring and portability,* Elsevier Science, '82.

3. S.R.Bunch, and J.D.Day, *Control structure overhead in TCP,* Proc. of IEEE Computer Networking Symp., May '80, pp. 121-127.

4. J.Feder, *The evolution of UNIX System performance,* AT&T Bell Laboratory Technical Journal, vol.63, no.8, part 2, Oct.'84, pp.1791-1814.

5. D.R.Cheriton, *The V Kernel: a software base for distributed systems,* IEEE Software, vol.1, no.2, April '84, pp.19-42.

6. Lampson, M.Paul, and H.J.Siegart, ed., *Distributed systems architecture and implementation: an advanced course,* Springer-Verlag, '81.

7. K.Ramamrithm, and J.A.Stankovic, *Dynamic task scheduling in hard Real-time distributed systems,* IEEE Software, vol.1, no.3, July '84, pp.65-75.

8. A.D.Birrell, and B.J.Nelson, *Implementing remote procedure calls,* ACM Trans. on Comp. Systems, vol.2, no.1, Feb.'84, pp.39-59.

9. S.K.Srivastava, and F.Panzieri, *The design of a reliable remote procedure call mechanism,* IEEE Trans. on Computers, vol.C-31, no.7, July '82, pp.692-697.

10. K.A.Lantz, et.al., *Rochester's intelligent gateway,* IEEE Computer, vol.15, no.10, Oct.'82, pp.54-68.

11. K.A.Lantz, *Uniform interfaces for distributed systems,* Ph.D. Thesis, Dept. of Comp.Sc., Univ. of Rochester, May '80.

12. R.F.Rashid, and G.G.Robertson, *Accent: a communication oriented network operating system kernel,* Proc. of 8th Symp. on Operating Systems Principles, ACM SIGCOMM, Dec.'81, pp.64-75.

13. L.A.Rowe, and K.P.Birman, *A local network based on the UNIX operating system,* IEEE Trans. on Software Engineering, vol.SE-8, no.2, Mar.'82, pp.137-146.

14. J.Chang, and N.F.Maxemchuk, *Reliable broadcast protocols,* ACM Trans. on Comp. Systems, vol.2, no.3, Aug.'84, pp.251-273.

15. M.A.Malcolm, and R.Vasudevan, *Coping with network failures and partitions,* 4th Symp. on Reliability in Distributed Software and Database Systems, Oct. '84.

16. R.F.Rashid, *An Interprocess communication facility for UNIX,* Comp.Sc. Dept., Carnegie-Mellon Univ., TR # CMU-CS-80-124, June '80.

17. D.R.Cheriton, *Local networking and internetworking in the V-system,* Proc. of 8th Data Comm. Symp., ACM SIGCOMM, Oct. '83, pp.9-16.

18. B.Walker, et.al., *The LOCUS distributed operating system,* Proc. of 9th Symp. on Operating Systems Principles, ACM SIGOPS, Oct.'83, pp.49-70.

19. H.Tokuda, and E.G.Manning, *An interprocess communication model for distributed software testbed,* Proc. of Symp. on Comm. Architectures and Protocols, SIGCOMM 83, Mar. '83, pp.205-212.

Appendix-A

Code skeleton of file server access using our model of the kernel

```
int function file_open_client ( file_server_name, file_name, file_attributes )
  <Variable declarations>
   {
     status = create_port ( CLIENT_PORT, client_port_id );
     if ( status.error_code == FAILURE )
       return ( <error_code> );
     <rqst_msg> = <file_name, file_attributes, FILE_OPEN>;
     <file_srvr_id> = locate_server ( file_server_name );
     status = send_msg ( client_port_id, file_srvr_id.process_id,
                    file_srvr_id.port_id, rqst_msg );
     if ( status.error_code == MSG_SUCCESS )
       if ( msg.reply_code == OPEN_SUCCESS )
         {
          file_desc.client_port_id = client_port_id;
          file_desc.srvr_id.port_id = file_srvr_id.port_id;
          file_desc.srvr_id.process_id = msg.instance_id;
            /* The instance that handles the client is returned */
          file_desc.attributes = file_attributes;
          create_alias ( REMOTE, client_port_id, file_desc.srvr_id.process_id,
                      file_desc.srvr_id.port_id, VULTURE );
          return ( file_desc );
         }
     /* The association between the client port and the file server
       instance is set up by this interface */
       else
         return ( FILE_OPEN_FAILURE );
     else
       return ( MSG_ERROR );
  } /* End of function */
```

The code skeleton executed by the file server proprietor is as shown:

```
function file_server_proprietor ()  /* file_srvr.process_id */
  {
   forever
```

```
    {
     status = recv_msg ( file_server_port_id, SOURCE_ANY,
                         0, 0, BLOCK, msg );
     if ( msg.code == FILE_OPEN )
       {
        <instance_msg> = <file_name, file_attributes, FILE_OPEN,
                msg.client_process_id, msg.client_port_id>;
        status = create_process ( LOCAL, <file_server_port_id,
                          instance_msg, file_server_instance,
                           instance_state> );
        if ( status.error_code == FAILURE )
          {
           msg.reply_code = INSTANCE_NOT_CREATED;
           send_msg ( file_server_port_id, msg.client_process_id,
                       msg.client_port_id, msg );
            /* Reply message */
          }
       } /* If ... FILE_OPEN ... */
     if ( msg.code == STATUS_QUERY )
       {
        <msg> = <file_server_status>;
        send_msg ( file_server_port_id, msg.client_process_id,
                   msg.client_port_id, msg );
         /* Reply message */
       }
     .
     . <other operations done by the proprietor>
     .
   } /* forever loop */
  } /* end of function */


function file_server_instance( msg )    /* Associated with a file */
  {
   forever
    {
     do_as_requested ( msg );
     status = recv_msg ( file_server_port_id, SOURCE_ANY,
                         0, 0, BLOCK, msg );
    }
  }


function do_as_requested ( msg )
```

```
<Variable declaration>
{
 switch msg.code of
  {
    case FILE_OPEN:
     {
      .
      .<file server internal operations to set up data
      . structures associated with the file>
      .
      if ( <file open is successful> )
       {
        <reply_msg> = <instance_id, file_open_success_code>
        send_msg ( file_server_port_id, msg.client_process_id,
                   msg.client_port_id, reply_msg );
          /* Reply message */
       }
      else
       {
        <reply_msg> = <file_open_failure_code>;
        send_msg ( file_server_port_id, msg.client_process_id,
                   msg.client_port_id, reply_msg );
          /* Reply message */
        destroy_process ( SELF );
       }
     } /* End of case FILE_OPEN */
    case READ_WRITE:
     {
      .
      . <validate client request, do the requested operation>
      .
      <reply_msg> = <server_reply>;
      send_msg ( file_server_port_id, msg.client_process_id,
                 msg.client_port_id, reply_msg );
        /* Reply message */
     } /* End of case READ_WRITE */
    case CLOSE:
     {
      .
      . <Destroy all associations, return all resources>
      .
      <reply_msg> = <server_reply>;
      send_msg ( file_server_port_id, msg.client_process_id,
                 msg.client_port_id, reply_msg );
```

```
        /* Reply message */
      destroy_process ( SELF );
      }
   } /* End of switch loop */
} /* End of function */
```