State inconsistency issues in local area network-based distributed kernels

K. Ravindran & Samuel T. Chanson

Technical Report 85-7
August 1985

ABSTRACT

State inconsistency is an inherent problem in distributed computing systems (DCS) because of the high degree of autonomy of the executing entities and the inherent delays and errors in communicating events among them. Thus any reliable DCS should provide means to recover from such errors. This paper discusses the state inconsistency issues and their solution techniques in local area network based distributed kernels. In particular, we deal with state inconsistencies due to i) failures of processes, machines and/or the network, ii) packet losses, iii) new machines joining or exiting from the network, and iv) processes or hosts migrating from one machine to another in the network. The solutions presented are mostly provided within the kernel itself and are transparent to the applications.

State inconsistency issues in local area network-based distributed kernels

1.0 Introduction

As local area network (LAN) technologies move out of the laboratory and into the market place interests in LAN-based distributed computing systems (LDCS) have become widespread. Several experimental LAN-based distributed kernels have been built in the last few years including the V-system [3], Rochester's Intelligent Gateway [6], PORT [9], Accent [7], SHOSHIN [11], DEMOS[13] and Eden [18]. These distributed kernels manage objects which may physically reside on different machines. The executing entities in such systems exhibit a high degree of autonomy but may interact with one another from time to time. Thus a state change in one entity may cause the state of one or more other entities to change. State inconsistency is an inherent problem in such systems because entities may fail independently (and thus without immediate knowledge) of one another and because of the inherent delays and errors in communicating events among them [4,25]. The problem is compounded as entities often only maintain partial state information for practical reasons. Thus any reliable LDCS should provide means to handle such errors.

State inconsistencies may arise at different levels in a LDCS involving different issues and solution techniques. We shall confine ourselves to state inconsistency issues in the kernel of a distributed operating system. We describe four common classes of events that will lead to state inconsistencies and present solutions for each. The classes of events are i) failures of processes, machines and/or the network, ii) packet losses, iii) new machines joining or exiting from the system, and iv) processes or hosts migrating from one machine to another in the network. The solutions presented are mostly provided within the kernel itself and are transparent to the applications. The solution techniques are largely based on the concept of process aliases supported in our model of the kernel, and they make extensive use of a simple failure detection protocol. For this reason, we

shall first briefly describe the distributed environment assumed and the essential aspects of the kernel model in the next section.

2.0 The distributed environment

2.1 Message-based distributed kernel

This is the layer that provides the primitive abstract tools using which the operating system service layers and applications are structured. It runs on a set of workstations interconnected by a LAN. The characteristics of the underlying hardware base are:

- non-shared memory among the computing elements,
- a high bandwidth network medium with low error rate and a broadcast capability [24] that forms the interconnection backbone of the workstations.

The basic functions of this layer are:

(i) to provide the process and the host abstractions across machines with the underlying interprocess communication (IPC) by messages and interhost communication (the related issues form the focus of this paper),

(ii) the traditional local functions such as interrupt handling, device management, local scheduling of processes and memory management.

Several variants of message passing kernels have been designed and implemented [2, 6, 7, 10, 11]. A good treatment of the various message-passing abstractions may be found in [1]. We restrict ourselves to a specific form of message-based abstraction called multiprocess structuring [2] that has been adopted in the V-system and PORT [3, 9]. However, the solution techniques are applicable to other message-based models as well, and even to procedure-based models [5] with appropriate translation of the solution techniques to match the underlying model.

2.2 A typical message transaction

The client sends a message to the server and remains blocked until the server receives the message and replies to the client or an error message is dispatched to it. If when the server issues a *receive* request to the kernel, there is already a message from the specified process in its input buffer, the message is delivered to the server; otherwise the server is blocked until a message from the specified process arrives. After servicing the request, the server dispatches a reply message which unblocks the client; the reply operation is stateless in that it is not acknowledged. Thus there is an implicit association between the client and the server on a per-transaction basis. The synchronous operations (blocking send and receive) are desirable because they improve the clarity of the program structure and help make possible the verification of the program for correctness [2].

2.3 Process aliases [19]

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 aliases execute asynchronously to its associated process and may reside on different machines. An alias uses an identity derived from that 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.

(iii) Aliases are only support tools provided to perform limited, well defined functions. 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.

Our model of the kernel supports the above alias abstraction across machines.

2.4 A simple failure detection protocol

The purpose of the protocol is to allow executing entities on different machines to ascertain the existence of one another. It is an asynchronous polling type protocol. The basic structure of the protocol is very simple and has only two states NORMAL and AYT_AWAIT (AYT is a synonym for *Are You There?*). The protocol defines a single packet type AYT which is used to probe the existence of the peer. When the remote entity does not exist, the kernel on the remote site returns an error message NON_EXISTENT. When the site hosting the remote entity becomes inaccessible due to site failure or network failure, timeout enables eventual recovery. Thus the protocol *asynchronously* detects failure. The protocol is best described by the Finite State Machine (FSM) diagram in Figure 1. Such an asynchronous protocol may be built on top of any other protocol, as was done in LNTP [20] as a failure detection mechanism.

We now present the common classes of state inconsistencies and outline their solutions. Wherever appropriate, the recovery schemes used in existing systems are described and compared to our proposed solutions. Examples are mostly drawn from the V-system as it is one of the more complete distributed kernel in existence and is operational in our Distributed Systems Research Laboratory on a network of SUN workstations.

3.0 State inconsistency due to failures of some kernel objects

We shall call this type of failures *partial failures.* They may originate in different forms such as power failure, hardware failure, system crashes and software errors. Usually such failures are localised since the various active components do not depend on one another for their functional existence. Whatever the cause, partial failures are viewed by

the distributed kernel as one of process deaths, machine failures or network failures. Such failures occur asynchronously and induce state inconsistencies which might lead to loss of resources, heavy flow of unnecessary messages, etc. The recovery techniques for each class of partial failures are different and are described below.

3.1 Process deaths

The death of a process is generally not a problem unless the process is involved in an interaction with another process residing on a different machine. In this case unless the kernel provides adequate support for death detection and notification, loss of resources may result. We shall consider the client-server interaction which may either be connectionless or connection-based.

3.1.1 Connectionless client-server interaction

A connectionless client-server interaction is exemplified by the message transaction described in section 2.2. We examine the state inconsistencies caused by the death(s) of the interacting processes and the recovery techniques.

3.1.1.1 Failure recovery protocol in the V-system

In the V-system [3], a remote send operation by a client creates a passive alien descriptor at the site where the server resides. This alien descriptor is a mirror image of the process descriptor of the client, and its existence depends on the existence of the server process i.e., when the server dies, the client's alien descriptor is also destroyed. The kernel at the client site executes an asymmetric protocol by which a *remoteSend* message is retransmitted periodically. The kernel at the server site responds with a *breathofLife* message if the server is alive i.e., if the alien descriptor exists; it returns a negative acknowledgement *nAck* if the server has died i.e., if the alien descriptor has

been destroyed.

When the server replies, the kernel at the server site first checks for the existence of the associated alien descriptor. If the alien descriptor does not exist, the operation fails. If it is present, the kernel dispatches a *remoteReply* message. The alien descriptor is then scheduled for destruction after a specified time interval; if during this interval, a retransmitted *remoteSend* message arrives from the client site, the kernel retransmits the reply and reinitialises the destruction schedule of the alien descriptor. The alien descriptor is destroyed when the schedule expires.

Due to its passive nature, the alien descriptor is not automatically destroyed upon the client's death. Thus the existence of the client's alien descriptor at the server site does not guarantee the existence of the client. Since the reply operation is stateless, the server may never know about the status of the operation as viewed from the client site. Consider a simple scenario in which a client requests a server administrator [16] to create an abstract object say, a file. Normally such an operation is initiated with a single message transaction from the client to the administrator. If the client dies immediately after requesting the operation, the resources acquired for the client by the administrator may become lost. Thus the server must be notified of the death of the client in a proper recovery scheme. We now describe a proposed solution.

3.1.1.2 A robust model of failure recovery

Our model of failure recovery manifests in two forms:

(i) a symmetric protocol executed by active agents (aliases) for the client.

When a client issues a remote send operation, the kernel creates (on behalf of the client) two invisible aliases, one at the local site and one at the server site; these aliases engage in the failure detection protocol described in section 2.4.

(ii) a reliable reply operation.

A primitive of the form

reply_with_ack (msg, pid)

is used to ensure successful delivery of the reply message <msg> to the client identified by <pid>. The reply operation blocks the server until notification of delivery of the message to the client arrives. Absence of such a notification results in the failure of the operation.

In applications where reliable reply is not crucial say for example, a time request to a time server, the stateless reply operation may be used. In either case, the completion of the reply operation destroys the client alias at the server site. Let us see how this scheme handles process deaths with respect to a connectionless client-server interaction (see Figure 2 and the FSM diagram of Figure 3).

If the server dies before receipt of the message from $client_i$, the kernel returns a NON_EXISTENT message with which the client recovers. If the server dies after the receipt of the client's message, the alias at the server site (CA_{si}) detects this, sends an error message KILL_YOURSELF to its peer at the client site (CA_{ci}) advising close down of the transaction and then destroys itself. CA_{ci} learns of the failure of the transaction on receiving the error message (or if it is lost, via the failure detection protocol) in which case it destroys itself after unblocking the client with a failure code NON_EXISTENT_PROCESS, thereby terminating the interaction.

If $client_i$ dies after sending a request message, the kernel on the client site destroys the local alias CA_{ci} after dispatching the error message KILL_YOURSELF to CA_{si}; the news will eventually reach CA_{si} in which case it destroys itself. When the server replies to the client, CA_{si} may be in one of three states:

(i) normal state,

(ii) recovery state,

(iii) destroyed following the detection of the death of its peer CA_{ci}.

In case (i), the reply operation fails on receipt of the NON_EXISTENT error message from the kernel at the client site when CA_{si} dispatches the reply message. In case (ii), the reply is held back until CA_{si} detects the death of its peer CA_{ci} and destroys itself; then the reply operation fails. In case (iii), the reply operation fails immediately.

3.1.2 Connection-oriented client-server interaction

The problem is more severe for a connection-oriented client-server interaction consisting of a sequence of transactions (for example, a file access). Significant amount of resources are committed at both ends to maintain the connection (this is in addition to those acquired and released for each transaction). These resources may be tied up for a long time (or may even be lost) if either the client or the server dies *in between* transactions without the knowledge of the other. We illustrate the problem with the example of a TCP/IP [22, 23] Internet Server implemented on the V-system [17] (See Figure 4). The client process (*rlogin client*) requests the Internet Server to create a TCP connection to a well-known socket address on which the *rlogin server* on a remote host say, UNIX listens. The Internet Server creates a set of processes (TCP process, Query process, Timeout process, Network process, etc.) which maintain the peer state information for the connection at its end. Similarly, the UNIX kernel at the other end of the connection commits a set of resources (in the form of shared data structures, protocol control blocks, packet buffers, socket control blocks, etc.) to the connection. Suppose the client process suddenly dies, the resources committed at the Internet Server machine and on the UNIX remote host are to be reclaimed. The key issue is again the detection of the client's death by the server. However, the server in our current version of the V-system does not contain any detection and recovery mechanisms.

THOTH [2] uses a vulture-based scheme in a single machine environment whereby a server creates a vulture that becomes 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 directly using this scheme in a distributed environment is the notification of process deaths 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. The death of the server automatically destroys its creation, the vulture. But the death of the client is not notified 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 (say 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 resources being used by the vulture.

The kernel solution proposed in section 3.1.1.2 is to provide failure recovery *during* a connectionless interaction; thus the kernel-created aliases are destroyed after each message transaction. In a connection-oriented interaction, the death of the client *in between* message transactions must also be detected. A conceivable solution is to extend the above-referenced kernel solution to a connection-oriented interaction but this calls for additional connection-oriented kernel primitives, and the kernel has to maintain the connection information resulting in increased kernel complexity. The preferred solution is for the server S to dispatch a remote alias S_a to the client site (see Figure 5). The function of S_a is essentially that of the vulture used in THOTH. When S dies, the kernel destroys S_a because of the abstraction supporting the alias properties (that aliases do not have self-existence). The realisation of the binding between a process and its remote alias (in this case, S and S_a) is described in appendix-A.

An alternative scheme that is client-driven which we will call the *death-will* scheme is described below. The client prepares a death-will list containing a list of processes that are to be notified upon its death. Typically, when the client opens a connection to a server, it includes the server in its death-will list. It also optionally specifies the message to be delivered to the processes on the list and registers the death-will with the kernel. The kernel sets up an invisible alias at the client site and one alias for each of the processes on the death-will list to be resident at the site of the associated process. The death-will message is deposited with these remote aliases. Each of the remote aliases engages in the failure detection protocol (described in section 2.4) with the alias at the client site. When the client dies, the kernel destroys the local alias and dispatches an EXECUTE_DEATH_WILL message to each of the remote aliases. Upon detection of the client's death, the remote aliases deliver the death-will message to the respective processes and destroy themselves. Typically a server handles the message of type

DEATH_WILL_MSG by destroying the internal resources committed for the client.

3.2 Machine failures

From the server's or client's point of view, a machine failure has the same effect as a process death as far as high level recovery is concerned. The basic difference is that when a machine fails, all processes and foreign activations including the kernel on that machine die. Thus the underlying techniques to recover from machine failures and process deaths are different. In the former case, the absence of messages from the kernel concerned initiates recovery whereas in the later case, positive error messages initiate the recovery.

Consider the failure recovery protocol in the V-system for a connection-less client-server interaction (section 3.1.1.1). When the server site fails, the kernel at the client site times out after a fixed number of retransmissions and unblocks the client process with the error message KERNEL_TIME_OUT; but when the client site fails, a similar problem, as discussed in the case of process deaths, arises due to the semantics of the *reply* operation and the passive asymmetric structure of the alien descriptor. Now consider our failure recovery model of section 3.1.1.2 (see Figure 2). When the client site fails, C_i and CA_{ci} are destroyed. CA_{si} detects this by the protocol described in section 2.4. It returns an error message SITE_UNREACHABLE to the server if the later had already issued the *reply_with_ack* request. It then destroys itself. If the server replies later, the operation fails immediately due to the non-existence of CA_{si}. When the server site fails thereby destroying S and CA_{si}, CA_{ci} detects this and destroys itself after unblocking C_i with an error message SITE_UNREACHABLE.

In the case of the connection-oriented interaction as discussed in section 3.1.2, our proposed vulture-based solution enables recovery from site failures as well. When the client site fails thereby destroying C and S_a (see Figure 5), the underlying abstraction (refer to appendix-A) delivers the error message SITE_UNREACHABLE to S enabling recovery. If the server site dies destroying S, the same abstraction destroys S_a; the client

detects the death of S upon sending a message to it and recovers. The death-will scheme (discussed earlier) also enables recovery from site failures as follows: when the client site fails, the client alias at the server site detects this, delivers the death-will message to the server and terminates itself. If the server site fails, the client alias at the client site detects this and excludes the server from its polling list.

In our model of the kernel, a distributed program is composed of a root activation (process) and a set of dependent activations (aliases) some of which may execute on different machines [19]. When a machine fails, all processes and foreign activations executing on that machine are also destroyed. In general, the kernels running on the other machines in the network recover from machine failure by

(i) destroying the dependent activations created at their sites by processes in the failed machine,

(ii) sending notifications to the appropriate processes executing in their sites which might have created dependent activations on the failed machine.

Since the distributed kernel binds these dependent activations to the root activation with appropriate recovery messages against site failures (see appendix-A), the above recovery will eventually complete.

3.3 Network failures

The failure of the network may cause the partitioning of the subnet with the active machines delinked from each other [9]. This may result in the partitioning of the interacting processes. However, because of the complete break down in communications, to each of the interacting processes on different machines across a partition the network failure has the same effect as the failure of the site holding the process it is trying to communicate with. The recovery techniques are therefore the same as in the case of machine failures discussed earlier for the connection-less and connection-oriented interactions.

4.0 Inconsistency due to loss of packets

Messages may be lost in a loaded system due to channel errors, buffer shortage in the network interfaces and/or lack of high level resources [29, 30] such as alias descriptors. Let us see the implications of message loss with respect to the reply operation by the server during a message transaction (section 2.2). Both the stateless *reply* operation and the *reply_with_ack* operation in the case of a connection request initiated by a client will be considered. Refer to Figure 2.

4.1 Stateless reply

After the reply message is dispatched, CA_{si} is destroyed and the message transaction as viewed from the server site succeeds with the consequent committment of resources for the client. Suppose the reply message is lost, recovery may be in two ways:

(i) The kernel on the client site retransmits the *remoteSend* to the server. Since the *reply* operation is stateless and the kernel on the server site does not guarantee to maintain the sequence numbers of the various message transactions, it may be taken to be the retransmitted request packet of a new message transaction even if message sequence numbers and retransmission counts are kept in the packet.

(ii) CA_{ci} gets an error message NO_ALIAS (which indicates the existence of S and the non-existence of CA_{si}) from the kernel at the server site when the underlying protocol (see section 2.4) probes the later. Thereupon, CA_{ci} returns a TRANSACTION_FAILURE error message to the client process and destroys itself. The client process then reissues the request.

In both cases, the server creates a new instance of the connection thereby losing the resources committed to the previous request (see Figure 6). Thus the inconsistency leads to loss of resources if such duplicate transaction requests are not identified.

4.2 reply_with_ack

The *reply_with_ack* primitive which handles client deaths (see section 2.2.1.2) does not solve the problem fully. When CA_{ci} receives the reply message, it dispatches a "reply_ack" packet to CA_{si}, returns a success code to the client process and destroys itself. Suppose the "reply_ack" packet is lost. CA_{si} times out, retransmits the reply message and gets an error message NO_ALIAS (which indicates the existence of $client_i$ and the non-existence of CA_{ci}) from the kernel at the client site. CA_{si} then terminates the operation by returning a TRANSACTION_FAILURE error message to the server and destroys itself. Such an inconsistency typically results in the client getting a false <instance-id> of the resource that was originally committed by the server and subsequently destroyed when the server detected the failure of the *reply_with_ack* operation.

4.3 Extension to i/o protocol

The inconsistencies arising due to packet loss can not be solved at the kernel level; the solution should manifest as a feature of the client-server i/o protocol. Our solution to the problem is to extend the standard i/o protocols [27, 28] by identifying each transaction request (typically an *open* request on a file) with a unique <transaction_request_id>. The server associates this <transaction_request_id> with the resource allocated. Thus a typical resource descriptor is of the form

```
struct instance
    {
     struct transaction_request_id trans_req_id;   /* Id of the last
            transaction operated on this instance */
     struct instance_id inst_id; /* id of the resource instance
                    that was created for the client */
     struct process_id client_pid; /* process id of the client for
                    whom the resource was committed */
     struct resource_type resource;  /* control blocks,
                    object descriptors, etc */
    } instance_descriptor;
```

Such a state information associated with the resource itself enables the server to discard duplicate transaction requests that may show up due to low level inconsistencies.

If, upon the failure of the previous *send* operation containing the transaction request, the client chooses to reissue the request, the i/o protocol requires that the client assigns the same id to the transaction request. Thus the server is able to distinguish duplicate requests. On the other hand, if the client chooses not to reissue the request, the i/o protocol allows the client to send a message to the server advising deallocation of any instance that might have been created in response to the particular transaction request. The code skeletons executed by the client and the server are shown in appendix-C.1.

If the server uses the *reply_with_ack* operation and the "reply_ack" is not received (section 4.2), the <instance_id> received by the client in response to its connection request is invaild. When the client makes subsequent transaction requests with this <instance_id>, the i/o protocol requires that the server responds with an INVALID_INSTANCE_ID error message with which the client recovers. The code skeleton executed by the server is shown in appendix-C.2.

5.0 Process migration

Process migration is desirable in a distributed system 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 [12, 13]. Since process migration directly manifests in a dynamic movement of the locus of execution from one site to another, the implementation of the process abstraction should be site-independent.

Typically, a process is identified by the pair

<process_id> = <site_id, local_pid>

where <site_id> is usually the network station address of the site in which the process was created and <local_pid> is an id generated locally to be unique within the site [26]. This ensures global uniqueness of the id and provides a direct mapping of <process_id> to the appropriate <site_id> for launching a packet [3, 11, 14]. The scheme is efficient since little time is spent for this mapping function. However the scheme tightly associates the process abstraction with the site, and hence the underlying protocols for local and remote message operations. In an environment which supports migration, the <site_id> field of the <process_id> only ensures global uniqueness but does not necessarily indicate the current location of the process. Instead, a mapping of the form

<process_id> --> <site_id>

is maintained by the kernel in a cache. When a message transaction is initiated by a client, the kernel seaches its cache to locate the corresponding mapping entry. Error may arise in two cases:

Case 1.

There is no entry in the cache (due to size limitations of the cache, update policy, etc). The protocol supporting the *send-receive-reply* then initiates a broadcast-based search for the process across the network. The kernel which hosts the concerned recipient process responds with its site id so that the client can complete the message transaction. The kernel at the client site may then, subject to its cache management policy, add the entry in its cache. The absence of a response to the search message results in the failure of the operation.

Case 2.

The kernel finds a mapping entry in its cache namely <process_id>--><site_id$_1$> but the mapping might be inconsistent due to migration of the recipient process from site$_1$ to site$_2$. Such an inconsistency arises since a migration event cannot be instantaneously propagated to other machines. This may cause messages to be misdirected and lost, and may even lead to high level state inconsistencies and resource loss. Take for example, the case of a connection-oriented interaction from a client

with, say a file server. If the file server migrates in the midst of an ongoing client-server connection, the kernel at the original site responds with a NON_EXISTENT_PROCESS error message to a client's request. The client may recover by say, releasing the resources (data structures, processes, etc.) committed from its end, thereby orphaning the resources committed at the migrated server's end.

Thus kernel level solutions as part of the abstraction that supports migration are necessary to handle such errors.

5.1 Solution techniques

Essentially, the solution approach consists of two steps namely i) handling the misdirected messages, and ii) updating the state (i.e., the mapping entry) in other sites to stop the flow of misdirected messages. A reliable broadcast of the state to all nodes in the distributed system is an obvious solution; but this entails heavy overhead [8]. Furthermore, when the process interactions exhibit some degree of locality with respect to other processes [4], it is unwarranted to send this information to all nodes. We outline a set of solution techniques based on process aliases [19]. All the schemes assume a mapping entry namely <process_id>--><site_id> in the cache at the client site which becomes stale when the server migrates.

5.1.1 Client-driven scheme

The kernel installs an invisible alias at the original site when the process migrates to a new site. The function of this agent process is to respond with a message PROCESS_MIGRATED to a client's request message. The underlying protocol that supports the *send-receive-reply* transaction then initiates a broadcast-based search for the migrated process. When the answering agent for the migrated process at the new site responds with a HERE_I_AM message, the kernel at the sending machine updates its cache with this information to be used for subsequent message interactions with the

migrated process, and then completes the message transaction; the failure of the search results in the failure of the transaction. The protocol implemented is depicted by the FSM diagram of Figure 7.

5.1.2 Message forwarding by agent

The kernel installs an intelligent agent at the original site when the process migrates. The function of the agent is to forward the messages directed to the migrated process to the new site. Thus unlike the client-driven scheme, the protocols supporting the *send-receive-reply* need not include a search for the migrated process. The agent at the original site executes the same *send-receive-reply* protocol as the client site but with a different <process_id>--><site_id> mapping. Since each message from a client to the migrated process as well as the reply message has to be forwarded, the number of such messages exchanged is atleast doubled.

A simple variant of the scheme is for the agent to piggyback on the reply message to the client the new state information which may be cached by the client site for subsequent communication with the server directly.

5.1.3 Server-driven scheme

In this scheme, when a process is migrated to a new machine, the kernel installs an invisible alias for the process at the new site. The function of this alias is to repeatedly broadcast the state

<process_id> --> <new_site_id>

to other nodes by a message I_HAVE_MIGRATED for a fixed number of times and then destroy itself; the interested nodes may update their mapping entry. Failure of a message transaction is still possible if a message has been sent before the I_HAVE_MIGRATED message is received or if the I_HAVE_MIGRATED message is not

heard by a client site. An error message NON_EXISTENT_PROCESS is returned in these cases which initiates a broadcast-based search for the migrated server across the network similar to that described in section 5.1.1. If the search also fails, then the message transaction fails. The FSM representation of the protocol is very similar to that of Figure 7 except the message that triggers the search is NON_EXISTENT_PROCESS instead of PROCESS_MIGRATED. This scheme is less complex than the client-driven and message forwarding schemes but the probability of failure is higher.

5.2 Failure recovery of the schemes

All of the above schemes have to address the common issue of how long these aliases should remain alive. Our model of the kernel [19] implicitly binds these aliases to the migrated process. In case of remote aliases (sections 5.1.1 and 5.1.2), such a binding is realised by the asynchronous polling protocol described in section 2.4; this protocol is executed by two kernel-created invisible aliases similar to those described in appendix-A. When the migrated process dies, the kernel eventually destroys the associated aliases.

6.0 New machines joining the network

The first network level activity initiated by a machine joining the network is the acquisition of a site_id. A simple static assignment of site_id's to machines is too restrictive and precludes easy reconfiguration and host level migration. A robust distributed technique is needed to dynamically allocate new site_id's. site_id's have the same requirement as that of process_id's, viz., they should be systemwide unique and non-reusable.

In the V-system, each kernel maintains a host address table of Ethernet station addresses used in the network. On powerup, the host reads the hardware Ethernet address of the local network interface and searches the host address table for a match. When a match is found, the index into the table is used as the <site_id> for the

machine. Since Ethernet addresses are known to be universally unique [15], the <site_id> thus generated is guaranteed to be systemwide unique. The scheme however tightly binds the <site_id> to the network station address precluding transparent host-level reconfiguration.

We now describe a three-stage approach for the dynamic allocation of unique site_id's.

6.1 Step1: Acquiring a tentative id

The host initially acquires by some means (such as one of the methods detailed below) a tentative id that can be used as a bid in the network for use as the <site_id>.

6.1.1 A localised scheme to generate a tentative id

A tentative id may be acquired by generating a random number with the network station address as the initial seed value. Though unsuitable to be statically allocated as the site id, the network station address is systemwide unique. This minimises the probability of two or more hosts generating the same tentative id.

6.1.2 Acquiring a tentative id from the network

In this scheme, a logical ordering in the assignment of id's to hosts is maintained based on the time of joining. A host joining at time t_j is allocated an id whose numerical value is greater than that of a host that joined at time t_i if $t_j > t_i$. Each host maintains a state-pair, namely,

<self_id, highest_site_id>.

where <self_id> is the site id of the host and <highest_site_id> is the highest host id

across the entire system. When a new host wishes to join the network, it broadcasts a search message looking for the highest host id in the network. The node whose <self_id> matches its knowledge of the <highest_host_id> responds with its id.

The initial search message may go unanswered if the host that wishes to join is the first machine in the network, or if none of the sites have their <self_id> equal to the <highest_host_id>; the later is possible if the machine holding the <highest_host_id> as <self_id> has failed. In such an event, the site broadcasts a second type of search request requiring all sites with

highest_host_id - Id_range <= self_id <= highest_host_id

(where <Id_range> is an integer specified by the new host) to send their respective <self_id> values to the broadcasting site. The host then filters the highest id from among the replies. Failure to get any response results in the host rebroadcasting with logarithmically incremented <Id_range> values until the MAX_ID_SPACE is exhausted. If there is no response still, the host assumes that it is the first machine joining the network, and takes LOWEST_HOST_ID as the tentative id.

Since there may be sites in the network whose state-pair is not up-to-date, there may be more than <Id_range> + 1 replies. This is illustrated by the simple scenario where a new $host_i$ joins the network and assumes an id $<host\_id_i>$; this information is subsequently broadcast to other sites. If $host_{i-1}$ did not hear the message then both $host_{i-1}$ and $host_i$ believe they are holding the highest host id. When a new host intends to join the network by broadcasting an initial search message with <Id_range> = 0, both $host_{i-1}$ and $host_i$ respond with their id's. A simple solution to this problem would be to field all replies to the initial broadcast message over a time interval and select the highest id from among the replies. Having thus obtained the highest host id used in the network, the new host then takes the next higher id as the tentative id. The code skeleton to acquire a tentative id by this scheme is given in appendix-B.

6.2 Step2: Resolving id clashes

After acquiring a tentative <site_id>, the host should check for any clash with other hosts in the network in the use of this id. The host broadcasts an IS_THERE_OBJECTION message containing the tentative id. Any host whose id clashes with that in the message raises an objection by replying with an OBJECTION message. If an objection to the bid is received indicating that the id is already in use, the host recompiles another id and rechecks for objections. When there is no objection from other hosts after a certain number of broadcast-based probe messages, the host acquires the id.

When more than one host try to establish their site id's at the same time (i.e., when a host sending the IS_THERE_OBJECTION message receives one from *another* host), the protocol requires that the colliding hosts backoff for a random interval of time and try again to acquire an id. This is similar to the CSMA/CD technique used in Ethernet to resolve collisions [21] but applied to a higher level problem.

6.3 Step3: Officialisation of the id

After affirming there is no objection to the id, the site officially announces its entry into the system by broadcasting its <site_id>--><network_interface_id> mapping information. Other nodes may cache this information subject to their cache constraints. Nodes which already have an inconsistent entry for this <site_id> update it with the new mapping. Such a mapping information is fundamental to every message transaction since the physical launching of the packet requires a physical station address to reach the site concerned.

7.0 Machines exiting from the network

There are two ways in which machines may exit from the network:

(i) machine failure (due to a machine crash or a local power failure or a local hardware failure),

(ii) scheduled shutdown of the machine.

In both cases, the high level recovery is as discussed in section 3.2.

In the case of machine failure, the <host_id>--><network_interface_address> mapping information available at the other sites is no longer valid. If a new machine joining the network independently assumes the same id as that of the failed machine, then the inconsistent mapping information at other sites is updated by the broadcast message from the new site.

In the case of a scheduled shutdown, the kernel executes a shutdown protocol whereby it systematically terminates all processes and foreign activations executing on the machine, and invalidates the <host_id>--><network_interface_address> mapping information by broadcasting a HOST_ID_INVALID message.

8.0 Host migration

A related issue is the migration of the host from one physical machine to another. This is functionally different from the process migration problem discussed earlier. Host migration becomes a requirement when machines are to be taken in and out of a network for operational reasons transparent to other hosts in the network. We consider a simple scenario to illustrate the feature. Let H_1, H_2, .. , H_i, .. , H_n be hosts on the network, and suppose H_i is to be taken out of the network for maintenance purposes. A new machine H_i' (identical to H_i) has to be introduced into the network, and all executions on H_i are moved onto H_i'; then H_i is shutdown and removed. From this point onwards, H_i' assumes the identity of H_i for all process level interactions though the underlying <host_id>--><network_interface_address> mapping has changed. Before the underlying mappings are updated at other sites, there is a short-term inconsistency resulting in misdirected messages. Though the recovery protocols on other machines have a built-in host search mechanism to acquire the new mapping information (similar to the process search discussed in section 5.1.1), the migrated host on its part, broadcasts this information so that the interested nodes may cache them to reduce loss of messages.

9.0 Conclusions

We have described four common classes of state inconsistencies to be handled in a LAN-based distributed kernel caused by failures of certain kernel objects, packet losses, machines joining or exiting from the network and process or host migration. Solutions based mostly on the concept of kernel-supported process aliases are outlined.

Because we have assumed an environment that supports migration of processes and hosts, the underlying issues of cache management and object search techniques across the network require efficient solutions. These issues need not arise if the kernel does not support migration; in this case, direct mapping techniques to find the address of the physical network interface can be used.

In most existing kernels [3, 6], object search is used solely to resolve service names to process_id's. In our model of the kernel however, object search is fundamental to the operation of the system; the search is used to resolve state inconsistencies and update the <process_id>--><site_id> and <site_id>--><network-interface_address> mappings.

Though cache management issues are mentioned in the paper, the techniques to manage the cache namely the policy to add/remove entries from the cache, the cache size and their effects on efficiency are not discussed as they are beyond the scope of this paper.

Reliable LAN-based distributed kernels must be able to cope with the large class of errors due to state inconsistency as discussed in the paper.

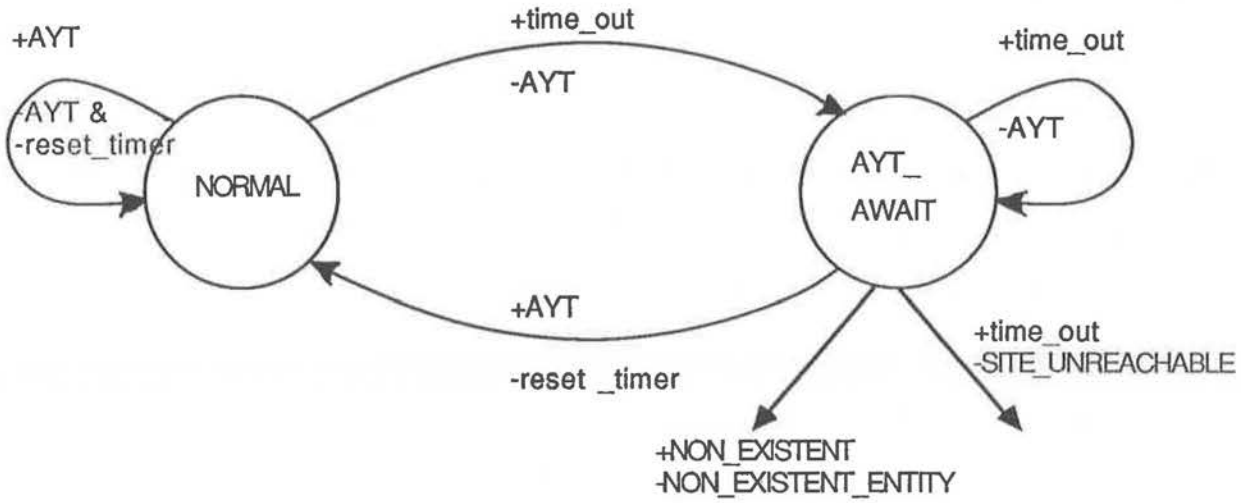


Figure 1. Basic structure of the failure detection protocol

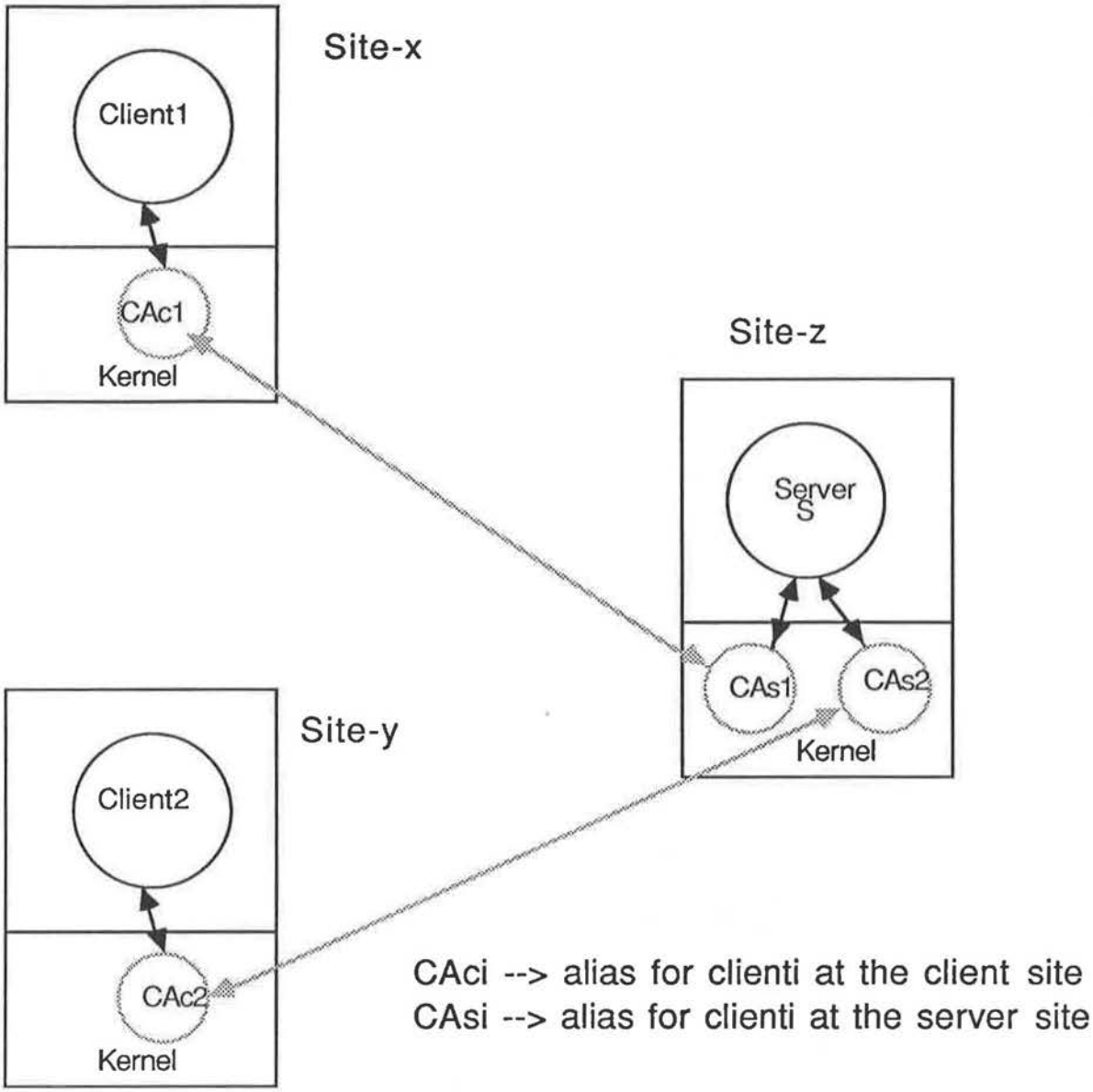
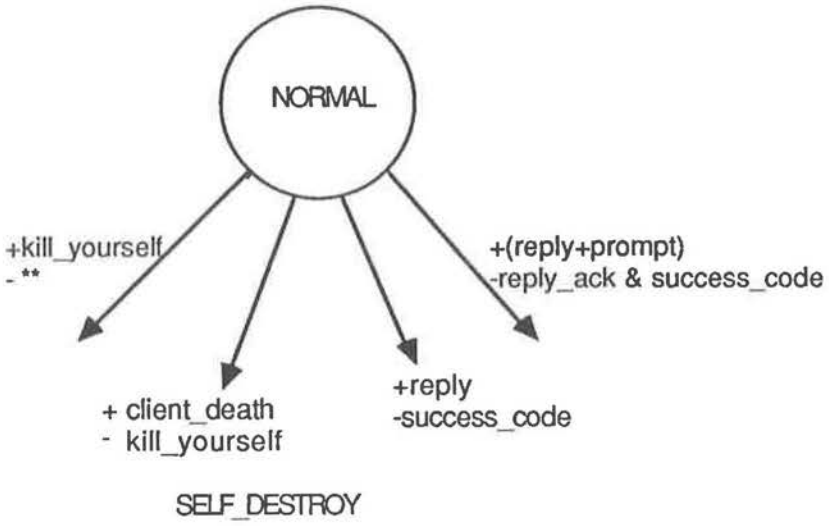
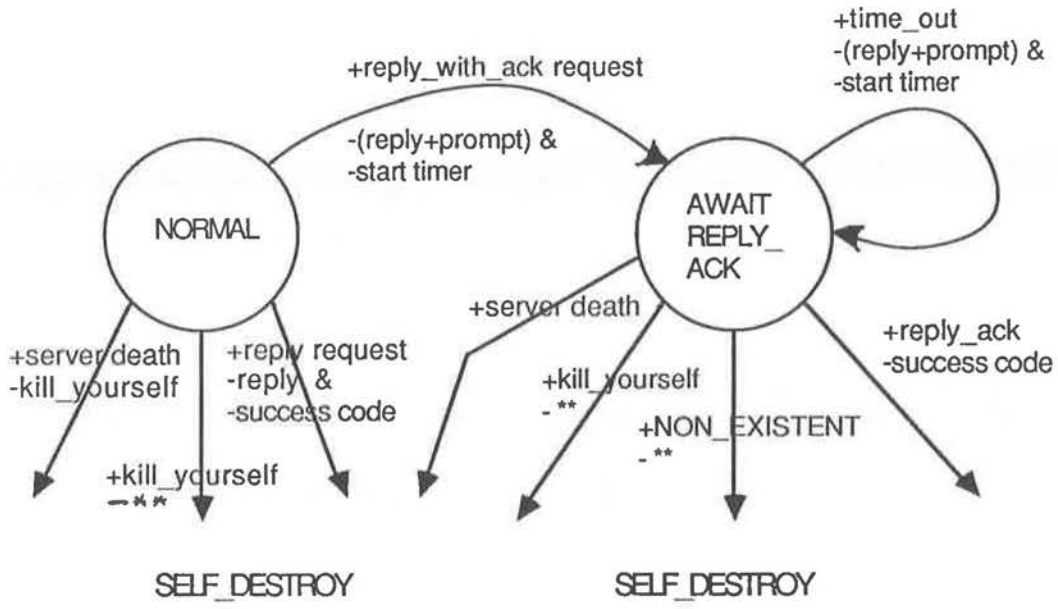


Figure 2. A pictorial representation of the aliases that implement the protocol for the message transaction



Protocol executed by CA_{ci}



Protocol executed by CA_{si}

** -- NON_EXISTENT_PROCESS

Figure 3. Finite State Machine representation of the protocols executed by the client aliases to handle process deaths

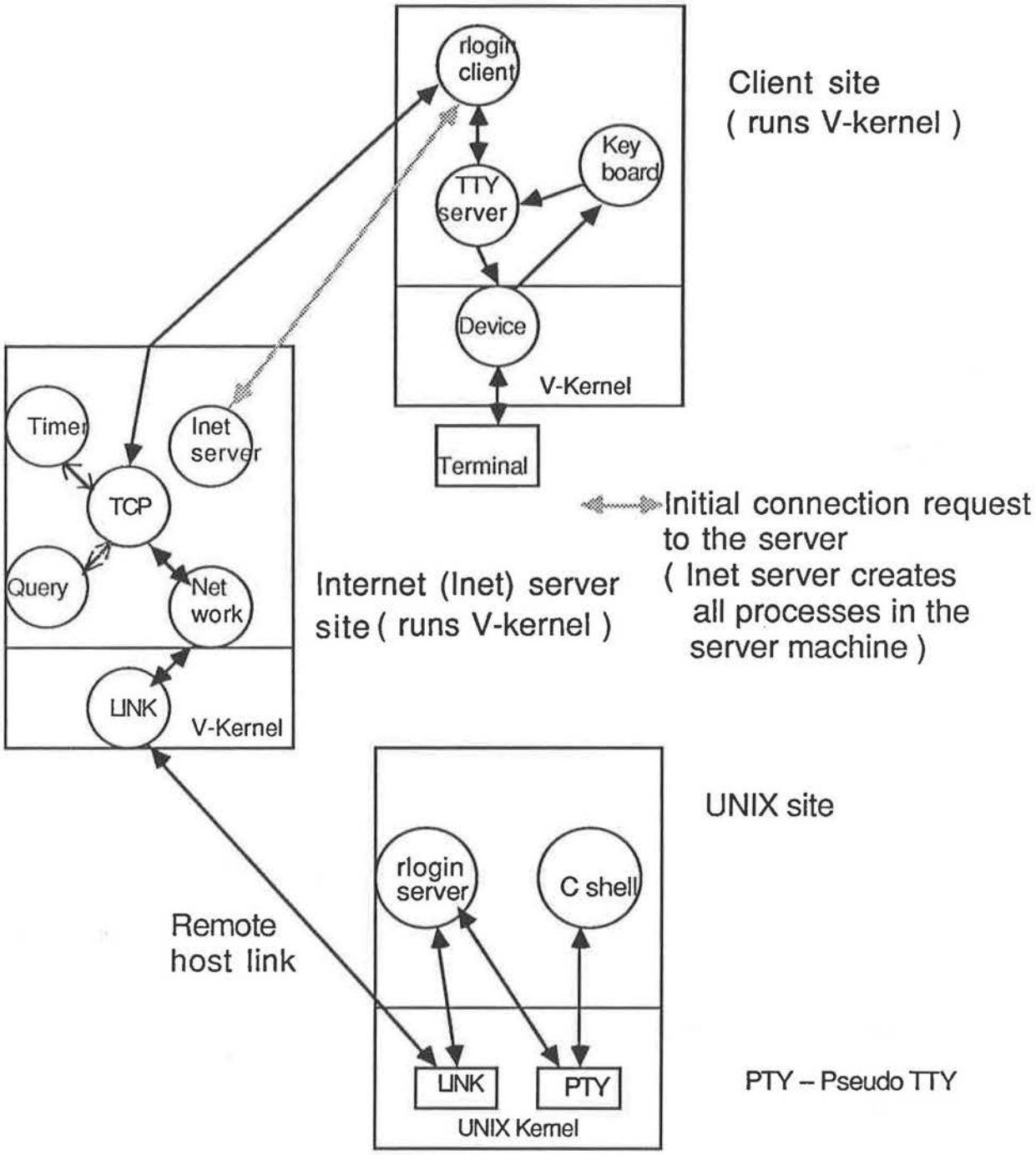


Figure 4. A connection-oriented client-server interaction to create a remote login connection to a UNIX host.

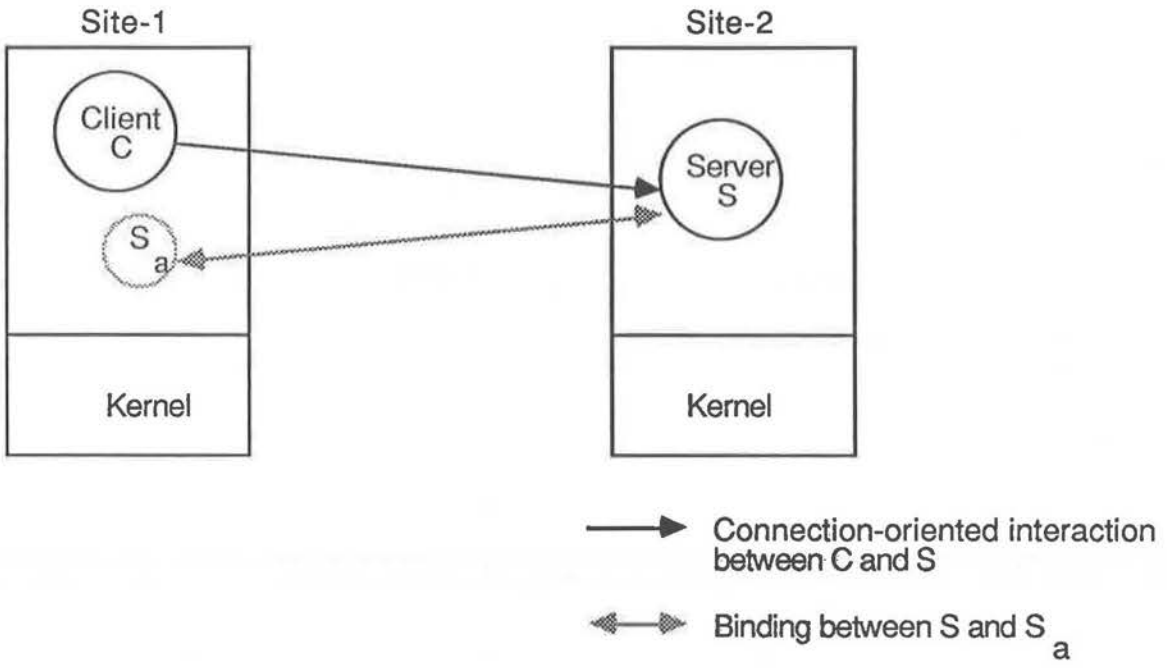


Figure 5. Alias based implementation of the vulture scheme.

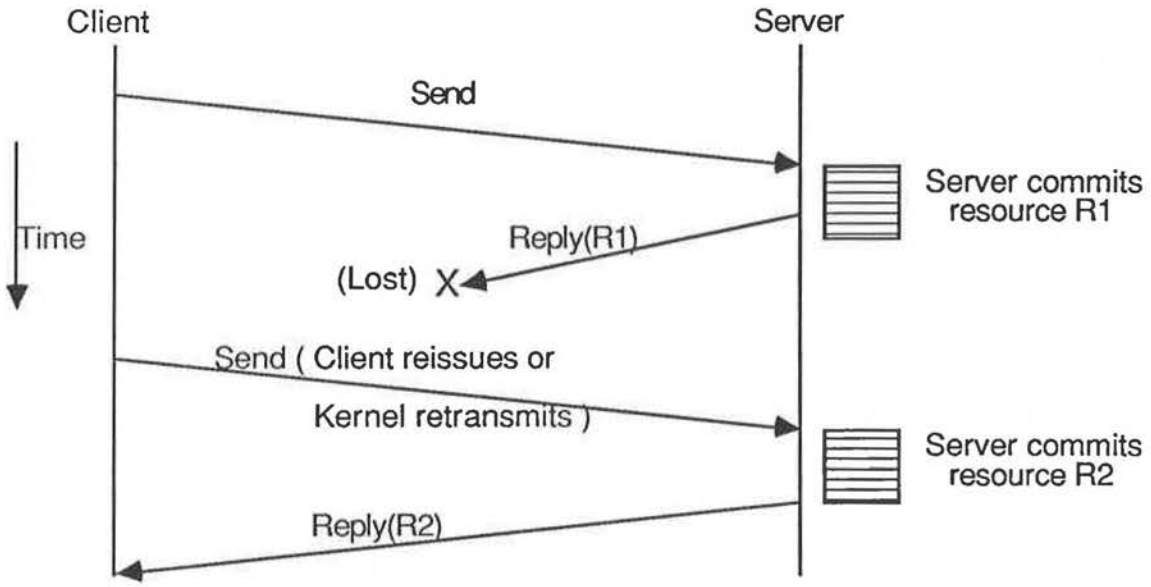
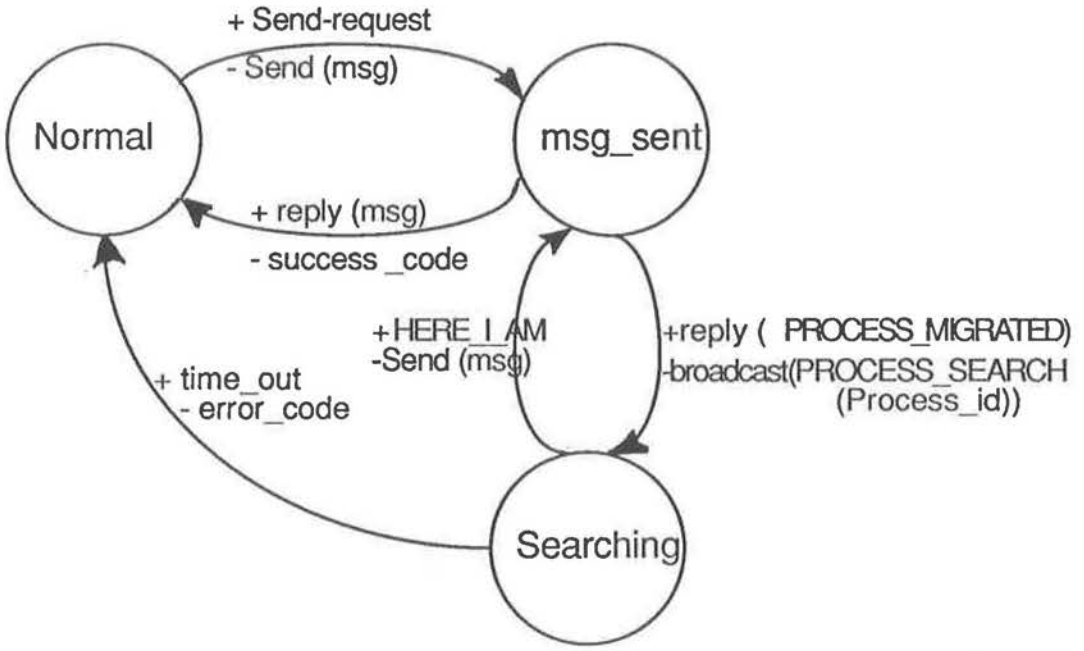
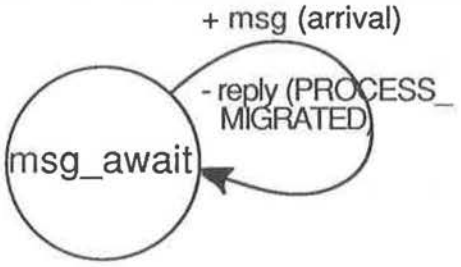


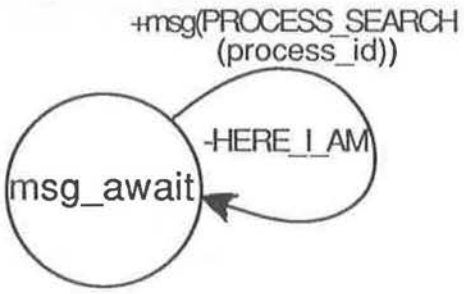
Figure 6. A scenario to illustrate the inconsistency arising due to packet loss.



Protocol implemented at the client site



Protocol implemented by the alias at the original site.



Protocol implemented by the answering alias for the migrated process.

Figure 7. Protocols implemented in the client-driven scheme.

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

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

5. 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.

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

7. 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.

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

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

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

11. 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.

12. L.M.Ni, et al., ***A dynamic process migration algorithm for distributed systems,*** Proc. of the 5th symp. on Distributed Computing Systems (IEEE-CS), pp.539-546, May '85.

13. M.L.Powell and B.P.Miller, ***Process migration in DEMOS/MP,*** Proc. of the 8th symp. on Data Commn., ACM SIGCOMM, Oct.'83, pp.110-119.

14. B.J.Wood, et.al., ***A local area network architecture based message passing operating system concepts,*** Proc. of 7th IEEE Conf. on Computer Networks, Oct.'82, pp.59-69.

15. Y.K.Dalal, ***Use of multiple networks in Xerox's network system,*** IEEE Computer, vol.15, no.10, Oct. '82, pp.82-92.

16. W.M.Gentleman, ***Message passing between sequential processes: the reply primitive and the administrator concept,*** Software practice and experience, vol.11, no.5, May '81, pp.435-466.

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. A.P.Black, ***Supporting distributed applications: experience with Eden,*** Technical report (TR.85-03-02), Dept. of Computer Science, Univ. of Washington, Mar.'85.

19. K.Ravindran and S.T.Chanson, ***On process aliases in distributed kernel design,*** Technical report 85-5, Dept. of Computer Science, Univ. of British Columbia, April '85.

20. S.T.Chanson, K.Ravindran and S.Atkins, *LNTP - an efficient transport protocol for local area networks,* Technical report 85-4, Dept. of Computer Science, Univ. of British Columbia, Feb.'85.

21. J.F.Shoch, et al., *Evolution of the Ethernet local computer network,* IEEE Computer, vol.15, no.8, Aug.'82, pp.10-27.

22. DARPA Internet program protocol specifications, *Transmission control protocol,* RFC 793, Information Sciences Institute, USC, CA, Sept. '81.

23. DARPA Internet program protocol specifications, *Internet protocol,* RFC 791, Information Sciences Institute, USC, CA, Sept. '81.

24. I.W.Cotton, *Technologies for Local Area Computer Networks,* Computer Networks, North-Holland Pub. co., vol.4, '80, pp.197-208.

25. J.A.Stankovic, *A perspective on Distributed Computer Systems,* IEEE Trans. on Computers, vol.C-33, no.12, Dec.'84, pp.1102-1115.

26. D.R.Cheriton, and M.A.Malcolm, *Process identification in THOTH,* Technical Report TR.79-10, Dept. of Computer Science, Univ. of British Columbia, Oct.'79.

27. E.J.Berglund, et al., *V-System reference manual,* Computer Systems laboratory, Dept. of Computer Science and Electrical Engg., Stanford University, Oct.'84.

28. R.Sandberg, et al., *Design and implementation of the SUN Network File System,* Proc. of the Summer USENIX Conference, June '85, pp.119-130.

29. D.R Cheriton, and W.Zwaenopoel, *Distributed process groups in the V-Kernel,* ACM Transactions on Computer Systems, vol.3, no.2, May '85, pp.77-107.

30. L.Svobodova, *File servers for network-based distributed systems,* Proc. of the IEEE, Dec.'84, pp.353-398.

Appendix-A

Binding between a process and its remote alias

When a remote alias P_a is created by a root process P, the kernel at the root machine creates an invisible alias I_{ap1} at the local site. The kernel on the machine where the remote alias executes also creates an invisible alias I_{ap2} at that site. These aliases implement the asynchronous failure detection protocol (see section 2.4) to ascertain the existence of each other (Figure A.1).

When P dies, the kernel destroys I_{ap1} and dispatches an error message KILL_YOURSELF to I_{ap2}; I_{ap2} then destroys P_a and itself. On the other hand, if P_a dies, the kernel destroys I_{ap2} and dispatches the message KILL_YOURSELF to I_{ap1}; I_{ap1} delivers an error message ALIAS_DEAD to P and terminates itself. P detects the death of P_a when it performs a *receive_any* operation, and initiates appropriate recovery.

When $site_1$ fails (thus destroying P and I_{ap1}), I_{ap2} detects this and destroys P_a and itself. When $site_2$ fails (thus destroying P_a and I_{ap2}), I_{ap1} detects this and destroys itself after delivering the message SITE_UNREACHABLE to P to enable recovery. P detects the site/network failure when it performs the *receive_any* operation, and initiates proper recovery.

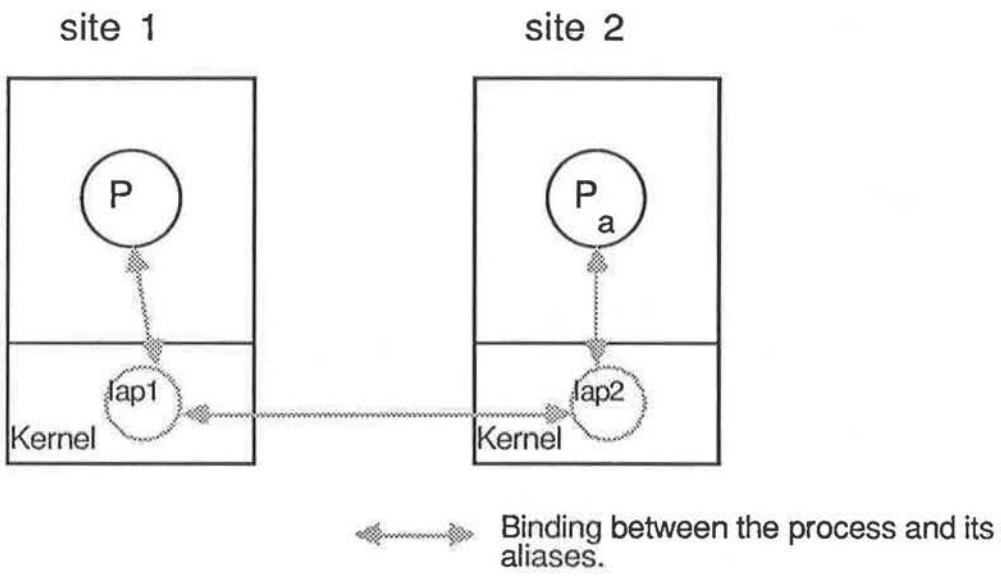


Figure A.1. Alias-based structure to provide binding between a process and its remote alias.

Appendix-B

B.1 Code skeleton executed by the kernel of a new site to acquire a tentative id from the network:

```
Kernel_init()  /* kernel initialisation */
 {
    .
    .
    .
    tentative_id = 0;
    for ( Id_range = 0, seq_no = 0; Id_range <= MAX_ID_SPACE;
                    seq_no++ )
     {
        rebroadcast_count =0;
        do
         {
            msg.type = SUBMIT_YOUR_ID;
            msg.station_adrs = <Network interface address
                                        of this site>;
            msg.brdcst_cnt = rebroadcast_count++;
            msg.seq_no = seq_no;
            msg.Id_range = Id_range;
            broadcast (msg );
            <initiate TIMER>;
            for ( sender = receive ( msg, ANY_PID );
                sender != TIMER; sender = receive ( msg, ANY_PID ) )
                 /* Initially the kernel assigns a logical
                    host id '0' for all local processes */
              if ( sender == NETWORK_RECEIVER )
                 <cache reply>;
         }
        while ( rebroadcast_count <= MAX_RETRIES )
        if ( <cache not empty> )
         {
            highest_id = get_highest_value ( <cache> );
            tentative_id = ( highest_id + 1 ) mod MAX_ID_SPACE;
            break;
         }
        else
         Id_range = ID_INTERVAL * BASE**seq_no;
```

```
                    /* BASE = 10, ID_INTERVAL = 10 ( say ) */
      } /* End of "for" loop */
     If ( tentative_id == 0 ) /* No other site in the network */
          tentative_id = LOWEST_SITE_ID;
     .
     .
} /* End of "Kernel_init" */
```

B.2 The code skeleton executed by the kernel on other sites is as follows:

```
forever
 {
     receive ( msg, NETWORK_RECEIVER );
     if ( msg.type == SUBMIT_YOUR_ID )
      if ( ( self_id + msg.Id_range >= highest_host_id ) )
          {
           response_msg.site_id = self_id;
           response_msg.type = SITE_ID_REPLY;
           response_msg.seq_no = msg.seq_no;
           response_msg.hdr.dstn_station_adrs = msg.station_adrs;
           response_msg.hdr.src_station_adrs = <this station address>;
           <dispatch response_msg to the broadcasting site>;
           <assemble the reply_msg for the NETWORK_RECEIVER>;
          }
     if ( msg.type == OFFICIAL_SITE_ID )
      {
          highest_host_id = msg.site_id;
          <update mapping cache>;
          <assemble the reply_msg for the NETWORK_RECEIVER>;
      }
     <check for other message types>
     .
     .
     reply ( reply_msg, NETWORK_RECEIVER );
 } /* end of "forever" loop */
```

Appendix-C

C.1 Code skeleton to recover from loss of reply packets

The following is a code skeleton to illustrate the client-server i/o protocol to guard against the failure of the *reply* operation due to packet loss.

```
main()  /* Client program */
 {
    .
    .
 tx_cnt = 0;
 repeat
    {
     msg.request = CREATE_INSTANCE;
     msg.attributes = <resource_attributes>;
     msg.transaction_id = <this_transaction_id>;
        /* Assign the same transaction id to all reissued requests */
     status = send ( msg, server_pid );
    }
 until ( ( status == SUCCESS ) || ( ++tx_cnt >= MAX_TRIES ) )
 if ( status == TRANSACTION_FAILURE )
    {
     /* The client chooses not to reissue */
     msg.request = DESTROY_INSTANCE;
     msg.instance_id = NULL;
     msg.transaction_id = <this_transaction_id>;
              /* Assign the same transaction id */
     status = send ( msg, server_pid );
    }
 else
    {
     /* Request successful or other errors like NON_EXISTENT_PROCESS */
     .
     .
     <recover appropriately>
     .
     .
```

```
    }
}    /* End of client's "main" */


main()  /* Server administrator */
 {
    .
    .
    .
  forever
    {
     sender = receive ( msg, ANY_PID );
     if ( msg.request == CREATE_INSTANCE )
        {
         <search instance_descriptor list>;
            /* check for instances already created for this client */
         if ( ( instance_descriptor.client_pid == sender ) &&
           ( instance_descriptor.trans_req_id == msg.transaction_id ) )
            {
                /* duplicate transaction request */
             <discard request>;
             reply_msg.instance_id = instance_descriptor.inst_id;
                /* assemble a reply message with the previous instance id */
            }
         else
            {
                /* No instance was created for this client or
                   those created for this client have different
                   <transaction_id>'s */
             <create a new instance>;
             instance_descriptor.trans_req_id = msg.transaction_id;
             instance_descriptor.inst_id = <new_instance_id>;
             instance_descriptor.client_pid = sender;
             reply_msg.instance_id = instance_descriptor.inst_id;
                /* assemble a reply message with the new instance_id */
            }
        }
     if ( msg.request == DESTROY_INSTANCE )
        {
         if ( msg.instance_id == NULL )
            {  /* Destruction request based on transaction id */
             <search instance_descriptor list keyed on msg.transaction_id>;
             if ( found )
```

```
                    <destroy instance>;
                  <assemble reply_msg>;
                }
            else
                { /* Destruction request based on instance id */
                  .
                  .
                }
          }
        <check for other request types>
            .
            .
            .
        status = reply ( sender, reply_msg );
      }   /* End of "forever" loop */
  }   /* End of server administrator "main" */
```

C.2 Code skeleton to recover from loss of reply_ack packets

The following is a code skeleton to illustrate the client-server i/o protocol to guard against the failure of the *reply_with_ack* operation due to loss of acknowledgement packet.

```
main()  /* Server administartor */
 {
    .
    .
    .
  forever
    {
     sender = receive ( msg, ANY_PID );
     if ( msg.request != CREATE_INSTANCE )
         {
          <search instance_descriptor list>;
             /* check for instances already created for this client */
          if ( ( instance_descriptor.client_pid == sender ) &&
            ( instance_descriptor.inst_id == msg.instance_id ) )
             {
                  /* Comply with the request */
```

```
                <service the request>;
                reply_msg.err_code = SUCCESS;
                    /* assemble a reply message with the success code */
                }
            else
                {
                    /* No named instance available */
                reply_msg.err_code = INVALID_INSTANCE_ID;
                    /* assemble a reply message with the error code */
                }
        }
    if ( msg.request == CREATE_INSTANCE )
        {
         <create a new instance>;
         instance_descriptor.trans_req_id = msg.transaction_id;
         instance_descriptor.inst_id = <new_instance_id>;
         instance_descriptor.client_pid = sender;
         reply_msg.err_code = SUCCESS;
         reply_msg.instance_id = instance_descriptor.inst_id;
            /* assemble a reply message with the new instance id */
        }
    status = reply_with_ack ( sender, reply_msg );
    if ( status == TRANSACTION_FAILURE )
        <destroy the instance that was created for this client>;
}   /* End of "forever" loop */
.
.
.
}   /* End of Server administrator "main" */
```