Host identification in reliable distributed kernels

Samuel T. Chanson & K. Ravindran

February 1986
Technical Report 86-5

ABSTRACT

Acquisition of a host identifier (id) is the first and foremost network level activity initiated by a machine joining a network. It allows the machine to assume an identity in the network and build higher levels of abstraction that may be integrated with those on other machines. In order that the system may work properly, host id's must satisfy certain properties such as uniqueness.

In recent years, distributed systems consisting of workstations connected by a high speed local area network have become popular. Hosts in such systems interact with one another more frequently and in a manner much more complex than is possible in the long haul network environment. The kernels in such systems are called upon to support various inter-host operations, requiring additional properities for host id's. This paper discusses the implications of these properties with respect to distributed kernel reliability. A new scheme to generate and manage host id's satisfying the various properties is also presented. The scheme is distributed and robust, and works even under network partitioning.

Host identifier management in reliable distributed kernels

1. Introduction

In a distributed system consisting of a collection of interconnected machines, a host is an abstract entity associated with an instantiation of a machine. A host is bound to an identifier (id) which allows it to interact and integrate with other hosts in the system to build higher levels of abstractions. So acquisition of a host identifier is the first and foremost network level activity initiated by a machine joining the distributed system.

In currently popular local area network (LAN) based distributed systems (typically consisting of workstations)[10], hosts interact with one another more frequently and in a manner much more complex than is possible in the long haul network environment. The kernels in such systems are called upon to support various inter-host operations, requiring various properties for host ids. Some of the properties such as uniqueness of the host id are mandatory and some are desirable. Whether a property is mandatory or desirable depends on the functionality required of the kernel. For example, robust failure recovery requires guaranteed non-reusability of the host id. As another example, multiple logical identifiers for a given machine may be used to achieve pre-emptable program migration across machines[12]. Also, id generation and management should be distributed and robust for reliability reasons.

Though some of the id management issues have been touched upon in several experimental systems and briefly mentioned in some papers[4,5,13,18], there has been no focussed attention to this layer of the distributed kernel in the open literature. Uniqueness of the host id is guaranteed in the schemes used in most of these systems. However, other properties are either not guaranteed or not supported for the sake of simplicity and efficiency in the generation and management of the ids and their contexts. In this paper, we first discuss the implications of these properties with respect to reliable LAN-based distributed systems. We then discuss various techniques to generate and manage host id's satisfying the different properties.

2. Properties of host id's

In the discussion that follows, the terms "host" and "site" are used interchangeably.

2.1 Uniqueness

This property ensures that each host in the system has a unique id with which it may be addressed and accessed. In many systems, the host id is used to generate id's for higher level objects to guarantee system-wide uniqueness[30]. For example, in the V–Kernel[13], each process is globally identified by the pair

<host_id, local_pid>

where <host_id> is the id of the host on which the process is created and <local_pid> is the id of the process which is locally unique within the host[11]. Other higher level objects are constructed in terms of these processes. For example, a file instance is characterised by

<file_instance_id> = <file_client_id, file_worker_id>,

and a network connection instance is specified by

<connection_id> = <network_worker_id, network_client_id>.

The uniqueness of these high level objects is guaranteed by that of the process id's which in turn depends on the uniqueness of the host id's. Issues such as non-determinism and security arising from the non-uniqueness of the id's are discussed elsewhere[1].

2.2 Non-reusability

Reusability of a host id refers to the reassignment of the id that was previously used by a site to its reincarnated version or to another site. This might cause state inconsistencies resulting in misdirected and misdelivered messages in certain situations, particularly in large distributed systems. An anthropomorphic example is the reassignment of a telephone number recently used by an individual to another, resulting in misdirected calls.

To analyse the implications, consider the following scenario (see Figure 1): let C_1 be a client process on $host_1$ that logs onto a remote host through the TCP/IP (Transmission Control Protocol/Internet Protocol)[8] Internet worker process T_{Inet} which runs on $host_2$ and allocates a TCP port for the connection. Suppose the Internet server site suffered a momentary failure and

then came up within a short interval (we assume that the pre-failure state is completely lost as a result of machine failure), it is quite likely that C_1 has not noticed the failure. Since the values of the <local_pid> field of process id's begin a fresh cycle, the <local_pid> previously used by T_{Inet} before site failure could have been reassigned to another process, P_{New}, say, in the reincarnated site. If the server host ($host_2$) reuses the same host id after the failure, then P_{New} incorrectly becomes an eligible recipient of messages from C_1. Error may arise in two cases:

Case 1. P_{New} is supporting a remote login connection for some other client C_2 on $host_3$.

If P_{New} is waiting for messages specifically from C_2, the message from C_1 is queued up for P_{New} but may never be read. However, since P_{New} is alive and accessible, C_1 thinks that the server is working on other messages. Recovery is possible by high level timeouts in C_1 whereby C_1 may abandon its end of the connection. If P_{New} is waiting for messages from non-specific processes (e.g., P_{New} could be supporting remote logins from multiple clients), the message from C_1 is delivered to it. If P_{New} does not have adequate filtering and checking mechanisms, the messages from C_1 will confuse P_{New} which may discard valid messages from C_2, and may even forcibly close its on-going connection with C_2. This anamoly is similar to the undetected packet misdelivery[17] that arises due to clash of high level identifiers.

Case 2. P_{New} is performing a completely different function.

Since the process id of the message originator (C_1) as well as the message contents are meaningless to P_{New}, P_{New} simply discards the message. To enable recovery by C_1, P_{New} may reply with an error message.

Other types of anamolies may occur under different scenarios due to the inability to distinguish between executions on a machine prior to and after its failure. For example, undetected mis-pairing of peers may occur in TCP connections in UNIX 4.2 BSD where the internet host address component of the TCP socket address remains unchanged after machine failure[18]. As another example, if a client crashes during a remote procedure call (RPC), its subsequent reincarnation might give rise to crash orphans[21].

A guaranteed non-reusability of host ids and a high level filtering based on them will prevent such anamolies. In the example above (Figure 1), the kernel will fail any attempt by C_1 to interact with P_{New} with an error message NON_EXISTENT_PROCESS. In response to this

low level error message, C_1 may destroy its end of the connection.

2.3 Independence from the underlying network

Network-independent host identification may be characterised by two orthogonal features:

(i) Network-independent syntactical representation of the host id.

An analogy drawn from programming languages is the use of symbolic strings as variable names that are syntactically independent of their physical addresses in the program.

(ii) Dynamic binding of the host id to the network address.

This property ensures that the host id is resolved to the underlying network address only at the last moment (when a packet has to be launched), and that there is no static binding between host id's and the underlying network. An analogy, again drawn from programming languages, is the dynamic binding of variable names for queue and list data structures to memory locations at run-time.

Property (i) deals with the syntactic structure of the host id, and may be easily provided by simple lexical mapping techniques. It allows a host to be addressed in a simple way and to retain the same name even when accessed from networks with different technology. This property is not necessary for primitive kernels. Property (ii) deals with the dynamic, context-dependent translation of the host id into the associated network address. The level of complexity depends on a variety of factors such as the size of the network and the context used for translation. This property is mandatory for significant extensions to kernel functionalities, which are briefly described below:

Host migration. Host migration refers to the transparent transfer of the various executions on a host from one physical machine to another with the latter assuming the role of the former. Host migration becomes a requirement when machines are to be taken in and out of a distributed system for operational reasons transparent to other hosts in the system. We consider a simple scenario to illustrate the feature. Let M_1, M_2, .. , M_i, .. , M_n be machines on the system with host id's H_1,H_2,..,H_i,..,H_n respectively. Suppose M_i is to be taken out of the system for maintenance purposes. A new machine M_i' (identical to M_i) has to be introduced into the

system, and all executions on M_i are moved onto M_i'; then M_i is shutdown and removed. From this point onwards, M_i' assumes the identity H_i for all process level interactions. However the underlying <network_address> used by H_i has changed. This calls for a dynamic binding of the host id to the associated network station address, and a network-independent syntactic representation of the host id.

Network fault tolerance. One of the schemes to achieve fault tolerance from network failures is to provide multiple redundant transport media which may be of different but complementary technologies, such as the hybrid LAN, as the underlying communication backbone[22]. In such an architecture, the packet routing layer of the network selects the path the packet is to travel to its destination based on the current state of the network. In order to provide this elegantly and transparently to higher level software, the host id should be logically independent of the underlying network topology, architecture and/or the redundancy. Such an independence should be in terms of both the syntactic representation as well as dynamic binding of the host id to the appropriate network.

Interconnected LANs. To provide a uniform distribution of objects across interconnected LANs, the host identification scheme should be uniform throughout the interconnected system. A transparency requirement is that the link level interconnection structure be invisible at the process abstraction layer. Since process id's are the only means of naming the communicants at this layer and host id's make up part of the process id's, any dependence of the host id's on the underlying network will violate the syntactic transparency requirement. For fixed topology interconnection, network-independent representation is adequate with the bindings between host id's and the network statically built into the kernel. To support flexible and dynamic interconnection, the bindings should be dynamic as well.

Kernel portability. The portability of the kernel across different LAN technologies with different addressing schemes and formats requires that host identification be independent of the underlying LAN technology. Violation of this requirement will impose network-dependent syntactic structures into the kernel design, hampering its portability. However, the binding between host id's and the network may be static or dynamic.

Dynamic change of network addresses. It is necessary sometimes to dynamically reassign the station address of the network interface transparently without affecting the ongoing executions on the host. It may arise for troubleshooting network interfaces, for example. Most of the commercially available network interfaces such as the 3COM's Ethernet and Pronet's token ring interfaces provide this feature as an option for low level reconfiguration[23,24]. The process abstraction should be independent of such low level activities by making host identification independent of the network both in terms of syntactic representation and dynamic binding of host id's to the network. An example is the Address Resolution Protocol[19] used in UNIX 4.2 BSD implementations to bind the internet host addresses to Ethernet addresses.

3. Some existing host identification schemes

A simple scheme to generate host id's which is also efficient to use is the assignment of the network station address as the host id. In most of the commercially available network interfaces, the default station address (which is systemwide unique) is hardwired into the network interface[23,24]. In other systems, the default station address is hardwired into the CPU system board, and the network interface initialises itself with this address on startup[20]. In both types of hardware, the software may optionally change the station address for a variety of reasons. The station address currently being used is acessible to software from the network hardware registers. Thus host id generation is a simple matter of reading this address from the hardware. Since the host id is also the physical address of the station, the mapping is trivial. In other words, the context is statically built into the identification mechanism. As the station addresses are guaranteed to be systemwide unique, so are the host id's. However the scheme has the following disadvantages:

(i) It does not provide non-reusability of the host id across machine failures,

(ii) It does not provide independence from the underlying network both in terms of the syntactic representation of the host id's as well as the static nature of the bindings between host id's and the network. This makes high level functionalities harder to accomplish[31] such as transparent host level reconfiguration.

In an extended model of the above scheme, each kernel maintains a table of network station addresses of all the machines in the network. When a kernel comes up, it reads the hardware station address of the local network interface and searches the address table for a match. When a match is found, the index into the table is used as the host id for the machine. The context for a host id is obtained by using the id as an index into the station address table to fetch the corresponding station address for launching packets. The V–System[13] originally used this approach for generating and managing host id's on an Ethernet LAN. The scheme guarantees uniqueness of the id and retains an efficient structure for implementation. Though the syntactic representation of the id is independent of the underlying network, the binding between the two levels of identifiers is static, precluding host level reconfiguration. To overcome this limitation and to support program migration, the new scheme requires the machine joining the network to use a dynamic id assignment protocol whereby it generates a tentative id using random numbers and follows it up with a mandatory check for id clashes to avoid picking an id currently being used by some other host[14]. However, neither techniques guarantee non-reusability across machine failures.

Some message-passing models use incarnation id's to guarantee the non-reusability of the host id's across host crashes[4,5]. In this scheme, a host id is represented by

<incarnation_id, host_id>

where <incarnation_id> is used to identify the particular incarnation of the kernel. It is usually stored in some stable local storage[25] (i.e., disk), and is guaranteed to be unique across machine failures. Thus every process created on a machine has the machine's incarnation id as part of its process id (<host_id> is presumeably managed by a static assignment scheme). Reincarnation numbers are used in a slightly different way in RPC protocols[2] to guard against undetected inconsistency in the client's cache due to server failure and its subsequent reincarnation. A server uses reincarnation numbers based on time stamps as part of the interface it exports to clients. The client presents this reincarnation number in every RPC request to the server which validates the request based on its local reincarnation number. When a request is rejected due to mismatch of the reincarnation numbers, the client's run-time system rebinds the interface it imports by contacting a binding agent. Schemes based on reincarnation id's suffer from the

following limitations:

(i) Two orthogonal techniques are needed, one to provide non-reusability of the id, and the other to provide independence from the network.

(ii) In systems consisting of diskless workstations, the <incarnation_id> is to be stored across the network on a remote stable storage. The code id based version identification scheme used by the boot server in the UNIVERSE system to detect remote system crashes[26] uses this approach. The disadvantage of this approach is the single-point failure of the remote repository of incarnation id's.

Several centralised dynamic schemes to assign station addresses have recently been proposed[16,28] that may be extended to guarantee non-reusability of such addresses. Besides being vulnerable to single point failures, most attempt to solve the address assignment problem in isolation from high level issues. The centralised addresss manager in one scheme[16] asynchronously and periodically probes all stations in the network to ascertain they are alive and reclaim unused addresses. We observe that the asynchronous protocols are not really needed when the address management issues are tackled from an operating system point of view since protocols to detect such low level inconsistencies may instead be integrated into the IPC mechanisms.

4. A new model of host identification

We first describe a unified, robust and distributed approach for the dynamic allocation of host id's that satisfies the various properties discussed earlier. We then present a scheme for the dynamic binding of host id's to the network. A broadcast capability in the underlying LAN is assumed. For simplicity, a single LAN case is presented; however, the scheme is adaptable to other LAN architectures such as interconnected LANs. Also, our model resorts to repeated broadcast rather than reliable broadcast in all broadcast-based protocols since any reliability mechanism built into such protocols imposes severe penalty on efficiency[3].

4.1 An overview of the scheme

The kernel on each machine may be considered as a server process. The collection of these kernels constitutes a well-known server group[27] KRNL_GRP; the group is statically bound to the broadcast address of the underlying network. Then, in terms of the distributed server model[6], a kernel process that gets instantiated on a new machine integrates into KRNL_GRP through the following logically distinct phases -- i) a name resolution phase that allows the kernel to locate KRNL_GRP, ii) a subscription phase that allows the kernel to broadcast messages to KRNL_GRP and to receive messages addressed to its network addresss (because it does not have a host id yet) as well as those addressed to KRNL_GRP, and iii) a state acquisition phase by which the kernel acquires its host id from other members of KRNL_GRP in the system so that it can logically merge into the system. The completion of the first two phases is made implicit by hard-wiring the membership in KRNL_GRP into the local instantiation of the kernel as it is brought up on a machine. The protocols employed by the kernel for the third phase (host id acquisition) are dependent on the constraints, namely, network independence and non-reusability of host id's imposed by the kernel functionalities described earlier. We now give an *abstract* description of how the third phase is carried out.

The global name space for host id's consists of a range of unsigned numbers between MIN_ID and MAX_ID which are the lowest and the highest values respectively of host id allowed in the system. Every machine joining the system is assigned an id from this name space. The id allocation is based on the chronological order of the times at which machines join the system. Thus a machine joining at time t_j is allocated an id whose numerical value is greater than that of a machine that joined at an earlier time t_i ($t_j > t_i$). The system maintains a *distributed state variable* <highest_host_id> which represents the highest host id that has been assigned from the global name space. The acquisition of a host id by a kernel may then be abstracted as `reading' this variable, using the next higher value as the new host id, and `incrementing' this variable. Since the variable is distributed, the `read' and the `write' operations on it must encapsulate mechanisms to guard against inconsistencies that may arise due to message losses, machine and network failures. This abstraction fits into Cheriton's model of problem-oriented shared memory[29] in which the `fetch' (read) and the `store' (write) operations defined on a memory depend on the consistency constraints imposed by the particular problem on the memory. In terms of this model, the protocols we describe in the following

sections are in fact a problem-oriented realization of the `fetch´ and the `store´ operations on the distributed state variable <highest_host_id>.

Section 5 deals with id generation, section 6 describes the id resolution scheme, and section 7 discusses how the scheme handles the problems of network partitioning.

5. Generation of host id's

The procedural realization of the distributed state variable contains an instance of this variable maintained by each host in the system. This instance represents the host's knowledge of the highest host id across the entire system, and is updated by broadcast messages from a joining machine. Each host thus maintains a state-pair

<self_id, highest_host_id>.

where <self_id> is the id of the host and <highest_host_id> represents the local instance of the distributed state variable maintained by the host. Then for any host,

(MIN_ID ≤ self_id ≤ highest_host_id ≤ MAX_ID).

Id generation is done in three stages (see Figure 2 for the finite state machine (FSM) representation of the protocol):

5.1 Step1: Acquiring a tentative id

When a new machine wishes to join the system, it first acquires a tentative id *from other hosts* in the system to be used as a bid in the system for use as its <host_id>. To do so, it broadcasts one or more search messages looking for the highest host id that has been assigned so far (note that this may be different from the highest active host id). During id acquisition, a machine uses its network address as the key which may be used by other hosts to reach it. The machine specifies two integers <Id_range1> and <Id_range2> in the search message REQUEST_HOST_ID(Id_range1, Id_range2). These integers satisfy the following condition,

(0 ≤ Id_range1 ≤ Id_range2 ≤ (MAX_ID–MIN_ID)).

The message requires all hosts with

$$(\text{highest\_host\_id} - \text{Id\_range2}) \leq \text{self\_id} \leq (\text{highest\_host\_id} - \text{Id\_range1})$$

to send their respective local values of the state variable <highest_host_id> to the broadcasting machine. The latter may then filter the highest id from among the replies to be used to generate the tentative host id. <Id_range1> and <Id_range2> may be thought of as specifying a *polling window* that qualifies a selected set of hosts to respond to a particular search message (see Figure 3). The size of this window is

$$[<\text{Id\_range2}> - <\text{Id\_range1}> + 1],$$

and the range of the host id space polled is

$$\{(\text{highest\_host\_id} - \text{Id\_range2}), (\text{highest\_host\_id} - \text{Id\_range1})\}$$

In the simplest scheme, the joining machine initially starts with Id_range1=0 and Id_range2=0 implying a window of size 1. On receiving this message, the host whose <self_id> equals <highest_host_id> responds with its value of <highest_host_id>. The initial search message may go unanswered if the machine that wishes to join is the first one in the system, or if none of the hosts have their <self_id> equal to the <highest_host_id> (the latter is possible if the host holding the <highest_host_id> as <self_id> has failed). Failure to get any response results in the machine rebroadcasting the search message with a different polling window until the entire host id space (MAX_ID - MIN_ID) is polled or sufficient number of responses are received, whichever occurs earlier. If there is no response even after polling the entire host id space, the machine assumes that it is the first one joining the system, and takes on MIN_ID as the tentative id. The polling window is a control parameter of the protocol. A typical choice is to increase the window size logarithmically and slide it along the host id space. The range of id space covered by each window should be mutually exclusive to avoid receiving a reply more than once. (See Appendix B for detailed discussion).

Having thus obtained the highest host id that has been assigned so far in the system, the new machine then takes the next higher id as the tentative id. Note that the protocol strives to avoid id clashes during generation of tentative id's.

5.1.1 Guarding against message losses. The broadcast protocols that are integrated into the id generation mechanism do not guarantee reliable message delivery. Messages may be lost due to i) buffer overflow at the other active hosts in the system which may lead to inconsistencies in the respective local instances of <highest_host_id>, and ii) the inability of the network interface at the joining machine to handle back-to-back reply packets[27] which may result in the machine choosing an id with insufficient information. Due to this inherent unreliability, the selected tentative id may be incorrect.

As an illustration, consider an example with three hosts H_1, H_2 and H_3 in the system. Let 8, 9 and 10 be their respective host id's, with 10 being the highest host id in the system. Let the state-pair maintained by each of the hosts be denoted by $H_1 \rightarrow \{8,10\}$, $H_2 \rightarrow \{9,10\}$ and $H_3 \rightarrow \{10,10\}$, i.e., the initial state is consistent. The following illustrates typical error situations when a new machine attempts to join the system:

Case 1. More than one host think they are holding the highest host id in the system.

This error situation may arise as follows: suppose a new machine (host H_4) joined the system, assuming a host id of 11. As soon as the id was acquired, H_4 broadcast this information to the other hosts. If only H_1 and H_2 heard the message, then the updated state entries are $H_1 \rightarrow \{8,11\}$, $H_2 \rightarrow \{9,11\}$, $H_3 \rightarrow \{10,10\}$ and $H_4 \rightarrow \{11,11\}$. Thus both hosts H_3 and H_4 believe they are holding the highest host id. When a new machine attempts to join the system by broadcasting an initial search message with <Id_range1> = <Id_range2> = 0, both H_3 and H_4 respond with their local values of <highest_host_id>, namely, 10 and 11.

Case 2. The <highest_host_id> value received by a joining machine is not the highest host id that has been allocated.

Suppose two new machines joined the system, and subsequently failed independently. Suppose further that H_1 and H_2 heard the host id allocation messages but H_3 did not. Then the updated state entries are $H_1 \rightarrow \{8,12\}$, $H_2 \rightarrow \{9,12\}$, $H_3 \rightarrow \{10,10\}$. When another new machine attempts to join the system and searches the system with a window of size 1, H_3 responds with the value 10 as the highest host id when the true highest host id is 12.

As another example, consider hosts H_1, H_2 and H_3 with state-pairs $H_1 \rightarrow \{8,10\}$, $H_2 \rightarrow \{9,10\}$ and $H_3 \rightarrow \{10,11\}$ in a dynamic state of the system. When a new machine probes the system

with a window of size ≥ 2, all the hosts respond to the probe with their respective <highest_host_id> values. However, the sequence in which their responses arrive at the joining machine is non-deterministic. Suppose the response from H_3 is discarded due to buffer overflow. Then the joining machine assumes 10 as the highest host id that has been assigned in the system whereas the true highest host id is 11.

The id generation protocol should guard itself against such inconsistencies. A solution to the problem manifests in two steps:

(i) Field a certain number of replies by polling over a certain range of the host id space before selection of the highest host id from among the replies received. The number of replies received and the range of the host id space polled so far also form control parameters of the protocol. See appendix-B for a discussion on the choice of these parameters.

(ii) Resort to a clash resolution protocol as described in the following sections.

5.2 Step2: Resolving id clashes

After acquiring a tentative <host_id>, the machine broadcasts an IS_THERE_OBJECTION message containing the tentative id. Any host whose id either clashes with or is higher than 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 has already been assigned, the machine recompiles another tentative id and rechecks for objections. When there is no objection from other hosts after a certain number of broadcast-based probe messages, the machine acquires the id.

5.3 Step3: Officialisation of the id

After affirming there is no objection to the id, the kernel initialises its local instance of <highest_host_id> to the acquired id. It then announces its entry into the system by broadcasting its <host_id> value in an OFFICIAL_HOST_ID message, and thus becomes an official host. Each host which receives this message updates its local value of the state variable <highest_host_id>. The joining host waits for a predetermined short period of time before engaging in inter-host activities to ensure the other hosts have time to receive and process the

message.

The host id's thus generated are independent of the network, and with a high degree of probability, guaranteed to be unique and non-reusable. The code skeleton to implement the scheme is given in appendix-A.

5.4 Collision resolution

A collision occurs when two or more machines try to establish their host id's at the same time (i.e., one or more machines have initiated procedures to acquire their tentative id's before a machine has completed officialising its host id). This is the case for example when a machine sending the IS_THERE_OBJECTION message receives one from *another* machine. If a joining machine detects a collision during the acquisition of tentative id, the protocol requires that it backs off for a random interval of time and tries again. If it detects a collision after it has acquired a tentative id but before officialised it, it sends a DEFER_HOST_ID message to the colliding machine advising the latter to back off as before. The back-off method is similar to the CSMA/CD technique used in Ethernet to resolve collisions[7] but applied to a higher level problem.

Collisions arising due to more than one machine trying to join the system *independently* at the same time are rare and usually resolvable in a few (1 or 2) retries (see section 5.5 for simulation results). However, a single external event such as an electrical power bump may cause a large number of machines to fail, and when they come up as the power is restored and scramble for a host id to join the system, they may collide with one another repeatedly causing a *collision surge*. This surge may lead to a situation where most of the machines are unable to join the system because of the congestion experienced in accessing the <highest_host_id> state variable.

A mechanism to control this congestion may be built into two stages of the id acquisition protocol:

- A machine, on power up, resorts to *collision avoidance* by waiting for a random interval of time before initiating the protocol. The wait time may range between, say, 0 to 200 msecs in steps given by the local clock resolution (~10 msecs). Because of the induced randomness

in the waiting time before the machine initiates the protocol, the technique eases (but does not eliminate) a posssible congestion due to a collision surge if one is in the offing. However, since the machine is not aware of a possible collision surge apriori, the technique increases the delay in acquiring an id in the normal case when there is no surge, but such an increased delay due to the initial wait time may still be within acceptable limits (see section 5.5 for simulation results).

•

As discussed earlier, when a machine experiences a collision, it backs off for a random interval of time, and tries to access the variable again. Because of collision surge, a machine may be unable to acquire an id even after a maximum number (MAX_BCKOFF) of retries (say ~ 5). Then as part of the recovery, the machine temporarily suspends its attempt to join and passively listens to on-going attempts by other machines (by tuning onto the network's broadcast address) until a few machines are able to join the system or a time out occurs, whichever is earlier. Then it again tries to acquire the id as before. To minimise the possibility of lock-step collisions, the back-off interval as well as MAX_BCKOFF are randomly chosen by the machine.

5.5 Simulated behaviour of the host id generation scheme

The protocol behaviour has been studied under a simulated dynamic environment in which a varying number of machines join the system, operate for a while and exit. The aim of the study was to assess how well the protocol enforces the properties discussed in section 2. Network independence is a qualitative parameter that is implicit in the binding mechanisms, and hence was not part of the study. The quantitative parameters were the probability with which the uniqueness and the non-reusability of the host id's were violated.

5.5.1 Simulation parameters. The controllable parameters were the number of machines in the system, their failure rates and the message loss probability. The simulated environment consisted of i) 20 machines, and ii) 100 machines in the system. Mean running time of the machines was 15 minutes with a variance of 10 minutes. Mean machine down time was 10 minutes with a variance of 9 minutes. (These values are not meant to be realistic, but are designed to test the

robustness of the scheme under strenuous conditions to provide `worst-case' results.) The messages IS_THERE_OBJECTION and OFFICIAL_HOST_ID were broadcast twice, the second one after a 100-millisecond (ms) wait. Message transmission time (including propagation delay) was assumed to be 2 ms whereas message processing time at a host (excluding queue wait time) was taken to be 4 ms (these are approximated from measured values on a network of SUN workstations (model 2) supported by TCP/IP on top of Ethernet). The number of effective replies used by a joining machine to make a decision ($N_{replies}$ in appendix-B) was taken to be 4.

5.5.2 Simulation results. A total of 50,000 samples (i.e., 50,000 host id generations) were taken for each simulation run. Each run was repeated many times to increase the statistical accuracy. The results are summarized in Table 1. We also found that the effect of channel errors on the performance of the protocol was found to be negligible with simulated packet error rates of 10^{-3} and 10^{-4} (in a LAN environment, packet error rates rarely exceed 10^{-4}) 9. The probability of reusing an id as a function of the message loss probability is given by the graph in Figure 4 (both for 100 machines and for 20 machines in the system). Note that when the message loss probability is below 0.6 when there are 100 machines in the system (0.5 for the case of 20 machines), the probability of a host acquiring an id that has been used before is practically zero. Systems where the message loss probability exceeds 0.3 are really too noisy to be usable anyway, and are rare in LAN environments. Also, increasing the number of repeated broadcasts from two to some higher value will lower the probability of acquiring a host id that has been used before.

It is interesting to observe that the probability of acquiring a host id that has been allocated before (whether being used currently or not) decreases as the number of active hosts in the system increases. This is because the probability of losing the state (<highest_host_id>) due to the failure of all the machines that hold the correct value of the state or choosing an incorrect state value from among the replies decreases as the number of hosts in the system increases. This adds weight to the protocol since the larger the network, the more important it is to guarantee the host id properties discussed in section 2. Our simulation results show that the probability of any of these properties being violated is practically zero for typical networks of 10 or more machines. The number of message exchanges required to acquire an id are within reasonable limits. More importantly, the CPU cycles a joining machine consumes on an active host in the system is low (between 10 to 20 msecs on a SUN-2 workstation). This is due, in part, to

the fact that an active host does not maintain any state information pertaining to a joining machine.

6. Dynamic resolution of the host id's

Each kernel maintains a mapping of the form

<host_id>→<network_address>

in its internal cache. Such a mapping information is fundamental to every message transaction since the physical launching of the packet requires the destination station address.

A cache entry may be in one of two states, namely i) SEARCHING, indicating that a search has been issued for the host with identifier <host_id> and <network_address> is not yet valid, and ii) VALID, indicating it is a valid entry as known to the host. The cache may be updated when machines join or exit from the system, and during IPC activities. This distributed cache forms the basic context for the identification mechanism.

Due to limitations on cache size, the cache on each host may describe only a subset of the other hosts in the system. An anthropomorhic example is the caching of selected telephone numbers in a phone book which may represent only a subset of the entire telephone directory. Ideally, the mapping entries in all the caches should be consistent for correct global behaviour of the system. However, practical achievement of global consistency of the distributed cache is very difficult. Thus localised corrective measures on detecting cache inconsistencies are needed to keep the system operational.

A host that has just acquired its host id broadcasts the id and its <network_address> in the OFFICIAL_SITE_ID message. Other hosts may cache this information subject to their cache constraints. Hosts which have an inconsistent entry for this <host_id> update it with the new mapping. To invalidate a <host_id>→<network_address> mapping (for scheduled shut-down of a host, for example), the kernel broadcasts a HOST_ID_INVALID message. However, because of possible message losses, host crashes and network partitioning, inconsistent mapping information may still exist.

When an IPC is initiated, the host abstraction layer must translate the host id into the corresponding network address in order to launch the packet. The kernel searches its internal cache for an entry pertaining to the given host id. Two possible error conditions may arise:

Case 1. No entry for the given host is found in the cache.

The kernel allocates a cache entry for the host, sets its state to SEARCHING, and makes a broadcast-based search across the system for the host. If the host exists and receives the message, it responds with its <host_id>→<network_address> mapping information. This entry is cached, the cache state updated to VALID, and the IPC message launched. The IPC operation fails if no reply to the search message is received.

Case 2. The entry in the cache is not up-to-date.

The IPC message is sent to the associated network station address. However, the entry may be inconsistent if the host has failed, migrated or is shut down. If the machine was down and has since been brought up or if a different host has migrated onto this machine, the new host responds with a I_AM_NOT_THE_HOST error message to enable the sender to detect the inconsistency. (Note that a reincarnated host is logically equivalent to a different host listening on the same network address). Otherwise, the sending host detects the inconsistency by the lack of any response. In either case, the kernel makes a broadcast-based search to locate the host. If the host exists on the network, it responds with its network address. The kernel then dispatches the IPC message at this address. If there is no response to the search, the kernel fails the IPC operation, and deletes the cache entry.

Cache management techniques and related issues such as cache size, cache update policy and the implications on the efficiency of message-passing are interesting problems but do not directly concern us and are not discussed here.

6.1 Overflow of the host id space

The dynamic nature of machines joining and exiting from the system tends to fragment the host id space over a period of time. Due to the finite host id addresss space and because successively larger id's are allocated, overflow of the host id space will eventually occur. When this happens, either the host id address space is allowed to wrap around and recycle or no new id is

issued. As the former method violates the non-reusability property, we choose the latter, requiring manual intervention of the system manager to reset all machines in the system. Practically though this should never happen. When the highest host id in the system exceeds a threshold close to MAX_ID, a warning message can be issued. This allows the system manager to schedule a shut-down of the system (which may coincide with the next regular maintenance if possible) to reset the host id's. For larger systems where shutting down the entire system is undesirable or impractical, the host id space should be chosen large enough to reduce the probability of overflow. Suppose a 24-bit host id space is used. Even on a large scale system where, say, 1000 workstations are brought up and down on an average each day, the system reset cycle is of the order of 45 years.

The scheme enables detection of the id overflow condition by the inability of a new machine to join the system for want of a host id. On overflow detection, the corrective measure requires manual resetting of the entire system. This is unlike conventional schemes where id's begin to recycle silently without any warning, violating the non-reusability property.

7. Network partitioning

Any dynamic id assignment scheme is susceptible to failure when the network partitions into two or more fragments some of which subsequently remerge. This is due to the fact that for each fragment, only partial information about the global state of the host id space is available to the id assignment protocol, leading to an uncoordinated allocation of id's in the various fragments. We illustrate the problem with a simple scenario. Consider a network that partitions into two fragments Net_A and Net_B. Any dynamic assignment scheme within Net_A and Net_B guarantees the uniqueness property only within each fragment. It is therefore possible for some of the hosts in Net_A and Net_B to have the same id's, i.e., a host id may be bound *one-to-many* to logical hosts. When Net_A and Net_B remerge, the high level issues that arise due to multiple hosts possessing the same id's are quite complex and very often non-deterministic. The severity of the problem depends on the time duration for which the network remained partitioned and the frequency with which machines joined and exit from the system.

7.1 Recovery procedure

Inconsistencies that arise when the partitioned fragments of a network remerge come to light mostly during an IPC across the erstwhile fragments. We consider the following scenario involving two partitions Net_A and Net_B illustrated in Figure 5. Let A_2 and B_2 be hosts in Net_A and Net_B respectively that possess the same host id h_{id}. Let A_1 and B_1 be hosts in Net_A and Net_B that have cached address entries for A_2 and B_2 respectively. Suppose Net_A and Net_B remerge. As long as the cached entries for h_{id} in A_1 and B_1 are not deleted, the interaction of A_1 with A_2 and that of B_1 with B_2 proceed correctly. The problem occurs if A_1 deletes the cache entry for h_{id} due to its internal cache management policy. When an IPC is initiated by a process on A_1 to a process on A_2, the kernel on A_1 allocates a cache entry for h_{id} with the state set to SEARCHING, and generates a broadcast-based search for h_{id}. Now both A_2 and B_2 respond to the message which may be the first available indication of the inconsistency. The scenario may be generalised to M hosts (A_2, B_2,..) possessing the same identifier. Recovery consists of two phases:

7.1.1 Detection by the sender. In response to the host search message broadcast by A_1, all the hosts with id h_{id} respond with their respective network addresses (NWA_{A2}, NWA_{B2},..). Let A_2 be the host whose response was first received by A_1. The latter caches the entry, updates the state of the entry to VALID, and dispatches the IPC message at the address NWA_{A2}. When the other responses to the search message are received, the cache in A_1 may be in one of two states:

(i) The previous entry $\{h_{id} \rightarrow NWA_{A2}\}$ exists still, and is in the VALID state.

(ii) No entry in the cache for h_{id}.

Both states indicate that the host is not searching for h_{id}. The kernel detects the inconsistency and initiates recovery by broadcasting a REASSIGN_HOST_ID(h_{id}) message. A_1 then fails any on-going IPC with h_{id}, and deletes any cached entry for h_{id}. Thus the host that initiated the IPC also detects the inconsistency (i.e., detection at *source*) and initiates the recovery procedure. Note that recovery procedures (discussed below) may be integrated into the name binding mechanism since the latter is dynamic.

7.1.2 Recovery by clashing hosts: forced crash. On receiving the REASSIGN_HOST_ID(h_{id}) message, each host in the system compares its host id against h_{id}. In the case of a match, the kernel on that host initiates a reset procedure cleaning up its local executions, and then attempts to rejoin the system. Since a machine gets a guaranteed unique id on (re)joining the system, the issue of multiple hosts possessing the same id is eliminated. If the host id does not match with h_{id}, the host searches its cache for the h_{id} entry, and invalidate it if found. Effectively, such a forced crash is considered as a host failure at the process abstraction level. Since most IPC mechanisms have built-in structural elements to guard against host failures[15], an on-going IPC with the processes on the host being reset will be terminated with an appropriate error message with which high level applications may recover. Any IPC that may be directed to the host with its old host id will eventually be terminated by a lack of response or a I_AM_NOT_THE_HOST message from the reincarnated host since the same network address will still be used but under a different host id.

7.1.3 Recovery by clashing hosts: host renaming. In this scheme, the kernel destroys the id associated with a clashing host, and rebinds the host to a new id. It is an *involuntary* activity at the process abstraction level. The implication is that the kernel no longer guarantees the constancy of the process id over the life time of a process. The scheme requires mechanisms to handle the following: i) acquisition of a new host id, ii) renaming all processes already created from the host since its incarnation, iii) handling the on-going message-passing activities the process is engaged in under its old name, iv) correcting all external references to the process, and v) handling future message-passing activities onto the process under its old name. The approach imposes additional structural elements in the client-server interface of programs to handle issue (iv) (the others may be directly handled by the kernel).

The advantage of the forced crash scheme is that it is simple to realise and does not require any additional mechanisms; however, it may degenerate into a harmful mechanism if it occurs too frequently. On the other hand, the technique based on host renaming imposes some normal case overhead in the form of checking for changed process id's and rebinding the references. However, it does not cause destruction of the host which may be advantageous in some environments.

8. Conclusions

We have discussed the mandatory and the desirable properties of host id's, and their implications in LAN-based distributed systems. A new scheme to generate and manage host id's satisfying the properties discussed has also been presented. The distributed scheme is robust against machine failures and message losses. Additional mechanisms to minimise the effects of network partitioning are also introduced.

The scheme provides an integrated approach to the id allocation problem enforcing the required properties. It is distributed in that the state of the host id space is maintained by the participating machines themselves and does not rely on a centralized id generator or disk-like stable storage. The notion of a dynamic polling window which slides along the host id space as well as changes its size allows selective polling of the hosts in the system by a joining machine thus minimizing the number of responses needed to select a tentative host id.

Some of the ideas encapsulated in the host id generation protocols, namely, collision detection and resolution, and the derivation of a consistent view of the distributed state variable from its potentially inconsistent instances, are applicable to other problems in distributed systems such as mutual exclusion and concurrency control. It is interesting to note, in this context, that well-established network level concepts such as token passing and ring configurations are being adapted as standard interfaces for LAN-based Operating Systems to solve high level problems[32].

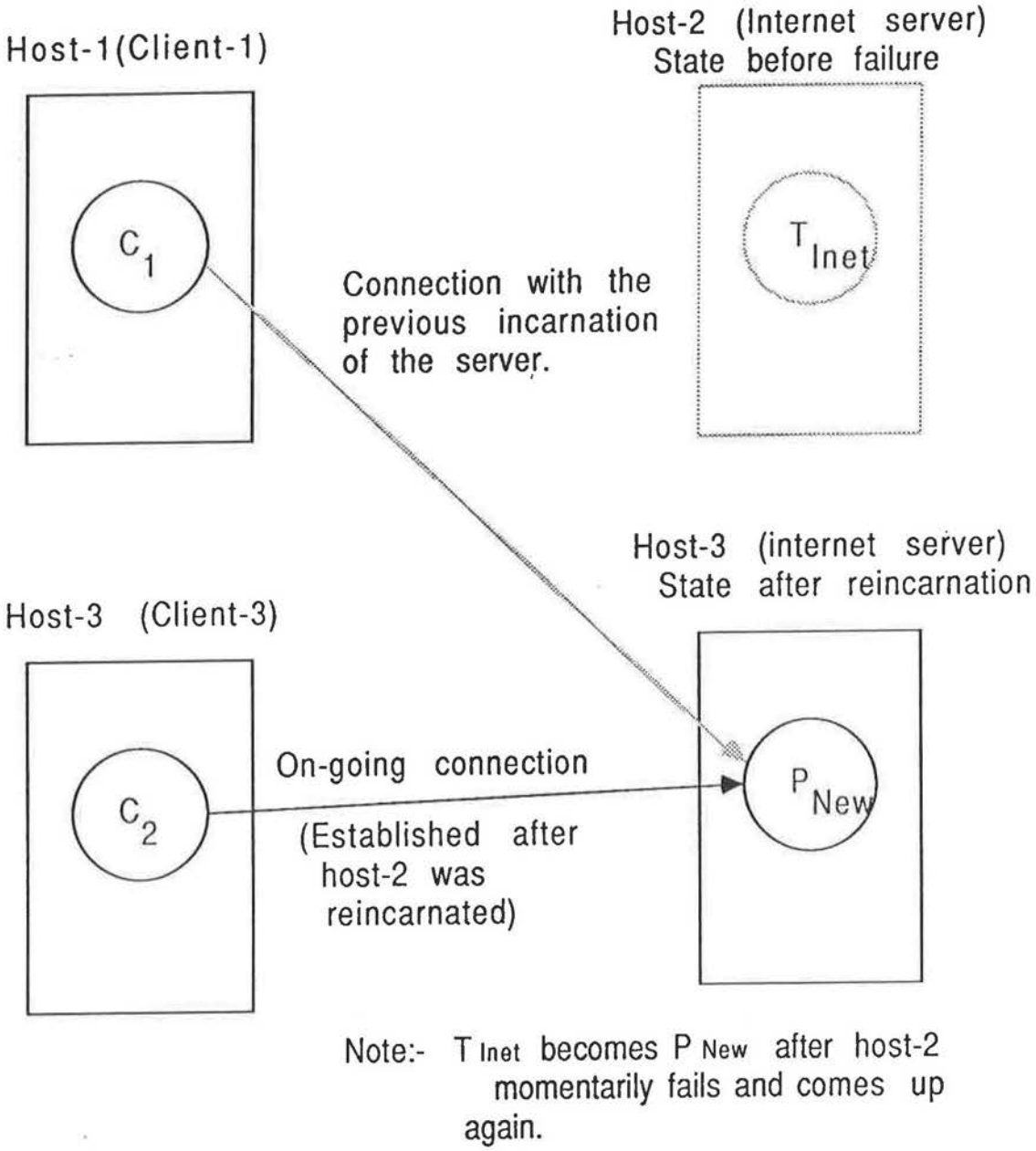
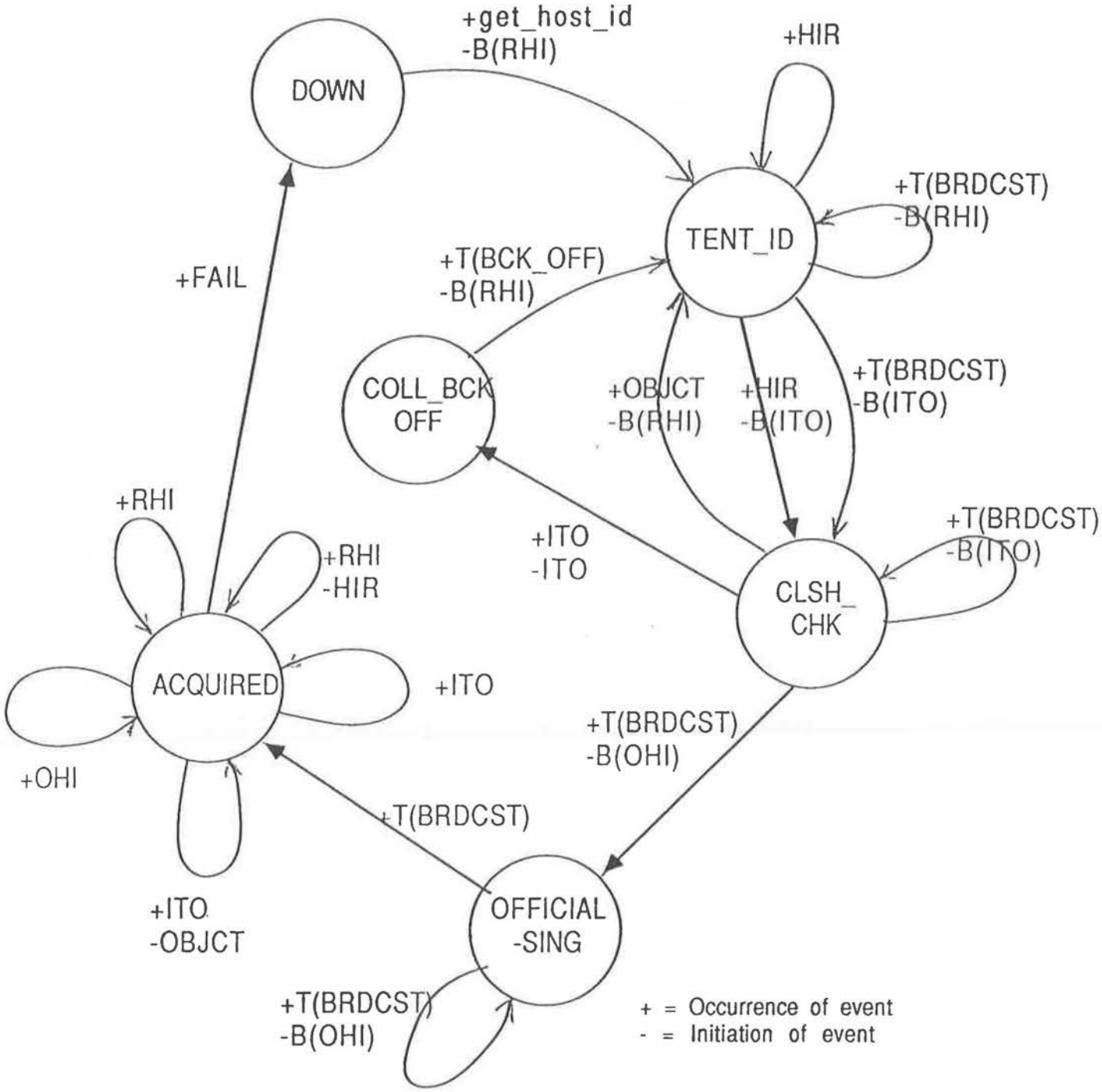


Figure 1. A scenario to illustrate the implications of reusing host id's.



RHI -- REQUEST_HOST_ID

ITO -- IS_THERE_OBJECTION

OHI -- OFFICIAL_HOST_ID

OBJCT -- OBJECTION

HIR -- HOST_ID_REPLY

T(E) -- Event E times out

B(M) -- Broadcast message M

Figure 2. FSM representation of the host identification protocol

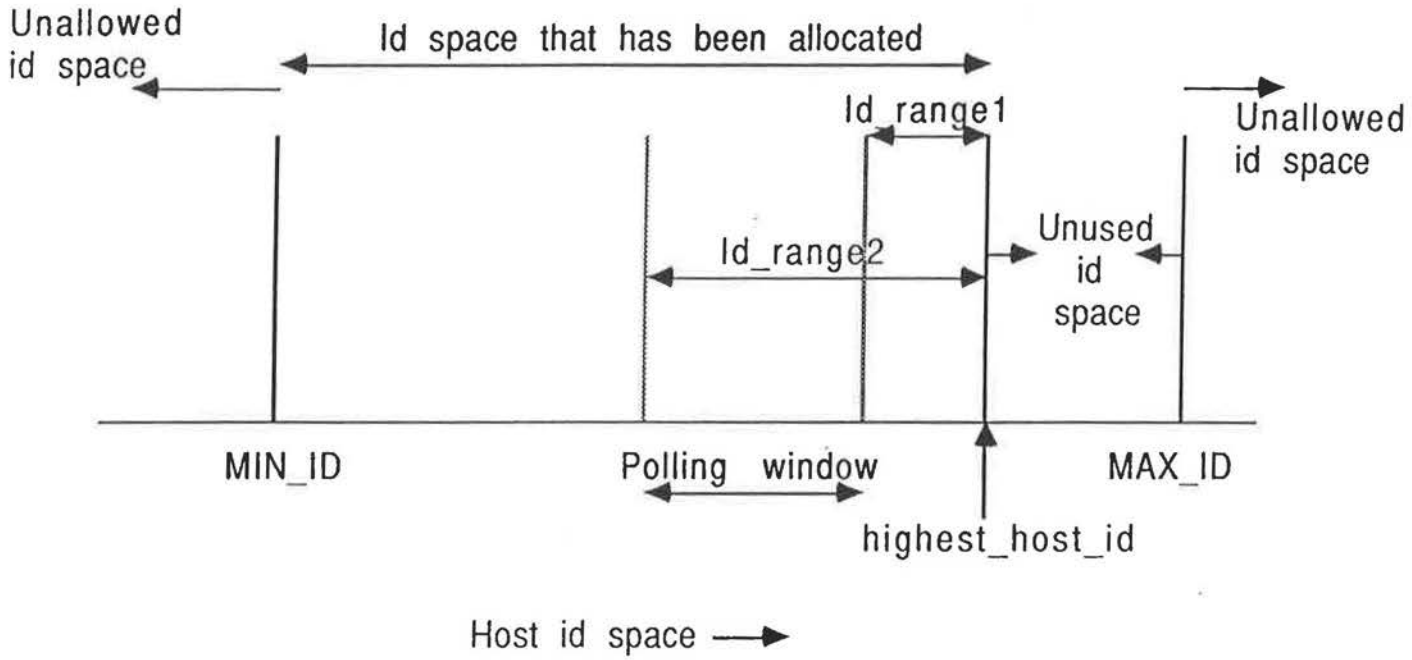


Figure 3. Diagram illustrating the polling window scheme.

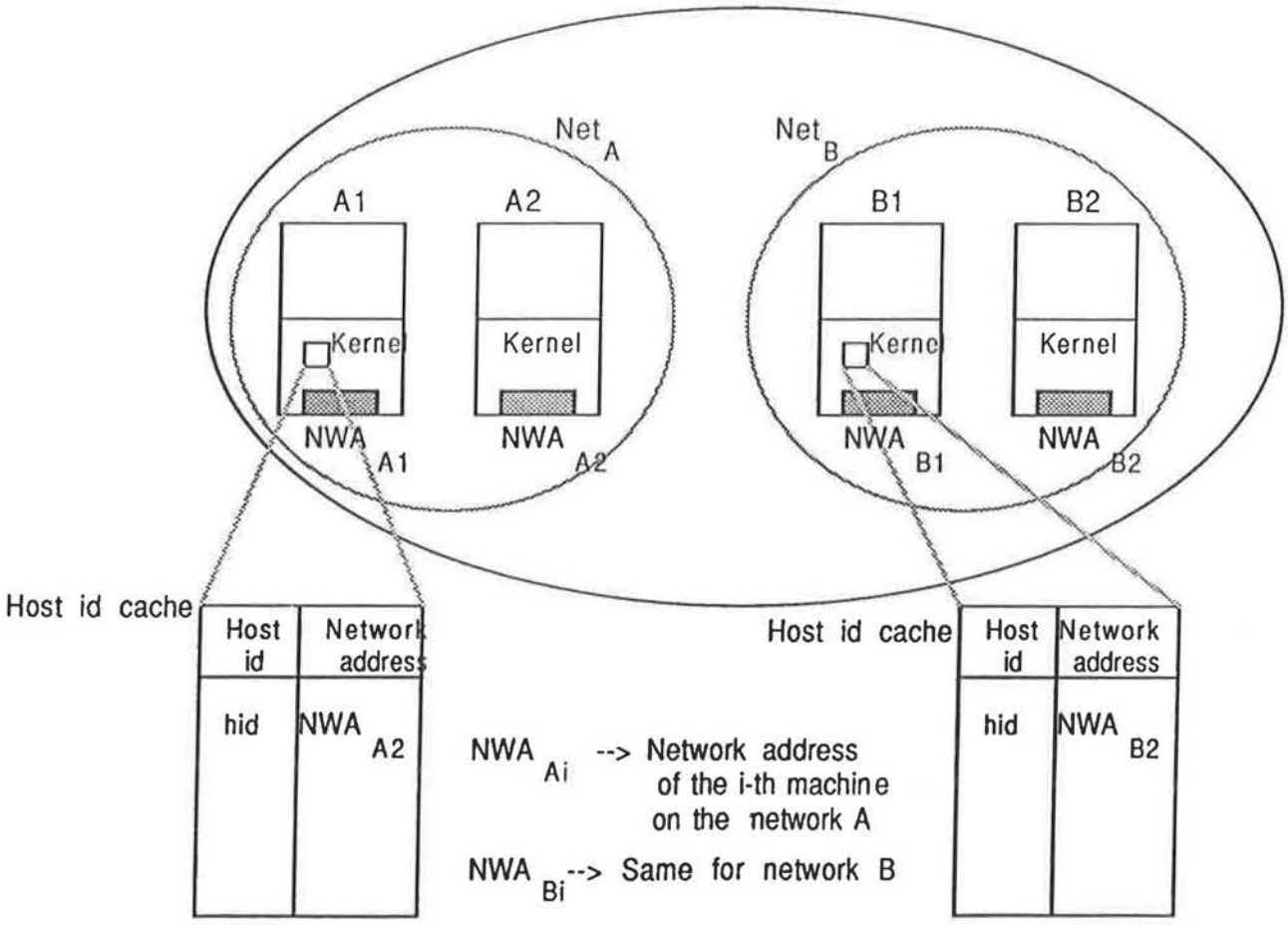


Figure 5. Diagram to illustrate the inconsistencies caused by network partitioning

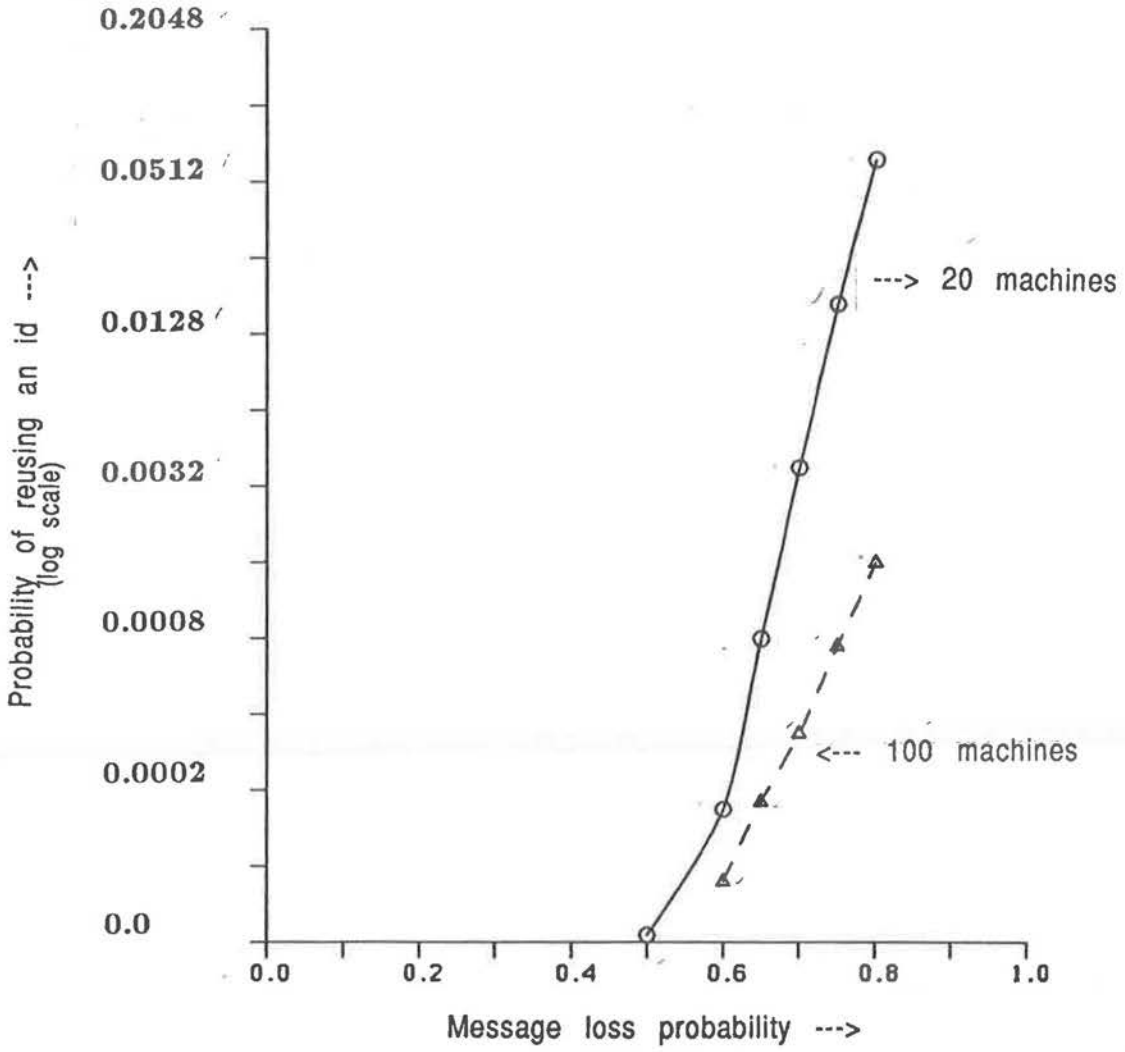


Figure 4. Probability of reusing an id as a function of message loss probability

Table 1. Summary of simulation results

Total No. of machines in the system	20	100
Msg. loss prob. beyond which the prob. of reusing an id becomes non-zero	0.5	0.6
Mean time for a machine to join the system	0.58 - 0.72 sec.	0.6 - 0.63 sec.
Mean No. of msgs. sent by a joining machine	5.8 - 7.5	5.5 - 6.1
Mean No. of msgs. received by a joining machine	4.2 - 7.9	7.5 - 8.1
Mean No. of msgs. put onto the network	Decreases linearly with msg. loss prob. 14 for loss prob. = 0.1 10.5 for loss prob. = 0.7	Decreases linearly with msg. loss prob. 20 for loss prob. = 0.1 15 for loss prob. = 0.7
Mean CPU cycles consumed on an active host by a joining machine	Decreases linearly with msg. loss prob. 26 ms for loss prob. = 0.1 11.5 ms for loss prob. = 0.7	Decreases linearly with msg. loss prob. 24 ms for loss prob. = 0.1 9 ms for loss prob. = 0.7
Collision prob.	Decreases linearly with msg. loss prob. 0.014 for loss prob. = 0.1 0.008 for loss prob. = 0.7	Decreases linearly with msg. loss prob. 0.073 for loss prob. = 0.1 0.045 for loss prob. = 0.7

References

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

2. 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.

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

4. 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.

5. 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.

6. K. Ravindran and S.T. Chanson, *Structuring reliable interactions in distributed server architectures,* Technical Report #86-13, Dept. of Computer Science, Univ. of British Columbia, June '86.

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

8. DARPA Internet program protocol specifications, *Transmission control protocol (RFC 793)* and *Internet protocol (RFC 791)*, Information Sciences Institute, USC, CA, Sept. '81.

9. W. Zwaenepoel, *Protocols for large data transfers over local networks*, Proc. of the 9th Data Communications Symposium (ACM SIGCOMM), Sept.'85, pp.22-32.

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

11. 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.

12. M.M.Theimer, et al., *Preemptable Remote Execution Facilities for the V-System*, Proc. of the 10th ACM symposium on Operating System Principles, Dec.85, pp.2-12.

13. D.R.Cheriton, and T.P.Mann, *Uniform access to Distributed Name Interpretation in the V-System,* Proc. of the 4th IEEE-CS symposium on Distributed Computing, May '84, pp.290-297.

14. M.M.Theimer, *Private communication*, Dec.'85.

15. K.Ravindran, and S.T.Chanson, *State Inconsistency issues in Local Area Network-based Distributed Kernels,* Proc. of the 5th IEEE-CS symposium on Reliability in Distributed Software and Database Systems, Jan.'86, pp.188-195.

16. I.S.Gopal, and A.Segall, *Dynamic Address Assignment in Broadcast Networks,* IEEE Transactions on Communications, Vol.COM-34, No.1, Jan.'86, pp.31-37.

17. W.S.Lai, *The integrity of Data Delivery in Computer Networks,* IEEE Transactions on Communications, Vol.COM-33, No.11, Nov.'85, pp.1222-1224.

18. S.J.Leffler, R.S.Fabry, and W.N.Joy, *A 4.2 BSD Inter process communication primer,* Comp. Syst. Res. group, Univ. of California, Berkeley, Draft of Mar. '83.

19. D.Plummer, *An Ethernet Address Resolution protocol,* RFC 826, MIT, SUN-800-1059-001.

20. SUN Microsystems Inc., Mtn.View, CA, USA, *SUN-2/120 and SUN-2/170 Configuration guide,* Feb.'84.

21. 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.

22. S.Vuong, S.Chanson, and R.Lee, *A hybrid Local Area Network for efficient and reliable communication,* Proc. of Int'l. Electronics and Electrical Conference and Exposition, Toronto, '83.

23. 3COM Corporation, Mtn.View, CA, USA, *Model 3C400 Multibus Ethernet Controller reference Manual,* May '82.

24. J.Sventek, et al., *Token ring Local Area Network - a comparison of experimental and theoretical performance*, Proc. of IEEE Computer Networking Symposium, Dec.'83, pp.51-56.

25. S.A.Bruso, *A failure detection and notification protocol for distributed computing systems,* Proc. of the 5th IEEE-CS symposium on Distributed Computing Systems, May '85, pp.116-123.

26. J.Crowcroft, and R.Hall, *A distributed approach to loading in a networked environment*, Ring Technology Local Area Networks, Elsevier Science Publishers B.V.(North-Holland), IFIP '84, pp.111-132.

27. 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.

28. W.M.Louck, et al., *Implementation of a dynamic address assignment protocol in a local area network*, Proc. of 10th Intl. Conf. on Local Computer Networks, Oct.'85, pp.149-157.

29. D.R.Cheriton, *Problem-oriented shared memory: A decentralised approach to distributed system design*, Proc. of the 6th IEEE-CS Symposium on Distributed Computing Systems, May '86, pp.190-197.

30. F.C.M.Lau and E.G.Manning, *Cluster-based addressing for reliable distributed systems*, Proc. of the 4th IEEE-CS Symposium on Reliability in Distributed Software and Data Base Systems, Oct.'84, pp.146-154.

31. J.H.Saltzer, *On the naming and binding of network destinations*, Local Computer Networks, North-Holland Publishing Co., IFIP, '82, pp.311-317.

32. G.P.Rossi and C.Garavaglia, *A proposal for an improved network layer of a LAN*, ACM SIGCOMM, Computer Communication Review, Vol.16, No.1, Jan./Feb.'86, pp.13-17.

Appendix-A

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

```
Kernel_init()   /* kernel initialisation */
  {
.
.
.
      window_size = 0;
      Id_range2 = 0;
tentative_id = 0;

for ( seq_no = 0; Id_range2 <= (MAX_ID - MIN_ID); seq_no++ )
  {
rebroadcast_count =0;
do
  {
msg.type = REQUEST_HOST_ID;
msg.station_adrs = <Network interface address of this site>;
msg.brdcst_cnt = rebroadcast_count++;
msg.seq_no = seq_no;
msg.Id_range1 = Id_range2;
            Id_range2 += window_size;
            msg.Id_range2 = Id_range2;
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 ( (<enough # of replies recd.>) &&
                  (<enough range of if id space polled>) )
            /* The kernel is confident to select highest id */
  {
highest_id = get_highest_value ( <cache> );
tentative_id = ( highest_id + 1 ) mod MAX_ID_SPACE;
```

```
break;
 }
else
window_size = generate_next_window();
                    /* Select next window by some algorithm */
 } /* End of "for" loop */
If ( tentative_id == 0 ) /* No other host in the network */
tentative_id = MIN_ID;
.
.
 } /* End of "Kernel_init" */
```

A.2 The code skeleton executed by the kernel on other hosts is as follows:

```
forever
 {
receive ( msg, NETWORK_RECEIVER );
     if ( msg.type == REQUEST_HOST_ID )
 if ( (self_id ≤ (highest_host_id - msg.Id_range1)) &&
        (self_id ≥ (highest_host_id - msg.Id_range2)) )
{
 response_msg.host_id = highest_host_id;
 response_msg.type = HOST_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.host_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-B

Control parameters of the host id generation protocol

In the process of acquiring a tentative host id, the protocol must determine when to stop searching and select the tentative host id based on the replies to the REQUEST_HOST_ID message(s). This decision depends on the number of replies received as well as the range of host id covered by the search so far. Intuitively, the larger the number of replies and/or the larger the host id space polled, the higher the probability of selecting the correct highest host id in the network. These two parameters are highly inter-related, reflecting the dynamic state of the hosts in the network. A weighted combination of the two may be used to stop further search as follows:

$$K_1 * N_{replies} + K_2 * R_{id} \geq \text{threshold} \qquad \text{.....(B.1)}$$

where $N_{replies}$ is the total number of replies received so far, R_{id} is the range of the host id space covered so far, K_1, K_2 are weights associated with $N_{replies}$ and R_{id} respectively and threshold is a predetermined constant. Whenever the condition in (B.1) is satisfied, the highest host id among the replies received so far is taken to be the highest host id in the network. This is incremented by one to produce the tentative host id.

Two boundary conditions are needed to estimate K_1 and K_2.

Boundary condition 1. When the machine attempting to join the network is the first one to do so, there will be no reply to any of its probes. Thus the entire id space (R_{id_max} = MAX_ID - MIN_ID) will have to be polled.

i.e., $N_{replies}=0$ and $R_{id}=R_{id\_max}$. Applying this boundary condition to the limiting case of equation (B.1), we get

$$\text{threshold} = K_2 * R_{id\_max} \qquad \text{.....(B.2)}$$

Hence equation (B.1) may be rewritten as

$$\left(\frac{K_1}{K_2}\right) * N_{replies} \geq (R_{id\_max} - R_{id}) \qquad \text{.....(B.3)}$$

Boundary condition 2. We assume that a machine needs D replies to make a decision in the first polling window itself.

Then, $N_{replies}=D$ and $R_{id}=D$ (the value of D is dependent on the size of the network but generally a small value not exceeding 10 would do). Applying this to (B.3) we get

$$(\frac{K_1}{K_2}) \geq \frac{R_{id\_max}}{D} \qquad \text{.....(B.4)}$$

since $R_{id\_max} \gg D$.

Suppose D = 10 and half of the host id space is polled, i.e., $R_{id}=\frac{R_{id\_max}}{2}$, then from (B.3) and (B.4) $N_{replies} \geq 5$. In other words, when half of the host id space is covered, the protocol may choose the highest host id from among the replies if at least 5 replies have been received.

Another control parameter is the size of the polling window used for each search. The size should be chosen such that the number of expected replies is neither too large (which causes unnecessary network traffic) nor too small (insufficient information to decide on a tentative host id). Note that for a given window size, the further the polling window is from the highest host id, the less the number of active host id's contained in it. This, together with the large range of allowable host id space, suggests the use of a logarithmically increasing function in selecting successive window sizes so that the condition (B.1) can be satisfied within a reasonable number of searches. Thus, if `m´ searches have been completed so far, the equation specifying the window size (W_{m+1}) for the next search may take the form

$$W_1 = D \qquad \text{for } m=0, \quad \text{and}$$

$$W_{m+1} = f(N^m_{rep}, W_m) * K_3^{**m} \quad (m=1,2..) \qquad \text{.....(B.5)}$$

where K_3 is a base parameter, say 10. One choice of the function `f´ may be

$$f(N^m_{rep}, W_m) = [1-Min(\frac{N^m_{rep}}{W_m}, r_s)] * K_4 \qquad \text{.....(B.6)}$$

where K_4 is another base parameter (say 10) and r_s is a constant ($0.0 \leq r_s \leq 1.0$). The latter puts a lower bound on the chosen window size.