Two-Part Names and Process Termination

Francis C.M. Lau

Computer Communications Networks Group
University of Waterloo
Waterloo, Ontario
Canada N2L 3G1

1. Preamble

At one point in designing our system† we are faced with the problem of killing a remote process. Here we introduce the particular naming scheme that has been chosen for the system and shows how it affects the solutions to the problem of process termination.

2. The System

At the highest abstraction, there are processes (users' and servers') that interact via messages. Message-passing is supported by a *message manager* implemented as an integral part of the kernel (could be thought of as just another process). Creating, killing, or halting of processes are done via requests to the *process manager* of the kernel. Processes are members of *clusters*. Clusters in general are of two types: *distributed (application) programs* (called *user clusters*) and *system services* (called *service clusters*). Processes of a user cluster work cooperatively towards some common goal; processes of a service cluster (called *servers*) are identical - they are multiple instances of the same service; their multiplicity leads to increased reliability and efficiency. Our processes therefore have a two-part name of *Cluster ID* (*CID*) and *Local ID* (*LID*) (notationwise: <CID, LID>).

CIDs are unique network-wide while LIDs are chosen arbitrarily (the former serves, in a sense, to disambiguate the latter). As for application programs, LIDs are hardcoded at coding time and CIDs are assigned at instantiation time (a fresh one each time). Instantiation turns a program into a unique instance as well as a unique user cluster. During the execution of a program instance, member processes communicate with each other using only LIDs, and the CID that has been assigned to the instance is patched in automatically by the runtime package. While user clusters' CIDs are generally not known publicly, CIDs for services are well-known (they are published). Adding a new instance (ie. service proper plus a server) of a service requires looking up the list of well-known CIDs and copying the appropriate CID into the CID field of the new server.

3. Addressing Modes

With this two-part name, the addressing modes that are supported include‡ (where "C" = a specific CID, "L" = a specific LID, "?" = "any", and "*" = "all"):

SS	<C, L>	(S = Specific)
SA	<C, *>	(A = All)
SN	<C, ?>	(N = aNy)
NN	<?, ?>	
NA	<?, *>	
AN	<*, ?>	
AA	<*, *>	

Modes AS and NS are not included because they don't seem to be meaningful. The following are the common types of communications that are seen in our system:

(1) *Within the same user cluster* - To address members in the same user cluster, simply use the corresponding LIDs (modes SS, SA, and

† A local distributed system that supports among other things, efficient migration of processes.

‡ See [McQuillan78] for a more extensive treatment on addressing modes.

SN). Note that the sender process need not specify the CID.

(2) *User to service* - Servers' names have only the CID part (ie. the kind of service) and they respond to request messages with addressing modes SA and SN. Upon receipt of a request message by all servers bearing the requested CID, the following steps occur:

- ELECT - using some distributed election algorithm, one of the server is elected (this could be done in advance). If the request message has addressing mode SA, this is ignored (ie. a replicated service).
- CONNECT - the elected server would acquire a unique number and use it as LID - thus converting the request to an SS mode.
- SERVICE - the request is serviced and the requestor will be communicating with a "specific" server having the common CID and an LID generated anew.
- RELEASE - the server process will relinquish the LID (it could be remembered if there is any use of it afterward).

There are yet some situations that are not as straightforward:

(3) *Between different user clusters* - This can only be accomplished through some external means. At least the CIDs have to be communicated before any meaningful interprocess communications (eg. modes SS, SA, and SN) can take place between processes belonging to the opposite sides. Special servers that are publicly accessible can be provided for this purpose. For example, "conference servers" may be used to provide meaningful communications among user clusters that are unknown to each other in the first place.

(4) *Within the same service cluster* - Since servers of the same service cluster are not individually named (except during services), communications among them have to be done using addressing modes SA or SN (which is relatively inefficient). However, this kind of communications seldom occur in our environment (except for election for which many ways of speeding things up exist).

In essence, ours is a system of processes using two-part names (logical addresses). The set of message managers across the network colaborate to implement a *routing kernel* on the one hand, and provide the abstraction of end-to-end message transport using two-part names on the other hand. Logical addresses are necessary (sufficiency depends on routing mechanism) for efficient object migration as well as adding/deleting objects in distributed environments. In our system, sending a message to specific addressees (SS mode, as well as SA and AA modes) incurs one unit of cost (per addressee); all the other modes are considered expensive and should be used only as an initial means to establish subsequent connections.

4. Killing a Remote Process

The operation that we want to implement efficiently is the immediate termination (after checking the rights) of a remote process (the *victim*) that belongs to the same user cluster as the requesting process (the *killer*). Note that the actual and ultimate termination of the process is done by the process manager on which this doomed process depends (ie. they are on the same machine). This implies some sort of type (2) communications - that is, between a user process (the killer) and a server (the process manager).

The possible choices are (let C = CID of this user cluster, w7 = LID of victim, and PM = CID of process manager cluster (ie. all process managers)):

(1) Send message to the victim directly: <C, w7> ← "Bang!"

(2) Send message to the process manager directly: <PM, *> ← "Please kill process <C, w7>"

(3) Send message to the victim with special indication: <C, w7>[i] ← "Bang!"

Choice (1), the "programmed," requires the victim to issue an explicit receive, and upon receiving the kill message, terminate itself. This is rejected because of its non-interrupt nature. It would be wasteful (busy checking) and inefficient (time lapse before the receive request), if adopted.

As for choice (2), since there is no way (unless some hints exist) to find out which process manager is the victim in question connected to, the kill request has to be broadcast. This requires all process managers form a (well-known) cluster (PM, in this case). A kill request issued by a user process will first be trapped locally. This is followed by the kernel's trap routine broadcasting the request to all process managers. Note that unlike most other request messages, the LID field for this request message is "*" instead of "?". Later, the one

manager who finds <C, w7> in its domain will perform the killing, and the effort made by all the other process managers is necessarily wasted (how much wasted depends on how efficient the looking up of process names in a local process table would be for those that are "unselected").

Choice (3) is the one we adopted. It introduces the overhead of a single bit, the *trap bit* (it should not be considered the "third part" of a name since it plays no role in identification, and two-part is sufficient hitherto). Apart from an extra bit that must be transmitted along with every message, whose overhead in terms of message load is negligible, the receiving site has to deal with the checking of this bit for every incoming message. The latter is, in fact, also negligible, as compared to the amount of computation needed to recognize (pattern-match) the name that comes with the message. Upon detecting an "on" value of the trap bit, the message is switched (by the message manager) to the local process manager (rather than the victim) who will then carry out the termination of the victim in due course. The advantages of this scheme, as compared with the other choices, are briefly:

- A specific message (instead of a broadcast message) is sent. By courtesy of the routing mechanism, this will go to the right machine (and thus the right process manager) with minimal cost.
- The checking (whether the message is a trap message) is done inside the message manager - thus avoiding an extra context switch (in case the managers are implemented as separate processes) that might have been needed if the checking were done at a higher level - which would be the case if no indication or whatsoever is included in the name field.

5. Remarks

The "trap-bit" approach can be generalized: one bit allows single redirection, two bits allows three-way redirection, and so on. A request of the form

OP(<C, w7>); (ie. <C, w7>[bits] ← "OP")

issued by some user process would be translated into

<S[bits], -> ← "Please perfrom OP on <C, w7>"

at the receiving site. Broadcast is avoided in this case and the sending site is alleviated from the burden of finding out the specific name(s) of the server that is to serve the request (ie. saving from two names/addresses to one).

6. References

[McQuillan78]McQuillan, J. M. "Enhanced Message Addressing Capabilities for Computer Networks." *Proc. of the IEEE*, 66, 11, November 1978. 1517-1527.