XEROX

SYSTEM DEVELOPMENT DIVISION
Communication Systems
October 13, 1977

To: Whoever

From: Crowther

Subject: Clearinghouse Protocol

Preamble:

I have no particular love for the name *clearinghouse*. It is similar to a *name lookup server* in the world of the pup protocols, and a *naming authority* in the world of mail systems. File people call a similar thing a *directory*, and the phone company would call it *directory assistance*. Others call it *resource location*. I use a different name here to avoid confusion with these other things, regretfully acknowledging that they have already taken the better names.

Similarly the ideas here are hardly my own. They came from all over the place, wherever reasonable people attempted to cope with certain immediate problems in distributed systems. The motivation for this memo is to work out yet one more such system, namely OIS. Criticisms, suggestions, pet ideas, and even hate mail are solicited.

This document assumes the existence of a *remote procedure call* protocol. Jim White has worked out some details of such a thing, and has memos describing them. For the purposes of this memo, all that is implied by such a concept is that there is a mechanical rule for translating between a Mesa like description of a procedure call and the detailed format of a pair of messages passing across the network (with due regard to retransmission and suppression of duplicates and all that), with the further notion that the messages somehow turn into the call and return of a procedure in the remote machine. There is of course a set of real network protocols underlying the higher level stuff. It might be bytestream, or paged access, or request-reply, or something else: this memo won't talk about the underlying protocols. The point of remote procedure calls is that one can talk about a network protocol as though it were a set of procedure calls, and even one day be able to change them as easily as we now change procedure calls.

The Problem:

The clearinghouse protocol is an attempt to pull together the common thread from several

problems, and to solve them all with one mechanism. A brief list of a few such problems follows:

1.) Given the name of a user, to find whether he is on the net, and if so to identify the machine he is on.

2.) Given the name of a user, to find the location of his mailbox in a mail system.

3.) Given the name of a service, to find the location of the service. Some sample services are: a compiler, a printer, a gateway, a boot server, an error logger, a Juniper server, a Juniper directory server, a Juniper FTP server, and a mail system authentication server..

4.) Given the name of a distribution list in a mail system, to find the names it contains.

5.) Given the name of a machine, to find its unique identifying number.

6.) Given the serial number of a machine, to find which network it is on.

7.) (maybe) Given the name of a file, to find its unique file identifier.

8.) (maybe) Given the file identifier of a file, to find the location of the file in a distributed file system.

9.) Given the name of a clearinghouse, to find its address.

10.) Given the number of a Juniper disk pack, to find the address of the Juniper server where it is to be found.

The common thread of all these problems is that they involve a table lookup in a distributed environment. All share the usual problems associated with the notion that a few of the key machines will be malfunctioning or down at any given time, and all need to concern themselves with the usual problems of efficiency and delay when using network access. A further thread shared by most of the problems is that the result of the lookup is related to a particular machine, often being its network address. We will make use of this fact by trying for a system in which the lookup will fail in precisely those cases when the target machine is inaccessible to the user.

A user interface, part 1 (fuzzy)

The user of such a distributed lookup system would like to be able to register pairs of items - a name and a value - in such a way that any other authorized user could recover the value given the name. I use the words name and value because most of the examples above go from a text string to a network address, but any reasonable implementation would place no particular limits on the type of the name and value. There may well be limits on the size of items which can be stored.

In addition to the primitives register and lookup, one can immediately forsee a need to delete and enumerate entries. There are also issues of access control which I choose to ignore in this memo.

It is presently unclear where partial specification will be handled. Perhaps the clearinghouse should expand the characters *, ESC, and @, or perhaps the user wants to do that. Also, the user might want to specify that names should be matched without regard to the case of the characters.

There are a strange set of issues related to identical names. For various reasons it seems necessary to permit the registering of identical names in the case where the names can be disambiguated by the geographical location at which they were registered. For example, Smith at Parc and Smith at Stanford. Also Printer at Parc and Printer at Stanford. This

constraint will lead to geographical fields in the user commands, with a set of default rules and lookup strategies.

With the inclusion of identical names it becomes possible that a lookup will result in the discovery of several items registered with the same name. Therefore the result of a lookup will be a list of answers rather than a single answer.

The system under consideration clearly envisions several types of lookup. The first section of this memo lists 8 such types, and there will surely be several more before we are through. These types should be completely independent, so that there will be no confusion between identical names in different types. It is as though the type were concatenated to each name, creating a larger name guarenteed to be unique across types, (although for practical reasons it is better to keep type as a separate item). Unfortunately at least one potential application has already blurred the independence of separate types: the mail system wishes to lookup both mailbox names to addresses and distrbution list names to distribution lists (presumably strings). The problem here is that the syntax of the proposed mail system user interface is such that the system cannot tell a mailbox name from a distribution list. It therefore needs to query across two separate types. There are many solutions to this little problem, each with its own variation of uglyness. In any case, the clearinghouse presented here knows nothing at all about the syntax of the items it stores, beyond the fact that each item has a user specified length and that it belongs to a type (specified by a 16 bit CARDINAL).

Implementation

It is perhaps best to think of the clearinghouse as a Mesa module implementing a particular algorithm. In this view there is a clearinghouse in every machine. Some of these clearinghouses live in workstations. Such a clearinghouse has tiny storage capacity, and is used primarily for access to the outside world and to provide minimal service when all other clearinghouses are down. Other clearinghouses service a larger area - a network or a campus. Such clearinghouses would require greater storage capacity, but might coexist with gateways, printer servers, file servers, or similar specialized machines. There might be regional clearinghouses, where the size of a region might be a city, or a state, or a whole nation. Such machines would likely be stand alone units, perhaps with special disk units. The important thing is that these units may differ in size, but they all execute the same basic algorithm and communicate in the same way.

The clearinghouses are organized into a single large tree. A clearinghouse need only know the name and address of the next higher node in the tree in order to perform its normal functions. A tree structure seems to be the most efficient way to provide quick and easy access to a large mass of distributed data, particularly when the pattern of access is expected to mirror the tree structure, with most lookups being local. In the event of failures (to be discussed at some length below) the immediate concern of the clearinghouses is to reconfigure the tree. In order to do this a clearinghouse will need to know or discover all of the connectivity associated with its own level in the tree as well as that of the next higher level. This reconfiguration is the only truely distributed computation involved in the algorithm. The rest of the interactions involve a pair of machines in a strict master/slave relation.

A single datum will be stored in several places in the tree. One such place is special, in that it is the primary store for that datum. From the primary store copies of the datum will propagate up the tree until either the top of the tree or a specified level is encountered. These copies are permanent, in the sense that the tree is expected to store them, but they can be regenerated from the primary store if necessary. Other clearinghouses will also store the datum on a cache basis. These copies are ephimeral, and disappear on fairly short timeout or when their space is needed. Ideally the primary store for a datum is located at the bottom of the tree in the very machine which is needed to use the datum. In practice there will be compromises, since the bottom leaves of the tree are less reliable and may contain the wrong disk. The algorithm does not care where they are stored. The use for an upper limit is not immediately obvious: partially it stems from a simple desire to unload the top of the tree, which would otherwise contain all the information stored anywhere. This is not the primary motivation however, because the total data expected to be stored is not all that large, even when one includes all of the non-local files in a huge system. Rather, it is expected that some information will be restricted to a subtree because access to that information is restricted to the subtree, perhaps along company boundaries. This is all a bit fuzzy, but an upper limit is easy and apparently worthwhile.

Lookup will proceed up the tree from a starting node until the information is found or the lookup fails. Normally the lookup will start at the user's own node, but if the user has some idea of a better place to start (Jones *at Washington* for example) he has that option. The user may also specify upper and lower limits for the search, so that one may ask for a local service like a printer without getting a printer from some other city.

I deliberately have not said much about the form of the store. It may be hashed or b-treed or sorted. It may be mostly in core or out on huge disk files. My silence covers a deep ignorence. The first version, (which already exists) uses a linear search of a tiny unsorted in-core table. Better stuff comes as need arises.

Reconfiguration

The clearinghouses structure can fail in several ways. We take the approach that it is unnecessary to prevent failure, just so long as the structure puts itself back together in a reasonably short time. This is a considerably easier problem than maintaining a fully reliable data base. We can afford this approach because the sort of material stored in the clearinghouse typically locates a service for a client. The client can verify for himself whether the service exists at that location. Thus the result of a clearinghouse lookup is basically a hint (although it is reliable if it comes from the primary store for that item).

The reconfiguration rules are pretty straightforword, and apply equally well to a machine just coming up and to a machine losing the next higher node.

1) first try to connect to the higher level using a list of possible next higher level guys.
2) then try to connect to someone on your own level using a similar list.
3) then broadcast as far as you can in the hope that someone out there is listening.
4) then give up, and pretend you are the top of the tree. Occasionally try it all over again

from step one.

The lists mentioned above are partially built in at compile time or in a user.cm, at least for the machines above the level of workstation. They also include information stored from previous runs, when the geometry was perhaps better. In particular it would be wise to remember the address of the node two above oneself in the tree, and several nodes at the level just below that one. One might also wish to remember some nodes at ones own level.

A different set of rules apply to the repair of a clearinghouse. In this case normal, well-running machines must realize that there is a new node available and adapt to it. This is accomplished by periodic exercise of the reconfiguration algorithm, plus preference for the standard configuration if it can be realized.

There is a problem maintaining the data base in a consistant way. If only information about changes in the data base were to propagate, as the description above seems to imply, then the data base would gradually become inconsistant. Periodically, the whole collection of data for a particular type is refreshed from the next lower machine. This could be done on the initiative of either the upper or the lower node, apparently with equal effect. Since there is already the notion of asking a clearinghouse for all of its information (of a particular type), the initiative will be placed in the hands of the upper level, and it will use the normal request mechanism.

One might wish to store a significant amount of data locally, say in the case of a file system. The obvious way is to register all of this data with the local clearinghouse. A more efficient way is for a particular user to implement his private version of a clearinghouse, which knows about information of only one type, and *stores* that information in already existing tables. (Thanks to P. Bishop for this trick). Such a clearinghouse would then communicate with the local clearinghouse in the normal way, and need know nothing about networks or message formats. This is one of the virtues of a remote procedural call protocol.

There is an obvious problem both with the notion of clearinghouse level and with the notion of default names when the clearinghouse tree structure reconfigures. A machine which temporarily takes on the duties of a higher level must somehow respond to messages at two different levels, much as though there were two clearinghouses sharing the same machine (and the same data base). Similarly, when a clearinghouse substitutes for a machine at an equal or lower level, default values for services like printer should behave as though a single clearinghouse were really two clearinghouses temporarily sharing the same machine.

A user interface, part 2 (detail)

The clearinghouse provides a single procedure called **ClearingHouseDo** to its users. The same procedure is called by the clearinghouse algorithm when it wants to communicate with other clearinghouses. The procedure takes a single record of type **Entry** as a parameter and returns a record of type **Return.**

```
ClearingHouseDo:PROCEDURE[Entry] RETURNS[Return];
```

An **Entry** contains 10 items which completely define the nature of the request to the ClearingHouse:

```
call:CallType,
type:CARDINAL,
duplicateIndex:CARDINAL,
nameHandle:POINTER,
nameLength:CARDINAL
valueHandle:POINTER,
valueLength:CARDINAL,
sourceCH:Address,
primaryCH:Address,
bottomLevel:CARDINAL,
topLevel:CARDINAL
```

The key items in the record are the type, name, and value. The whole purpose of the clearinghouse is to allow one to register a value as belonging to a particular name in the caragory type, and then to retrieve the value given the name and type. As far as the clearinghouse is concerned there is no internal structure to the name and value beyond the ability to describe them with a pointer and length. The clearinghouse will copy these items, so that the user may reclaim their space after the call. As indicated, the type is a 16 bit number.

call is an enumerated type, with defined values Register, Delete, Overwrite, and Lookup. **call** defines the nature of the operation the user wishes to execute using the rest of Entry as parameters. It may be convenient one day to provide 4 separate procedures to implement the four operations: for the moment there is only one. The operations are fairly self-explanitory: the only surprise is Overwrite. It seemed reasonable to return an error message to anyone attempting to Register a name which already existed. The user is therefore required to delete an entry before registering a changed value. But that left the possibility of someone making a routine inquiry about a standard name and getting the "does not exist" answer, which seemed rather confusing when the truth is that it exists but is changing. To circumvent this problem, the user is permitted to delete and reregister in a single operation called overwrite.

Not all of the operations use all of the items in Entry. For example, Lookup does not supply a value. For the most part though, all of the items are used in all of the operations, which partially prompted the idea of making ClearingHouseDo a single procedure.

The **sourceCH** is simply the network address of the clearinghouse making the request. The format of an address is defined in the Pup package, and the clearinghouse uses it only as

parameters to the pup package. Normally the user will supply the default "Me" address, since the user is expected to querry his local clearingHouse and allow it to initiate remote inquiries.

The **primaryCH** is the address of the clearingHouse which is to have primary responsibility for the data. There are three separate ideas here: first, the data will be maintained by the primary clearinghouse, in such a way that any user who has network access to the primary clearingHouse via any path will get a successful result when he tries to access the stored item (even though he may not actually talk to the primary clearinghouse) On the other hand, if the primary clearingHouse is down or partitioned away from the user, the result of a probe is ill defined: it may or may not be successful, depending on what information has been cached by clearingHouses on his side of the partition. Second, the primary clearingHouse address acts as a hint to the system, telling it where the answer to a querry is likely to be found. Third, the primary clearinghouse address acts as an extention to the stored name, where its chief function is to disambiguate two otherwise identical entries. This last function allows one to register one Smith at Parc and another at Stanford, and then perform a successful lookup at the SanFrancisco Clearinghouse. PrimaryCH corresponds to naming authority in the mail system vocabulary. While it is imortant to have a concept like primary clearinghouse is the system, I expect that most users will supply a default, meaning that they simply want to use the branch of the tree they are located on, at the appropiate level(as explained below).

The clearingHouse algorithm has the concept of a tree built into it, and each clearinghouse is expected to know its level in the tree. At the moment the levels are not well worked out, but we can imagine the workstation being level one, a grouping of several workstations perhaps on a single Xerox Wire as level two, a facility like a plant complex being level three, and a political unit like a city level four.
The user is permitted to specify both a bottom and a top level for each clearinghouse request. Such a specification selects a subtree from the whole clearinghouse tree, and the operation will proceed only within the subtree. In the case of Register, the information would be known only in the local branch of the tree between the specified levels, and in the case of Lookup the search would only extend to that subtree. If the user specifies a primaryCH, then the subtree is built upward from that address; otherwise the subtree is based on the users own machine. The user may well register an item using the default primaryCH and specifying a bottom level of two! This would mean that the item would be unknown in his own machine, but would be stored as though its primaryCH were his network clearinghouse. He would then be free to boot his machine and the entry would be preserved.

Lastly, the duplicate index provides a way to retrieve multiple entries with the same name. Normally the duplicate index is set to zero, and a lookup returns the first value it finds. If the index is set to one, then one value will be skipped and the second returned, etc.
The return indicates whether there are duplicate entries yet unseen, so the user can know whether to try again if he cares to. (see below for an explination the return).

The **return** is a three item record:
result:Result,
valueHandle:POINTER,
valueLength:CARDINAL

where **Result** has the values

{ok,noRoom,duplicate,noSuchPlace,notFound,okButThereAreMore}

The **value** is only of interest for Lookup, and some of the results do not apply to all operations. Nevertheless all operations return the same three word record.

finishing up

There is a very crude implementation of this now running on an alto. It seems to be rather small and clean, so perhaps the whole idea is not a terrible one. It uses Jim White's data structure protocol to implement a kind of remote procedure call, which hopefully means that the implementation is subject to easy change and experimentation.

The point of this whole thing is to try to gather together in one place a number of problems so they can be dealt with by a single mechanism. This will work only if the mechanism in fact deals with the problems in an adequate way. Comments from designers working in these problem areas are urgently solicited.