The EVAL Server -- A PUP Network Inter-Communications Facility for Interlisp-D

File: <Lispusers>EvalServer.press
Revised: Feb 21, 1983, by JonL White

The file EVALSERVER.DCOM contains routines to facilitate communication, over the EtherNet, between two or more D-machines running Interlisp-D. In simple usage, one or more D-machines (Dolphin, Dorado, or Dandelion) are started up with "servers" running, and a "client" of the service merely requests (via the ethernet) that a server EVALuate some form, and return the results. A user may direct his "remote" evaluation request to a specific server, or may broadcast it on his directly-connected network; in the latter case, at most one of the servers will be given the go-ahead to perform the service, but there are plans for future extensions to permit results from many servers for a broadcast request. (Likely a request for service will be coupled with a function to be applied to any results that come back).

These routines work most conveniently when the multiple process mechanism is enabled (see <Lispusers>Proc.press), but function quite adequately without it; additional hints will be given at the end of this note for those who cannot have the process mechanism enabled, but who would like to have two Interlisp-D's talking to each other.

Starting up the "server" side

The function call
(EVALSERVER <duration.minutes> <clientslst> <gaggedclientslst>)
will initiate a background process named "EvalServer.Listening" (of course, this will be a "foreground" process, if the multiple process mechanism isn't enabled). All arguments are optional, and have the following meaning:

- <duration.minutes> -- if non-null, will run the service for only the number of minutes specified; otherwise it will run indefinitely.
- <clientslst> - - a list of ether host names (or numbers) whom the server is willing to service: if T is on this list, it will service any request addressed to this specific host (called, "flirtatious" mode), if NIL is on the list, it will service any request broadcast on the connected network (called, "promiscuous" mode). Default for this list, NIL, is converted into the union of "flirtatious" and "promiscuous" modes.
- <gaggedclientslst> -- a list of hosts for whom service will not be performed; NIL on this list means no service for broadcast requests; T on this list means no service for "myself" (i.e. attempts by some process running concurrently on the server host to send a "remote" evaluation request to the same host, will be rejected).

In any case, the global variables EvalServerClientHosts and EvalServerGaggedHosts will be dynamically consulted for the same information; this is so that you may tailor the client screening process while the server is running. The "gags" always have precedence over the "hosts" lists.

While the server is running, a call to
  (EVALSERVER.ABORT <transaction> <guiltyparty> <errorflg>)
will stop a currently running process, such as one that is in a loop, or that is "poaching" more time off the server than is desireable; see below under EVALSERVER.STATUS for a definition of the identification terms.
  <transaction> may be either an integer, the "identification" number of some service, or a cons of the id number and the host; the "cons" form may be required if there are two or more services, from separate requesting hosts, with co-incidentally the same id number.
  <guiltyparty> permits recording why a service was stopped; it is optional, and the default recording is "Aborted Locally by Error/Quit".
  <errorflg> is optional, and if non-NIL, will cause an ERROR to occur if there is no transaction as specified by <transaction> or of the transaction seems to be wedged.

While the server is running, a call
  (EVALSERVER.STATUS <wherependingflg> <id>)
will return a list of information associated with the state of currently-running services, already-completed services, and requests still in the input queue. A "service" is identified by a cons of a transaction number and a host number; the transaction number is in fact the packet id in the PUP used to communicate over the EtherNet (a similar idea will eventually be used when the server is implemented in NS protocols).
  <wherependingflg> -- selects one or more of the information lists to be output, as follows:
    DONE (or COMPLETED or FINISHED) adds in the remnants of the list of completed services (which list is pruned down from time to time by the "cleanup" process mentioned above. A global variable, \ES.PURGEINTERVAL.SECS, controls the frequency of the "cleanup" actions, and also the maximum time for which old completed service records are kept);
    RUNNING (or CURRENT) adds in the data for currently executing evaluations requests;
    INPUT (or INPUTQUEUE) adds in the data for PUP's still waiting for service.
    T acts like the union of DONE and RUNNING;
    ALL acts like a union of all three of DONE, RUNNING, and INPUT;
    NIL, and no argument, default to same as ALL.
  <id> -- if non-null, then only records with that identification number will be included in the result.
Similarly, a call
  (EVALSERVER.STATUS.WINDOW <optional-region>)
will put up a window which continuously monitors and displays the above information. The middle mouse button is active in this status window to delete the various items; deleting a RUNNING or INPUT request is the same as calling EVALSERVER.ABORT on it.

A trace facility may be enabled/disabled by a function call:
  (EVALSERVER.TRACE <flg> <region>)
where
  <flg>, if non-null, enables traceing; disables if null.
  <region>, if non-null, should be a "region" and marks a region for the trace window; if null, the user is prompted to "mouse"

a region.
Both the left and m idle mouse buttons are "active" in the trace window: LEFT toggles a "flg" which turns traceing on or off, and MIDDLE does a quick clear of the trace window.

The Client: Using a remote Eval Server

The basic use of EVAL at a remote host is invoked by:
(REMOTEVAL <form> <serverhost> <multiple.responses?>)
The S-expression <form> is shipped via the ether net to the host <serverhost>, where it is EVALuated by an EvalServer and shipped back If <serverhost> is NIL or 0, then the request is not directed to a partricular server, but is merely broadcast on the network; if any server is willing to service such a request, then it will "handshake" with the requestor and do so. If the remote evaluation causes an error, then that error will be "brought back" locally, and an error will result which will incorporate the remote message (of course, the stack and environment of the remote host is not "brought back"). A current limitation is that the PRIN4 form of <form>, and that of the result, must fit within one PUP -- about 530 bytes.

The argument <multiple.responses?> can currently be only either 0 or 1 (NIL defaults to 1); if 0, then the REMOTEVAL function will not wait around for the result from the remote host, but will return as soon as there has been an acknowledgement that the service is being performed; its value in this case will be the identifier number use for that transaction. This is especially useful when invoking a lengthy task which is done primarily for "effect" rather than value. Also it may be usefule when "broadcasting" some evaluation which will later directly send a note back to the initiator; thus there is no need to go through the several "handshake" packets, and general "broadcast" time-consuming protocols.

Also,
(REMOTEAPPLY <fun> <arglist> <serverhost> <multiple.responses?>)
invokes a similar remote use of APPLY (as opposed to EVAL)

Perhaps one may want to see how some server is progressing on its tasks:
(REMOTEVAL '(EVALSERVER.STATUS 'ALL) <server>)
will get the list documented above. If one decides to cancel some request, then
(REMOTEABORT <transaction> <serverhost>)
provides a convenient entry into the EVALSERVER.ABORT function at the remote host. <transaction> is either an id number, or a cons, as described above for EVALSERVER.ABORT; however, if only the id number is given, it is cons'd with the local host number before sending to the server.

If a lengthy computation is initiated on a server which is not running multiple processes, it would be a good idea for it to check the input queue from time to time, say by (EVALSERVER.STATUS 'INPUT), and explicity call (EVALSERVER 1)when there are waiting requests; for it may be that one of these waiting requests is an abort for the current process. Consider the example below, where

each time through the PROG loop (which is being sent for remote evaluation) there is a check for possible inputs, and a short call to EVALSERVER from that code itself, to insure that subsequent calls will have a chance to run.

```
74_(REMOTEVAL
     '(PROG ((CNT 0))
         LP (DISMISS 10000)
            (add CNT 1)
            (AND (EVALSERVER.STATUS (QUOTE INPUT))
                 (EVALSERVER 1))
            (GO LP))
     'PLAZA
     0)
55

75_(REMOTEVAL '(EVALSERVER.STATUS 'ALL) 'PLAZA)
((Completed.Transactions:
    ((ID#.ClientHost: 54 BuickoSaurus)
       (HowStop.#Seconds: COMPLETED 127)))
 (CurrentlyRunning.Transactions:
    ((ID#.ClientHost: 56 BuickoSaurus)
       (Process: RUNNING))
    ((ID#.ClientHost: 55 BuickoSaurus)
       (Process: RUNNING))))

77_(REMOTEABORT 55 ' PLAZA)
ABORTED
```

Note that the transaction id for this "lengthy" process was returned by the call to REMOTEVAL, since we requested no waiting around for the result (third arg of 0). Then when the remote EVALSERVER.STATUS was done, both that transaction, and the one actually causing the evalserver status to be read, show up as ID's 55 and 56. A subsequent remote abort for number 55 will stop it, and leave a record that it was remotely aborted. Of course, this is not necessary if the EVALSERVER is running under the PROCESSWORLD.

Some Implementation Details:

The semi-well-known socket number 668 is used for receiving eval service requests.