Communication Daemon (commd) and Communication Library (commlib)

Stefan Englhardt

April 19, 2001

1 Targets

Grid Engine 3.2 had problems with the tight coupling of Grid Engine components and the communication system. The mean problem was given when two server components (qmaster/execd) had to talk to each other. Deadlocks appeared which were resolved by watchdog timers (alarm()) interrupting the I/O calls. This gave a nondeterministic communication behavior which tends to collapse in case of a busy system.
Target of COMMD/COMMLIB is to loosen the dependencies between Grid Engine components and the communication layer. The server processes should only wait for the completion of a communication if its really necessary. Communication and real work should overlap. Availability of server processes should be as high as possible.
Ensure the process/process communication: Normal socket communication only ensures the delivery to a host. Delivery to a process on this host is not ensured as well. Communication with COMMLIB ensures, that if a communication call is done successfully, the receiver process has gotten the message in his process space. COMMLIB of course cant ensure the success of the communication itself, nor it can ensure that the receiver process does the right thing with what he got.

2 Components of the new communication system

The new communication system is interfaced by calls to a newly created library (commlib). Calls and parameters are prototyped in a header file.
This library contains socket communication calls. Via this library calls the client program talks to a responsible commd. The commd is a program handling all communications. It is a multiplexor which gets, holds and sends messages through socket pipes. Communication partners of the commd are commprocs linked with COMMLIB or other commds. The whole communication system is not limited to a single commd. There could be a lot of configuration possibilities. The structure of the whole system is limited (at this time) through the fact that the commd itself does no routing. The only information the commd has in order to forward a communication, is the information given by the sender. Thus there has to be a commd on the host of the receiver or the receiver is known by a commd the communication passes.

Extremities in commd configurations are, one commd on each host or one commd for all hosts.
To control and test the communication system there are some executables:

- tstsnd/tstrcv are clients interfacing the commlib and doing calls to the commlib driven by command line switches
- commdcntl is a client which allows the caller to do some sort of tracing communication, making a snapshot of commds data structures and shut down commds.

3 Addresses

In order to interact, communication partners have to know how to get in contact with the communication partner. In CODINE 3.2 this was done via host names and services. The initiator of a communication had to know on which host his communication partner lives and what service/port he uses. The initiator then makes a socket connection directly to the receiver.
The new communication system knows the communication partners as communication processes (commprocs). A commproc is identified by the host he lives on, by his name and by an identifying number. The number is needed for commprocs who could be started more than once at a time. This identifier makes the address unique. The name and the number are given by the commproc. If the commproc is not interested in a specific number he can leave the job of getting a unique number to the commd.
This addresses are agreed upon startup time of a commproc. Communication calls contain this triple to identify the communication partner.

4 How Communication Works

Communication is handled based on messages. The sender gives a message buffer and the length of this buffer to the communication routines. The contents is not interpreted by the communication system. The sender specifies an address for the receiver. This address can contain wild cards for the name and the id of the receiver. A wild card does not work like a multi cast. Only one receiver will get the message.
There are two modes to send and receive a message. Synchron and asynchron. A synchron send/receive blocks the caller/receiver until communication is done. An asynchron send/receive returns as soon as possible.
If sending synchron we wait for the receiver to get the message. The synchron send returns if an error occurs, a timeout takes place or the message is copied to the process space of the receiver successfully.
The asynchron send transfers the message to the commd and then returns immediately. The sender can test later whether the message arrived. It is not possible to ask for more than one asynchron message, cause there is only one field to store the last acknowledged asynchron message.

If receiving synchron the receiver blocks until a message arrives, an error condition occurs or a timeout takes place. This timeout can be set by the commproc using the set_commlib_param function.
If receiving asynchron the receiver gets a message if present at the commd. If there is no message, the receiver can immediately continue his work. This is a polling for a message.
To distinguish between different messages a receiving commproc can get, there is a field called "tag" in receive_message/send_message. This allows the receiver to wait for a specific message ignoring other messages addressed to him.

5 Interface

This is a summary of the commlib functions. They are prototyped in commlib.h. All functions return a status code. 0 means OK, all negative returns are error conditions positive returns may be return values of the called function (e.g. receive_message). The status codes are defined in commlib.h. To get an error string the function cl_errstr can be used.

int enroll(char *name, u_short *id)

Enroll to commd. Address of commproc will be set. If id=0 id returns an unused id. Addresses are unique. If the caller tries to get an used address the call will fail.

int leave(void)

Unregister from commd. The commproc has to ensure that leave is called before the commproc terminates.

int ask_commproc(char *host, char *commprocname, u_short commprocid)

Ask about a commproc. This is to ensure a given commproc is enrolled. Returns 0 if not.

int send_message(int synchron, char *tocomproc, int toid, char *tohost, int tag, char *buffer, int buflen, u_long *mid)

Send a message to another commproc.

- synchron: if 1 wait for completion of communication
- tocomproc, toid, tohost: Address of recipient
- tag: message type used to distinguish between messages
- buffer, buflen: message body to transfer
- mid: in case of asynchron messages this id is used to ask for the completion of the transfer

Wild cards are allowed for tocomproc and toid. Wild cards are NULL for a pointer and 0 for an integer.

int receive_message(char *fromcommproc, u_short *fromid, char *fromhost, int *tag, char **buffer, u_long *buflen, int synchron)

Receive a message.

- fromcommproc, fromid, fromhost: sender address
- tag: type of message
- buffer, buflen: returns allocated message and its length. The receiver has to free the message if it is no longer needed.
- synchron: if ==1 wait for a message to receive

Wild cards are allowed for fromcommproc, fromid, fromhost and tag. Wild cards are 0 for an integer and "" for a string. If a wild card is used, the corresponding parameter will be filled upon return with the data of the received message. fromhost has to be allocated to length MAX_HOSTNAME and fromcommproc has to be allocated to length MAX_COMPONENTNAME.

char *cl_errstr(int n)

Returns a pointer to an error string corresponding to the error number n, which is returned by commlib functions.

int set_commlib_param(int param, int intval, char *strval)

Set a parameter of the commlib interface. Possible parameters are:

- CL_P_RESERVED_PORT: secure mode: use reserved ports to communicate
- CL_P_COMMDHOST: set host the commd lives on
- CL_P_TIMEOUT: set timeout for communication routines
- CL_P_TIMEOUT_SRCV: set timeout for a synchron receive. This allows a server process to do some work on a regular time basis without getting a request.
- CL_P_COMMDPORT: set the port the commd is waiting on

int last_ack_message(u_long *mid)

Ask for the last acknowledged asynchron message. This number can be compared with the mid gotten from send_message().

int getuniquehostname(char *hostin, char *hostout, int refresh_aliases)

This is to get a unique host name out of a host name the commproc got from a side other than the commd. This function is for the purpose of comparing hosts. Because hosts can have aliases, a simple string compare may fail. For host names gotten from commd this is no problem, because they are unique. But if the user enters a host name to a commproc or the commproc asks for the name of the local host via gethostname(), this may not be a unique host name. Parameters are:

- hostin: host the commproc knows about
- hostout: returns the result
- refresh_aliases: if == 1 forces the commd to reread his alias file

int cntl(u_short cntl_operation, u_long *arg, char *carg)

This is a controlling interface used by commdcntl(). It should not be used in normal commprocs. Possible operations are:

- O_KILL: shutdown commd
- O_TRACE: trace messages passing commd (output is sent to stdout)
- O_DUMP: force commd to dump his structures to /tmp/commd/commd.dump
- O_GETID: get id of an enrolled process (carg==name of commproc, arg returns the id if the call is successful)
- O_UNREGISTER: unregister commproc (carg==name of commproc; arg==id)

This functions are accessible via commdcntl from the command line and should not be used in normal client programs.

6 Setting up a communication system

A simple example of a communication system can be build by commd, tstsnd and tstrcv. Simply start commd then tstsnd and tstrcv. Parameters are given below.
Commd usage (generated by commd -h):

```
usage: commd [-s service] [-p port] [-S] [-ml fname] [-ll loglevel] [-nd] [-a aliasfile]
             [-dhr]
             -s   use this service for connections from other commds [commd]
             -p   use this port for connections from other commds [commd]
             -S   enable port security
             -ml  message logging to file
             -ll  logging level
             -nd  do not daemonize
             -a   file containing host aliases
             -dhr disable regular hostname refresh
```

The default (but insecure) mode of operation is to simple start commd without parameters. This will start commd daemonized in the background. commd waits on a port specified by service "commd". So it is necessary to install a service using /etc/services or NIS before starting commd. If this is not possible -s or -p can be used to specify a service/port. "-ml" enables message logging to a file. "-ll" sets a logging level for output to the /tmp/commd directory. "-nd" holds commd in the foreground, so logging can be seen directly. "-S" starts commd in secure mode. In secure mode all communication partners use reserved ports to communicate. A reserved port can only be allocated by root.
tstsnd usage (generated by tstsnd -h):

```
usage: tstsnd [-s] [-w] [-host host] [-commproc name] [-id id] [-mt tag] [-enrollname name]
              [-enrollid id] [-t timeout] [-p commdport] [-S]
tstsnd        [-p commdport] -ae hostname name id name=any, id=0 means any id
tstsnd        [-p commdport] -uh hostname 0—1 search unique hostname/force commd to
                                              reread the alias file
              -s synchron send
              -w wait for acknowledge
              -host -commproc -id receiver address
              -mt message tag
              -enrollname -enrollid enroll with this address
              -t set timeout for communication
              -p port under which to contact commd
              -S secure mode
```

Tstsnd sends a small message to the given receiver. To receive this message tstrcv can be used. commdcntl -t can be used to trace what is going on.
tstrcv usage (generated by tstrcv -h):

```
usage: tstrcv [-s] [-host host] [-commproc name] [-id id] [-mt tag] [-enrollname name]
              [-enrollid id] [-t timeout] [-p commdport] [-S]
              -s receive synchron (wait until message arrives or TIMEOUT_SRCV happens)
              -t set timeout TIMEOUT_SRCV and TIMEOUT
              TIMEOUT_SRCV is the time we maximal wait in a synchron receive
              TIMEOUT is the time we maximal wait in a read on a communication file descriptor.
Specify target: -host host -commproc name -id id
Enroll with: -enroll name -enrollid id
              -S secure mode (use reserved ports)
```

commdcntl usage (generated by commdcntl -h):

```
usage: commdcntl [-k | -t level | -d] [-p commdport] [-S] [-gid commprocname]
                 [-unreg commprocname id]
                 -k kill
```

-t trace
-d dump structures to /tmp/commd/commd.dump
-p port commd is waiting on
-S secure mode
-gid get commproc id
-unreg unregister commproc

All clients look at the following environment variables. This environment variables are overruled by command line switches if present.

COMMD_HOST	Host commd lives on
COMMD_SERVICE	Service at which to access commd
COMMD_PORT	Port at which to access commd

Example of a simple communication system:

```
[25] commd
using service commd
bound to port 2222

[26] tstsnd -host balin -commproc tst
enroll returns 0 id=1
send_message returned: 0 mid=1
leave returned 0

[27] tstrcv -enrollname tst -mt 0
enroll returns 0
enrolled with id 11
rcv_message returned: 0
buflen = 6
buffer >hallo<
fromcommproc = tstsnd
fromid = 1
fromhost = BALIN.genias.de
tag = 0
leave returned 0

[28] commdcntl -k
```

7 Internals

The commd actions are data driven. In principle there are two structures that control operation. The first is the commproc structure. For each enrolling commproc commd creates

a commproc structure. The structure is freed if a commproc calls the function leave or if the commd considers the commproc dead. This happens if the commproc terminates while waiting in a synchron receive.

```
typedef struct commproc {
  char name[MAXCOMPONENTLEN];
  host *host;           /* pointer to host entry this commproc lives on */
  u_short id;                   /* identifier of this commproc */

  /* if commproc is waiting for a message the following describes the wait
     conditions */
  int w_fd;                     /* fd commproc is waiting on */
  char w_name[MAXCOMPONENTLEN]; /* commproc name we wait for */
  host *w_host;                 /* pointer to host entry we wait for */
  u_short w_id;                 /* commprocid we are waiting for */
  int w_tag;                    /* message tag to wait for */

  u_long last_ack_mid;          /* mid of last acknowledged message */
  u_long lastaction;            /* for timeout of commproc enroll */

  /* next commproc in list */
  struct commproc *next;
} commproc;
```

commd holds one list of commprocs describing the status of all commprocs. The w_... fields describe a wait condition. This allows the commd to decide in the moment a message arrives whether a commproc is ready to get this message. w_fd holds the open file descriptor of the connection to the commproc.
Keeping the fd open has the advantage that commd can recognize a breakdown of the commproc. A select on this fd shows activity a following read returns an error condition. commprocs breaking down are unregistered automatically.
last_ack_mid is filled when an acknowledge for an asynchron message comes in. It is overwritten be any following acknowledge. lastaction is a time value which allows to unregister a commproc in case of long inactivity. Because there is no elegant way to control processes on remote hosts commd needs a timeout mechanism for the commproc structure.

The message structure controls the state and flow of a message. There is one message list in commd containing all known messages. Messages are created at the time somebody makes a connection to the commd. Thus all communication with commd is done via messages. The message "scheduler" (process_received_message.c) is responsible for the decision what to do with a message. Message deletion is not done in a central place in the code. Messages are deleted when they are no longer needed. Deletion depends heavily on the sort of message (cntl/asynchron/synchron).

```
typedef struct message {
  commproc from;                /* commproc this message came from */
  u_long mid;                   /* message id for control of delivery */
  commproc to;                  /* commproc this message should reach */
  int tag;                      /* message tag (type) */
  u_long flags;                 /* kind and status of message */
  unsigned char *bufstart;      /* start of malloced buffer area */
  unsigned char *bufsnd;      /* pointer into buffer what is to send next */
  unsigned char *bufdata;       /* points to first data byte after header */
  unsigned char *bufprogress;   /* next to read or write */
  u_long buflen;                /* malloced size */
  u_short headerlen;            /* length of header */
  unsigned char prolog[8];      /* buffer for reading/writing the prolog */
  unsigned char senderkey[KEYLEN];      /* unused at the moment */
  char ackchar;                 /* char for ACK/NACK to sender */
  int fromfd;                   /* sfd of incoming message */
  int reserved_port;        /* true if message came from a reserved port */
  struct in_addr fromaddr;      /* address the message actually came from
                                   (not necessary the original sender) */
  int tofd;                     /* sfd for sending the message */
  u_long first_try;             /* first time of delivery; needed for error
                                   handling */
  struct message *next;
} message;
```

This structure holds all the information concerning a message. The structure and therefore the message goes through a lot of states in its lifetime. At each state different fields are modified. The intention behind this is to portion the tasks into a lot of pieces, so that processing the message can be interrupted and continued at any time. This is necessary due to the single threaded nature of commd. If the processing of a message is blocked due to a slow communication partner, the message processing will be freezed in its state and continued if the blocking condition vanishes.

There is a third structure needed for handling hosts and aliases.

```
typedef struct host {
    struct hostent he;          /* copy of what we got from gethostbyname */
    char mainname[MAXHOSTLEN];  /* This is what the administrator think it
                                   is the mainname */
    int deleted;                /* if we can no longer resolve this host */
    struct host *alias;         /* chain aliases */
    struct host *next;
```

```
} host;
```

Hosts entries are created when new hosts appear. This is e.g. when a message arrives from an unknown host. There is a single resolve via gethostbyname() when creating the structure. This information is refreshed on a regular basis. Old hosts will not be removed, because they may be referenced from within other structures. This is why we need the deleted field. The commd reads an aliasfile if he is started with the -a switch. If he cannot resolve a host or he is forced to do so by getuniquehostname, he rereads this file. The aliasfile has a simple structure. Every line contains a number of hosts, which are aliases to each other. Delimiters are blank or comma.

8 Host name resolving

Host name resolving is a difficult issue because there is no standard way to handle names in a network. Problems are:

- There are 3 different ways to specify the name to address resolution. DNS, NIS and the /etc/hosts-table.
- Each host can have a different translation for a host name.
- Resolver libraries have a different behavior.
- Machines with more than one network interface may have more than one entry in the resolver tables.

DNS is case insensitive, NIS and /etc/hosts arent. Some resolvers return different cases depending on the case of the input. The simpliest approach (this is what commd actual does) to this is not to distinguish between small and capital letters in hostnames at all.

Having the same name for different hosts is not a real failure of the network administrator. It may make sense to name a host "file_server". And if a Grid Engine installation crosses network boundaries there may be two file servers within a Grid Engine cluster. This situations can be handled by using fully qualified hostnames (e.g. file_server.acme.com). In case resolving tables return a short name and this name is ambiguous, the aliasing mechanism of commd helps. This aliasing mechanism can be used to overrule the resolving tables. Aliasing will be enabled by starting commd with an alias file (-a switch). The alias file contains one line per host which has to be aliased. A line "file_server.acme.com" forces commd to resolve this host at startup time and make "file_server.acme.com" the main name. This main name will be used by commd whenever the host is addressed (getuniquehostname;send/receive_message).

The alias file can be used to solve another problem. If a host has more than one interface, this can be handled in two ways. If DNS is used, it is possible to assign more then one address to a hostname. So fileserver.acme.com can have entered address 199.99.99.1 and 199.99.100.2 in the resolving table. There is no problem with this. But if DNS is not used...

To not use DNS doesn't mean to do not use DNS at all. A combination of DNS, NIS and /etc/hosts is possible. This is forced by the fact, that e.g. the resolver library of SunOS 4.1 can't talk to DNS without using NIS. In most installations with two interfaces in a host there is one entry in the resolver table per interface (e.g. fileserver_eth.acme.com and fileserver_fddi.acme.com). If this is the case it would be difficult for any component of Grid Engine to say explicitly this two hostnames belong to the same host. For at least security issues this is not tolerable. A line like "fileserver_eth.acme.com fileserver_fddi.acme.com" in the alias file will solve this problem.

To summarize the usage of an alias it can be sayed: The alias file helps solving ambiguities with hostnames. It can not be used for aliasing in general. A line "fileserver_eth.acme.com my_favorite_host" does not introduce a new name. If a name is not resolvable he will not be accepted.

In order to minimize resolving activities, commd holds a list of known hosts. Resolving is a relative expensive operation if it has to be done via network. This causes a problem if the resolving tables are changed while commd is running. Commd tries to refresh its host list on a regular basis. Nevertheless it may take a while that changes take effect. This can not be avoided due to the fact, that some resolver libraries cache resolving information. One have to take 10 minutes into account. A newly started commd of course gets the actual information. The automatical refreshing can be disabled by the "-dhr" switch.

9 Commd and files

Due to the nature of commd it is a problem handling file access within commd. File system access is in terms of computers a very slow operation. This can be even strengthened if the accessed file system is a networked file system. Commd can be blocked by this operations. So file system access is minimized. In normal operation without aliases there is no file access at all. If there are aliases there are accesses every time a host cannot be resolved. This can be caused by ill commprocs and has to be avoided. Other file system accesses are for the purpose of logging messages to file. This is done into /tmp/commd if existent (not created by commd). /tmp is usually a local file system. Accessing local file systems should not slow down the operation of commd in a disturbing fashion.