Network Working Group J.E. Donnelley
Request for Comments: 712 Lawrence Livermore Laboratory
February 1976

A Distributed Capability Computing System (DCCS)

This paper was prepared for submission to the international Conference on Computer Communication, ICCC-76, August 3, 1976, Toronto, Canada.

This is a preprint of a paper intended for publication in a journal of proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited without the permission of the author.

The work reported in this paper was supported in part under contract #EPA-IAG-D5-E681-DB with the Environmental Protection Agency and in part under contract #[RA] 76-12 with the Department Of Transportation. The report was prepared for the U.S. Energy Research and Development Agency under contract #W-7405-Eng-48.

A Distributed Capability Computing System (DCCS)

This paper describes a distributed computing system. The first portion introduces an idealized operating system called CCS (Capability Computing System). In the second portion, the DCCS protocols are defined and the processes necessary to support the DCCS on a CCS are described. The remainder of the paper discusses utilizing the DCCS protocol in a computer network involving heterogeneous systems and presents some applications. The applications presented are to optimally solve the single copy problem for distributed data access and to construct a transparent network resource optimization mechanism.

The Capability Computing System (CCS)

The CCS, though not exactly like any existing operating system, is much like some of the existing capability list (C-list) operating systems described in the literature [1-7]. Many of the features of the CCS come from a proposed modification to the RATS operating system [1-3].

In the documentation for most computer systems there are many references to different types of objects. Typical objects discussed are: files, processes, jobs, accounts, semaphores, tasks, words, devices, forks, events, etc. etc.. One of the intents of C-list systems is to provide a uniform method of access to all such objects. Having all CCS objects accessed through a uniform mechanism allow DCCS to be implemented in a type independent manner.

The CCS is a multiprocessing system supporting an active element called a process. For most purposes, the reader's intuitive notion of what a process is should suffice. A process is capable of executing instructions like those in commercially available computers. It has a memory area associated with it and has some status indicators like "RUN" and "WAIT". In C-list systems, however, a process also has a capability list (C-list). This list is an area in which pointers to the objects that the process is allowed to access are maintained. These pointers are protected by the system. The process itself is only allowed to use its C-list as a source of capabilities to access and as a repository for capabilities that it has been granted. Figure 1 diagrams some typical processes that are discussed later. In the diagrams, the left half of a process box is the C-list and the right half is the memory.

The key to the uniform access method in the CCS is the invocation mechanism. This is the mechanism by which a process makes a request on a capability in its C-list. An invocation is closely analogous to a subroutine call on most computer systems. When a request is made, the invoking process passes some parameters to a service routine and receives some parameters in return.

There are, however, several major differences between the invocation mechanism and the usual subroutine calling mechanisms. The first difference is that the service routine called is generally not in the process's memory space. The service routine is pointed to by the protected capability and can be implemented in hardware, microcode, system kernel code, in another arbitrary process, or, as we shall see in the DCCS, in another computer system. In Fig. 1. for example, the serving process is servicing on invocation on the semaphore requestor.

A second difference is that, when invoking a capability, other capabilities can be passed and returned along with strictly data parameters. In the DCCS, capabilities and data can also be passed through a communication network.

The final important distinction of the invocation mechanism can best be illustrated by considering the analogy to the outside teller windows often seen at banks. These windows usually contain a drawer that can be opened by the customer and teller are not both. Except for this drawer, the customer and teller are physically isolated. In the case of the invocation mechanism, the invoking process explicitly passes certain capabilities and information to the service routine and designated C-list locations and memory areas for the return parameters. Except for these parameters, the invoking process and the serving routine are isolated. In the DCCS, this protection mechanism is extended throughout a network of systems.

In the CCS, invoking a capability is the only way that a process can pass or receive information or capabilities. All of what are often referred to as system calls on a typical operating system are invocations on appropriate capabilities in the CCS. A CCs C-list envelopes its process. This fact is needed in order to transparently move processes as described in the second application on network optimization (page 23).

CCS Capabilities

To build the DCCS, we will assume certain primitive capabilities in the CCS. The invocations below are represented for simplicity rather than for efficiency or practicality. In practice, capabilities generally have more highly optimized invocations with various error returns, etc.. To characterize a capability, it suffices to describe what it returns as a function of what it is passed. In the notation used below, the passed parameter list is followed by a ">" and then the returned parameter list. In each parameter list the data parameters are followed by a "" and then the capability parameters.

1. File Capability

   a. "Read", index; > data;

      "Read" the data at the specified index. "Read" and the index are passed. Data is returned.

   b. "Write", index, data; > ;

      Write the data into the area at the specified index. "Write", the index, and the data are passed. Nothing is returned.

2. Directory Capability

   a. "Take", index; > ; capability

      "Take" the capability from the specified index in the directory. "Take" and the index are passed. The capability is returned.

b. "Give", index; capability> ;

"Give" the capability to the directory at the index specified. "Give" and the index are passed information. The capability is also passed. Nothing is returned.

c. "Find"; capability> result, index;

A directory, like a process C-list, is a repository for capabilities. The first two invocations are analogous to the two file invocations presented except that they involve capability parameters moved between directory and C-list instead of between file and memory. The last invocation searches the directory for the passed capability. If an identical capability is found, "Yes" and the smallest index of such a capability are returned. Otherwise "No" and 0 are returned.

3. Nil Capability

When a directory is initially created, it contains only nil capabilities. Nil always returns "Empty".

4. Process Capability

a. "Read", index; > data;

b. "Write", index, data; > ;

c. "Take", index; > ; capability

d. "Give", index; capability> ;

e. "Find"; capability> result, index;

f. "Start"; > ;

g. "Stop"; > ;

The a. and b. invocations go to the process's memory space. C., d., and e. go to its C-list. F. and g. start and stop process execution.

The CCS Extension Mechanism

There is one more basic capability mechanism needed for the CCS implementation of the DCCS. This mechanism allows processes to set themselves up to create new capabilities that they can service. Such mechanisms differ widely on existing C-list systems. A workable mechanism is described. Another primitive capability is needed to start things off:

5. Server Capability

   a. "Create requestor", requestor number; > ; requestor

   b. "My requestor?"; capability> answer, requestor number;

   c. "Wait"; > reason, requestor number, PD; request

Two capabilities were introduced above besides the server, the requestor and request capabilities. These capabilities will be described as the invocations on a server are described.

The first invocation creates and returns a requestor capability. The number that is passed is associated with the requestor. The requestor capability is the new capability being created. Any sort of invocation can be performed on a requestor. This is their whole reason for existence. A process with a server capability can make a requestor look like any kind of capability.

The "My requestor?" invocation can be used to determine if a capability is a requestor on the invoked server, it returns either:

   "Yes", requestor number; or "No",0;

The last invocation "Wait"s until something that requires the server's attention happens. There are two important events that a service routine needs to be notified about. If the last capability to a requestor is overwritten so that the requestor cannot again be invoked until a new one is created, the "wait" returns:

   "Deleted", requestor number, 0; Nil

The last two parameters, 0 and Nil, are just filler for the returned PD and request (see 5c). When a "wait" returns "Deleted", the service routine can recycle any resources being used to service the numbered requestor (e.g., the requestor number).

The most important event that causes a "wait" to return is when one of the requestors for the server is invoked. In this case the server returns:

   "Invoked", requestor number, PD; request

The third parameter, labeled PD, stands for Parameter Descriptor. It describes the number of each kind of parameter passing each way during a requestor invocation. Specifically, it consists of four numbers: Data bits passed, capabilities passed, data bits requested, and capabilities requested.

The last parameter received, the request capability, is used by the serving process to retrieve the passed parameters and to return the requested parameters to the requesting process. Accordingly, it has the following invocations:

6. Request Capability

   a. "Read parameters"; > {The passed parameters

   b. "Return", {The return parameters}> ;

The "Return" invocation has the additional effect of restarting the requesting process.

One thing that should be noted about the server mechanism is that invocations on a server's requestors are queued until the server is "wait"ed upon. This is one reason that a request is given a separate capability. The serving process can, if it chooses, give the request to some other process for servicing, while it goes back and waits on its server for more requests. The corresponding situation in the outside bank window analogy would be the case where the teller gives the request to someone else for service so that the teller can return to waiting customers. The request capability points back to the requesting process so that the return can be properly effected.

A sample service, that of the well known semaphore [8] service routine keeps a table containing the semaphore values for each semaphore that it is servicing. It also keeps a list of queued requests that represent the processes that become hung in the semaphore by "P"ing the semaphore when it has a value less than or equal to zero. The invocations on a semaphore are:

7. Semaphore

   a. "P"; > ;

   b. "V"; > ;

A diagram and flow chart for the semaphore serving process is given in Figures 1. and 2. The flow charts are given include most of the basic capability invocations, but do not include detailed descriptions of table searches. The table structure for the semaphore service routine includes entries for each supported semaphore. Each entry contains the semaphore value and a link into a list of pointers to the requests hung in the semaphore (if any).

The most important feature of the server mechanism is that, by using it, the functioning of any capability can be emulated.

This property, similar to the insertion property discussed in [9], is the cornerstone of the DCCS. The basic idea of the emulation is to have the server "wait" for requests and pass them on to the capability being emulated. Such emulation of a single capability is flow charted in Figure 3. The emulation flow charted is an overview that doesn't handle all situations correctly. For example, a capability may not return to invocations in the same order that they are received. These situations also appear in the DCCS, so their handling will be discussed there rather than here. It is important to note that, except for delays due to processing and communication, the emulation done in the DCCS is exact.

The DCCS Implementation

The DCCS will initially be described on a network of CCS systems. We will assume that there exists a network capability:

8. Network Capability

   a. "Input"; > Host no., message;

   b. "Output", Host no., message > ;

   It is assumed that the "Output" invocation returns immediately after queuing the message for output and that the "input" invocation waits until message is available.

For pedagogical purposes, the description of the DCCS will be broken into two parts. First a brief overview of the important mechanisms will be given. The overview will gloss over some important issues that will be resolved individually in the more complete description that follows the overview.

The intent of the DCCS is to allow capabilities on one host to be referenced by processes on other hosts having the appropriate capabilities. To do this, each host keeps a list of capabilities that it supports for use by other hosts. Each host also supports a server, which gives out requestors that are made to appear as if they were the corresponding capability supported by the remote host. When one of these emulated requestors is invoked, its parameters are passed by the emulating host through the network to the supporting host. The supporting host then sees to it that the proper capability is invoked and passed the parameters. When the invoked parameters are passed back through the network to the emulating host. The emulating host then returns the return parameters to the requesting process.

For example, let us take the "Read" request on a file diagrammed in figure 4. When the emulated file (a requestor) is invoked, the emulating process receives "invoke", requestor number, PD; request. The requestor number that is returned is actually a descriptor consisting of two numbers: Host number, capability number. These descriptors are called Remote Capability Descriptors (RCDs). An RCD identifies a host and a capability in the list of capabilities supported by that host. After receiving a request, the emulating process reads the parameters passed by the requesting process and sends them along with the Parameters Descriptor to the remote host in an "invoke" message.

When the remote host receives this information, it passes the parameters to the supported file capability by invoking it and specifies the proper return parameters as noted in the Parameter Descriptor. When the invoked file return parameters, the returned data is passed back through the network to the emulating host in a "Return" message. The returned data is then returned to the requesting process by performing a "Return" invocation on the request capability initially received by the emulating host. When the requesting process is awakened by the return, it will appear to it exactly as if a local file had been invoked.

This works fine when the parameters being passed and returned consist simply of information, but what happens when there are capabilities involved? In this case the routines use the existing remote capability access mechanism and pass the appropriate descriptor. As an example, the "Take" invocation on a directory is diagrammed in figure 5. The only essential difference is the fact that a capability has to be returned. When the capability is returned by the invoked directory (or whatever it really is), the supporting host allocates a new slot in its supported capability list for the capability and returns a new descriptor to the emulating host. When the emulating host receives the descriptor, it creates a new requestor with the returned descriptor as its requestor number and returns the requestor to the invoking process. The requestor so returned acts as the capability taken from the remotely accessed directory and can be invoked exactly as if were the real capability.

One important thing to notice about this mechanism is that neither the emulating host nor the supporting host need to have any idea what kind of capabilities they are supporting. The mechanism is independent of their type. Also important is the fact that neither host need trust the other host with anything more than the capabilities that it has been rightfully granted. Even the DCCS processes themselves need only be trusted with the network capabilities and with the supported capabilities. Finally, note that no secret passwords which might be disclosed are needed for security. The DCCS directly extends the CCS protection mechanisms,

A more complete description of the DCCS will now be given. To avoid unnecessary complication, however, several issues such as error indications, system restart and recovery, network malfunctions, message size limitations, resource problems, etc. are not discussed. These issues are not unique to the DCCS and their solutions are not pertinent here.

As noted earlier, the complete DCCS must address several issues that were glossed over in the initial overview. As these issues are discussed, several message types are introduced beyond the "Invoke" and "Return" messages discussed in the overview. The formats for all the DCCS messages are summarized in figure 6.

A. Timing -

Invocations can take a very long time to complete. We saw an example in the semaphore capability earlier. An even more graphic example might be a clock capability that was requested to return nothing AFTER 100 years had passed. Clearly we don't want to have the emulating process wait until it receives a "Return" message from the remote host before servicing more invocations.

What is done in the emulating host is to add the request capability to a list of pending requests after sending the "invoke" message to the supporting host (this is somewhat like the semaphore example earlier). The emulator can then go back and wait for more local requests.

There is a similar problem on the supporting side. We don't want the process waiting on the network input capability to simply invoke the supported capability and wait for return. What it must do is to set up an invocation process to actually invoke the supported capability so that pending network input can be promptly serviced. The invoking process must then return the parameters after it receives them.

These additional mechanisms add complication of multiple requests active between hosts. These requests are identified by a Remote Request Number (RRN). The RRN is an index into the list of pending requests.

B. Loops -

If host A passes a capability to host B, and B is requested to pass the requestor that is being used to emulate the capability back to host A, should B simply add the requestor to its support list and allow A to access it remotely? If it did, when the new requestor was invoked on A, the parameters would be passed to B where they would be passed to the requestor by the invoking process. Invoking the requestor would cause the parameters to be passed back through the network to A where the real capability would finally be invoked. Then the return parameters would have to go through the reverse procedure to get back A via B. This is clearly not an optimal mechanism,

The solution to this problem makes use of the "My requestor?" invocation on a server capability described in 5b. When B checks a capability that is to be returned to A with the "My requestor?" invocation and finds that the capability is one of its requestors with a requestor number indicating that it is supported on A, it can simply return the requestor number (recall that is this is really a Remote Capability Descriptor, RCD) to A, containing the fact that the capability specified is one that is local to A and giving A the index to the capability in its supported capability list.

C. Security

The mechanism presented in B. brings up something of a security issue. If B. tries to invoke a capability in A's supported list, should A allow B access without question? If it did, any host on the network could maliciously invoke any capability supported by any other host. To allow access only if it has been granted through the standard invocation mechanism, each host can maintain a bit vector indicating which hosts have access to a given capability. If a host does receive an invalid request, it is an error condition.

D. Indirection

There is an additional twist on a Loop problem noted in B.. This variation comes up when A passes a capability to B who then wants to pass it to C. Here again B may unambiguously specify which capability is to be passed by simply sending the Remote Capability Descriptor (RCD) that is knows it by. The RCD indicates that the capability, however, A would probably not believe that C should have access to it.

B must tell A. "1, who have access to your 1'th capability, want to grant it to host C". To do this, another message type is used. The "Give" message specifies the supported capability and the host that it should be given to (refer to figure 6). Here again, giving away a capability that you don't have is an error condition.

E. Acknowledgement -

There is one last problem with the "Give" message. If B sends the "Give" message to A and then continues to send the Remote Capability Descriptor (RCD) to C, C may try to use the RCD before the "Give" is received by A. For this reason, B must wait until A has "ACK"nowledged the "Give" message before sending the RCD to C. This mechanism requires that hosts queue un"ACK"nowledged "Give"s. The format for an "ACK" is given in figure 6. This queueing may be avoided for most "Give"s after the first for a given RCD, but only at the cost of much additional memory and broadcasting "Delete"s (See F. below).

F. Deletion -

If all the requestors on A for a given capability supported on B are deleted. A may tell B so that B may:

a. Delete A's validation bit in the bit vector for the specified capability and

b. If there are no hosts left that require support of the given capability, the capability may be deleted from the supported capability list.

This function requires a new "Delete" message.

Figure 6 is a summary of the message formats. Figure7-11 flow chart the complete DCCS. In the flow charts, abbreviations are used to indicates the directories:

CSL - Capability Support List

RRL - Remote Request List

IPL - Invocation Process List

The table manipulation is not given in detail. Three tables are needed. The first is associated with the CSL and contains the bit vectors indicating access as noted in C. above. The second table is associated with the RRL. It contains a host number for each active request. An attempted return on a request by a host other that the requested host is an error. The final table is a message buffer containing the pending "Invoke" and "Return" requests.

In order to avoid hazards in referencing the CSL and its table, a semaphore called the CSLS is used. A message buffer semaphore, MBS, is similarly used to lock the message buffer. For the RRL and IPL no locks are needed with the algorithms given.

Generalization and Application

To implement the DCCS, we assumed a network of CCS systems. The specifications of the CCS were, however, very loose. For example, no mention was made of instruction sets. Any CCS-like implementation could use the mechanisms described herein to snare their objects. A process passed to system with a different instruction set, for example, could be used as an efficient emulator.

The most important generalization of the DCCS is to note that a given implementation has no idea what kind of host it is talking to over the network. Any sort of host could implement a protocol using the messages given. For example, a single user system might allow its user to perform arbitrary invocations on remote capabilities and keep a table of returned capabilities. Such a system might also support some kind of standard terminal capability that could be given to remote processes. On a multi-user system, similar functions could be performed for each user.

In some sense, any system implementing the DCCS protocol becomes a C-list system. The single user system could, for example, set up remote processes servicing remote server capabilities giving out requestors to the single user system or any other systems. Returns from invocations could appear on the single user's terminal by remote invocation of the terminal capability, etc..

Implementing the DCCS on non-C-list systems is similar in some respects to what happened with some host to host protocol implementations on the Department Of Defense's ARPA network [10]. The ARPA network host to host protocols allows a process on one system to communicate with a process on another. Many of the ARPA net protocol implementations had the effect of introducing local process to process communication in hosts that formerly had none.

Applications

I. Single Copy

The first application is a solution to what I have dubbed the single copy problem for information resources. Whenever a process receives information from a information resource, it can only receive a local copy of the information. This fact is apparent when the information come from a distributed data base, but is also true in tightly coupled virtual memory situations where information from shared memory must be copied into local registers for processing. Once a process has a local copy of some information, it might like to try to insure that the information remains current, i.e., that it is the single copy.

The traditional solution to this problem is to lock the information resource with a semaphore before making a local copy and then invalidate the local copy before unlocking the resource. This solution suffers from the fact that, even though other processes may not be requesting the copied data, the data must be unlocked quickly just in case. This can result in many needless copies being made.

What is needed is a mechanism for invalidating local copies exactly when requests by other processes would force invalidation. To offer such a mechanism, an information resource can have, in addition to the usual reading and writing invocations, the following:

```
"White lock", portion; > ; write notify

"RW lock", portion; >; RW notify
```

The important invocation on the notify capabilities is:

```
"Wait for notification"; > reason;
```

The basic idea is to allow a process to request that it be notified if an attempt is being made to invalidate its copy. If the copy is used for reading only, the process need only request notifications of attempted modifications of the data ("Write lock"). When a process is so notified, it is expected to invalidate its copy and delete its write notify capability to inform the information resource server that the pending write access may proceed.

In the read write lock case, the RW notify capability may also be used for reading and writing the portion. Any other access to the portion will cause notification. When notified, the process with the RW notify capability is expected to write back the latest copy of the information before deleting its RW notify capability.

Space does not permit presenting more details for this mechanism. The important fact to notice is that it permits an information resource to be shared in such a way that, though the information may be widely distributed, it is made to appear as a single copy. This mechanism has important applications to distributed data bases.

II. Network Resource Optimization

The application that probably best demonstrates the usefulness of the DCCS is the sort of network optimization capability that can be used to create at least the primitive capabilities introduced earlier:

9. Account Capability

   a. "Create", type; >; capability

   The passed type parameter could at least be any of: "File", "Directory", "Process", or "Server". The appropriate type of capability would be returned. The resources used for the capability are charged to the particular account.

Now suppose that a user on one CCS system within a DCCS network has remote access to account capabilities on several other CCS systems. This user could create what might be called a super account capability to optimize use of his network resources. The super account capability would actually be a requestor serviced by a process with optimization desired would be completely under user control, but some of the more obvious examples are presented:

1. Static Object Creation Optimization

   a. When a new file is requested, create it on the system with the fastest access or the least cost per bit.

   b. When a process is requested, create it on the system with the fastest current response or with the least cost per instruction.

2. Dynamic optimization.

   To do dynamic optimization, the super account would not give the requesting process the capability that it received from the remote account after its static optimization, but would give out a requestor that it would make function like the actual capability except optimized.

   a. When network conditions or user needs charges, files can be moved to more effective systems. changes in cost conditions might result in file movement. Charges in reliability conditions might result in movement of files and/or in addition or deletion of multiple copies.

   b. If system load conditions or CPU charges change, it might be effective to relocate a process. The super account service process could: create a new process on a more effective system, stop the old process, move the old C-list and memory to the new process and start the new process up. The emulation process given to the user would never appear to change.

   c. Similar optimizations can be done on any other capabilities.

   Such a super account can automatically optimize a user's network resources to suit the user's needs without changing the functional characteristics of the objects being optimized.

Final Note

The DCCS mechanisms defined in this paper are currently being implemented on a Digital Equipment Corporation PDP-11/45 computer for use as an experimental protocol on the ARPA computer network [10]. The DCCS protocol will also form the basis for a gateway between the ARPA network and Energy Research and Developement Agency's CTR network [11]. It is the authors hope that the DCCS mechanism will hasten the approach of the kind of networks that are needed to create a truly free market in computational resources.

Acknowledgements

The author would like to thank the administrators and staff of the Computer Research Project at the Lawrence Livemore Laboratory for creating the kind of environment conductive to the ideas presented in this paper. Special thanks are due to Charles Landau for many of the C-list ideas as implemented in the current RATS system.

References

1. C. R. Landau, The RATS Operating System, Lawrence Livermore Laboratory, Report UCRL-77378 (1975)

2. C. R. Landau, An Introduction to RATS (RISOS/ARPA Terminal System): An Operating System for the DEC PDP-11/45, Lawrence Livermore Laboratory, Report UCRL-51582 (1974)

3. J. E. Donnelley, Notes on RATS and Capability List Operating Systems, Lawrence Livermore Laboratory, Report UCID-16902 (1975)

4. B. W. Lampson, "On Reliable and Extendable Operating Systems", Techniques in Software Engineering, NATO Sci Comm. Workshop Material, Vol. II (1969)

5. W. Wulf, et. al., "HYDRA: The Kernel of a Multiprocessor Operating System", Communications of the ACM 17 6 (1974)

6. P. Neumann et. al., "On the Design of a Provably Secure Operating System" International Workshop on Protection in Operating Systems, IRIA (1974)

7. R. S. Fabry, "Capability-Based Addressing", CACM 17 7 (1974)

8. E. W. Dijkstra, "Cooperating Sequential Processes", published in Programming Languages, F. Genuys, editor, Academic Press, pp. 43-112 (1968)

9. F. A. Akkoyunlu, et. al., "Some Constraints and Tradeoffs in the Design of Network Communications", Proceedings of the Fifth Symposium on Operating System Principles, Vol. 9 No. 5 pp. 67-74 (1975)

10. L. G. Roberts and B. D. Wessler, "Computer Network Development to Achieve Resource Sharing", AFLPS Conference Proceedings 36, pp. 543-549 (1970)

11. "National CTR Computer Center", Lawrence Livermore Laboratory Energy and Technology Review, Lawrence Livermore Laboratory UCRL-52000-75-12, December (1975)

The figures are not included in the online version. Interested readers can obtain a hardcopy version of the documents including the figures by requesting a copy of UCRL-77800 from:

Technical Information Department
Lawrence Livermore Laboratory
University of California Livermore, California 94550

Questions or comments would be appreciated and should be directed to the author:

Though the U.S. mail:

James E. Donnelley
Lawrence Livermore Laboratory L-307
P. O. Box 808
Livermore, California 94550

By telephone:
(415)447-1100 ext. 3406

Via ARPA net mail:
JED@BBN

"This report was prepared as an account of work sponsored by the United States Government. Neither the United States nor the United States Energy Research & Development Administration, nor any of their employees, nor any of their contractors, subcontractors or their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness or usefulness of any information, apparatus, product or process disclosed, or represents that its use would not infringe privately-owned rights."