Technical Report
BUCS-TR-1993-020

AIDA-Based Distributed File System[§]

Azer Bestavros
Mohammad Makarechian

Computer Science Department
Boston University
Boston MA 02215

[§] This work was conducted under the supervision of Professor Azer Bestavros (best@cs.bu.edu) in the Computer Science Department as part of Mohammad Makarechian's Master's project.

1 Purpose of Study

The goal of the study is to implement a file server for a distributed environment based on *AIDA* (*A*daptive *I*nformation *D*ispersal *A*lgorithm) [1]. We are particularly interested in measuring and evaluating the end product's performance in response to requests from applications with time constraints.

2 Overview of *AIDA*

AIDA is a dynamic bandwidth allocation scheme for a distributed environment designed with two main motivations. First, it will service requests from all applications regardless of their time basis, but the target application is one with time-constraints. For example, a multimedia system which stores and reads its data files from remote (and distributed) databases would utilize *AIDA*'s capabilities particularly well. Such a system would be classified as real-time and also, will not be tolerant to site/network failures. In other words, it must be guaranteed a certain quality of service throughout a session. Second, as computer systems proliferate, security becomes an important factor. *AIDA* offers a high level of protection against security violations. In what follows, we present a concise description of *AIDA*.

AIDA operates on an object which in our discussion will be treated as a file F. The size of F is denoted by $|F|$. F is to be stored in a distributed fashion by being dispersed over several sites. The sites are interconnected by a communications network [1]. If there are N sites available for the dispersal of F [2], and F can be reconstructed from the components stored in only m sites ($m \le N$), *AIDA* distributes F over N sites, i.e., distributes N copies of a $\frac{|F|}{m}$ block, but reconstruct F from n sites, where $m \le n \le N$. Therefore, there is a $\frac{n-m}{n}$% redundancy introduced. In other words, F is inflated by a factor of $\frac{n}{m}$. The redundancy introduced by the algorithm provides a certain level of availability for the application requesting F. Specifically, the service becomes tolerant to the failure of n-m sites. The actual process of dispersal is based on *IDA* [2], which presents an effective method of distributing a file over a certain number of sites by choosing a set of "secret key" vectors. *IDA* provides both, a fault-tolerant and secure dispersal of a file.

2.1 Obtaining n and Other Parameters

We assume a lower bound on total communication delay (i.e., forward+processing+return) of D_{min}. Therefore, in order to calculate the expected communication delay, we treat D_{min} as a constant or lower bound, added to the random variable that represents the jitter associated with the sites, Y''. It is further assumed that the communication delay of the

[1] We are not concerned with the type or topology of the network at this point.
[2] N does not necessarily have to be the same for all files.

network can in general be modeled by an exponentially distributed random variable Y' for each single transaction processed by a single site, and so there exist N independent such variables overall. Thus, Y, the random variable representing the amount of time spent to process a request is related to Y'' and $D_{\min}$ by : $Y = D_{\min} + Y''$

First, each site can be characterized by the following equations:

$$f(z) = \lambda \cdot e^{-\lambda z} \quad \text{for } z \geq 0 \quad (1)$$

$$F(z) = P(Y' \leq z) = 1 - e^{-\lambda z} = 1 - e^{-\lambda(z - D_{\min})} \quad \text{for } z \geq D_{\min} \quad (2)$$

$$\therefore P(Y' > z) = 1 - F(z) = e^{-\lambda(z - D_{\min})} \quad \text{for } z \geq D_{\min} \quad (3)$$

Meanwhile, the expected value of the second component of Y is:

$$E(Y'') = \int_{D_{\min}}^{\infty} z \cdot f(z) \cdot dz = \int_{D_{\min}}^{\infty} (1 - F(z)) \cdot dz = \int_{D_{\min}}^{\infty} P(Y > z) \cdot dz \quad (4)$$

$P(Y > z)$ is calculated by a series of Bernoulli trials, where success is defined as a request that takes more than time z. This situation (time > z) occurs when more than n-m sites show delay larger than z. Therefore, such probabilities are calculated and added together for the range of station failures from n-m+1 to n *. In the following equations, the value of P is given by the distribution of the random variable Y' (see (1)-(3)).

$$P(Y > z) = \sum_{r=n-m+1}^{n} \binom{n}{r} \cdot P^r \cdot (1-P)^{n-r} \quad (5)$$

$$\therefore E(Y'') = \int_{D_{\min}}^{\infty} \sum_{r=n-m+1}^{n} \binom{n}{r} \cdot P^r \cdot (1-P)^{n-r} \cdot dz$$

$$\therefore E(Y) = D_{\min} + \int_{D_{\min}}^{\infty} \sum_{r=n-m+1}^{n} \binom{n}{r} \cdot P^r \cdot (1-P)^{n-r} \cdot dz \quad (6)$$

It can be shown that (6) can be reduced to:

$$\therefore E(Y) = D_{\min} + \frac{1}{\lambda} \sum_{r=n-m+1}^{n} \frac{1}{r} \quad (7)$$

---

* These events are mutually exclusive.

In order to obtain n, we need T_{max}^{F}, a user-defined[3] time-constraint on requesting file F (theoretically: $D_{min} < T_{max}^{F} \leq \infty$; more realistically: $E(Y) \leq T_{max}^{F} \leq \infty$).

$$\frac{m}{1-e^{-\lambda(T_{max}^{F}-D_{min})}} < n \leq N \quad (8)$$

2.2 Parameter Definitions

Equation (8) involves several parameters which will be defined as applicable to implementation of the server. Consider the file F:

- N — super-user-specified parameter -- represents the maximum number of sites over which F can be dispersed.
- m — super-user-specified parameter -- depends on the availability level and priority of a request and is defined when F is written (dispersed).
- n — determined from (8) -- the number of site over which F is dispersed.
- λ — determined based on statistics on network's past performance -- represents the mean total communication delay (forward+processing+return) per request.
- D_{min} — determined based on statistics on network's past performance[4] -- represents the lower bound on the total communication delay per request.
- T_{max}^{F} — user-specified parameter -- represents the time-constraint imposed on a request with the possibility of being rejected by the server (if $T_{max}^{F} \leq D_{min}$).

3 Problem Definition

"Implement a server which disperses a file F over several sites. This process requires knowledge of N and m. Service a request issued by the user to read F by reconstructing it from n sites. This process requires knowledge of λ, D_{min}, and T_{max}^{F}."

4 Interface and Design

The server consists of one main process. As implied by the description of parameters in the previous section, feedback plays an important role in implementing the server. The

---

[3] Details of defining T_{max}^{F} are discussed later.

[4] Therefore it has a dynamic nature and must be updated. (See section 4.)

ultimate decision of determining n is based on data that requires regular updating. It is decided to construct lookup tables for this purpose, and this calls for daemon processes which contain the tables and perform various operations on them.

In order to distribute the control element of the server as much as possible, there will be replications of these tables over all N sites each containing the following tables:

• Table 1: Name and address mapping

- for each file dispersed, N location-identifier blocks are allocated n of which will be used.
- each location-identifier can contain the system address[5] of a site or, in order to reduce storage and overhead, it can be an index in which case a means of resolving such indices must be provided (e.g., an array where each element contains the system address).
- port number of each site's lookup-table process will be contained in each location-identifier block

• Table 2: Site data

- for each site (1,2,...,N) the following parameters are needed: number of requests for that site[6] S_i, total communication delay (forward+processing+return) for that site upon processing the most recent request.

• Table 3: Description of open files

- the file's name
- the file's user-imposed time constraint (if any), T^F_{max} in seconds
- the file's user-imposed availability level, x
- *AIDA* parameter m
- size of the file, in bytes

These tables will be both, updated and used, by the server in an interrupt-driven fashion. Interrupts will be time-based, at the software level (signals in Unix terminology). As mentioned before, these tables must be replicated over all sites to provide maximum reliability and correctness in decisions made by the (stateless) server. This is not a trivial task since consistency issues will be introduced and have to be dealt with. The server, through the process of updating the tables, will verify the availability of each site regularly. Therefore, it will need a data on the availability of sites.

The server operates in two **modes**: *user* and *super-user*. It will always start in the super-user mode, possibly providing a password security measure. The reason for starting in the

[5] Refers to addresses which can be used by Remote Procedure Call (RPC) routines.

[6] This number naturally has an upper bound which when exceeded (steady-state) can be held constant at that value.

super-user mode is that the server must be configured, and to do so, it will ask for inputs. A possible sequence of events taking place at start-up is:

- request N
- based on N, request for each site's system address (or ask for an initialization file name which contains the addresses)
- update or initialize site address-index maps if the data structure is out-dated or uninitialized
- update and use lookup table data by interrupting processes containing the tables or being interrupted by them
- change mode to *user*

The above scheme provides location transparency for the user[3]. The commands supported by the server (and command parameters; "-" indicates optional parameters; order is important) are:

To open a file:

• `open` F $-T^F_{\max}$ $-x\%$

F	file name: Unix file naming conventions apply; F will be entered in a similar data structure as Unix open-files table
$T^F_{\max}$	time constraint: default is ∞ or no time constraint (in seconds)
x	percentage availability or fault-tolerance below which processing of the request must be aborted

To write (disperse) a file (can be done only once):[7]

• `write` F m

F	file name: Unix file naming conventions apply
m	parameter m (see section 2)

To read (reconstruct) a file:

• `read` F

F	file name: Unix file naming conventions apply; F must be open (by previously issuing `open`) and it may produce an output showing (simulating) the contents of F

To close a file:

• `close` F

F	file name: Unix file naming conventions apply; F must be open (by previously issuing `open`)

Thus it can be seen that another data structure must be accessed by the server which contains a list of open files with the various (if any) options and requirements specified by the user (and super-user).

---

[7] Since this is actually a super-user operation, the server may warn of change of mode or ask for super-user password.

An object-oriented approach will be taken in designing and implementing the server. One advantage of such an approach is that future enhancements or modifications to the server become easier due to clean and implementation-independent interfaces that is associated with every object. The distributed paradigm will be remote procedure call, and specifically, Sun RPC (Remote Procedure Call). A motivation for using RPC is the high level of network transparency that it provides for the programmer. Some knowledge of the underlying communications network is nevertheless necessary.

5 File Server Implementation

The program consists of two parts: a local segment and a remote segment. The remote segment must be run as a daemon process (in background) in each site and therefore acts like a server responding to the client. The client is the local segment of the program and must be run as a shell, just like other popular Unix shells such as the C shell. The file server[8] , called *DFSTC* (*D*istributed *F*ile *S*ystem with *T*iming *C*onstraints) is therefore the local segment. These two segments are formed from a series of code files (written in "C") but share two common header files. For more details on the contents of the files, and how to obtain and compile them, see **Files**.

As mentioned before, it was attempted to adopt an object-oriented (OO) methodology for *DFSTC*. However, it soon became evident that the design would be far from OO standards but the gain was considerable in terms of the modularity, and clean and well-defined interfaces that resulted. This made the debugging stage considerably more manageable by providing the ability for the programmer to isolate errors and locate them in a particular module. A possible disadvantage could be a slight increase in function overhead.

DFSTC revolves around two tables of data (and statistics): Tables 2 and 3[9]. The purpose and (logical) contents of the two tables were described previously. In order to conduct manipulation of the tables in a consistent and uniform manner, it was necessary to have various utilities routines which hid the details of table elements as much as possible from the caller. For these tables, there are the following "methods" defined (see **Manual Pages** for more details):

(1) Add-Entry: adds an initialized (i.e., empty) table entry
(2) Write-Entry: fills in an initialized table entry with specified values
(3) Delete-Entry: removes the specified table entry
(4) Get-Entry: searches and obtains the specified table entry
(5) Update-Entry: searches and updates the specified table entry with the new specified values

[8] The word "server" in this case, is not intended to be interpreted in the client-server terminology. It simply refers to the fact that the local segment provides a service to the user and not to other processes, as is the case in the client-server context.

[9] Table 1 was found to be redundant but nevertheless, it can be used in future versions of the program.

In addition to the above, there is an auxiliary routine for each table that displays the contents of the table, entry by entry, on the screen. These routines are currently installed in *DFSTC* for demonstration purposes.

The core of the program, however, consists of four functions which implement the user commands described in the previous section: Open, Write, Read, and Close. Of these functions, Write requires the user to have super-user access, which is defined by password protection. This mechanism which also appears initially during the start-up phase of *DFSTC* gives the user three[10] chances to enter the correct password. A successful verification of the password at the start-up phase allows the user to enter a list of the N sites used by the program. This is done manually by typing each site's IP address until all N addresses have been entered.

At this point the program has been properly initialized, except for the fact that Table 2, site statistics, is empty of any valid data. An important aspect of the server is to calculate n based on various parameters including the mean and minimum site total communication delays. Therefore, until proper data are collected, *DFSTC* assumes that n is equal to m. This decision makes sense with reference to equation (8): λ and D_{min} cannot be determined and so it is necessary to make an assumption.

A file F that is to be processed by *DFSTC* must first be "written" by issuing a Write command. This action is to be done only once. After providing the necessary parameters for Write the program writes F to all N sites[11] in a parallel fashion by creating sub-processes each assigned to one site. A site, upon receiving a Write request, creates a file labeled *.dfstc* which contains records of files with the following format:

(1) File name
(2) m

This is done to enable the local site to be as independent of any data as possible in order to make it tolerant to server crashes. For example, after writing a file F, if the server (or site) crashes, it is not necessary to re-write F upon reboot of the program because each of the N sites has already received their allocated block and kept a record of the associated parameter, m. A similar reasoning is applied to the actions that follow at the completion of every command: updating of all the tables at each of the N sites. Essentially, each of the three tables is copied to all the sites, entry by entry. At each site, when an update request is received, depending on the table, (maximum) of four sequences are possible:

(1) If *DFSTC* signals an initialization, the local table is initialized to empty
(2) If *DFSTC* signals a Close, the specified entry is removed from the local table
(3) The local table is searched, and if the specified entry is found, it values are updated with the new specified values

---

[10] This number can be modified by changing the `MAX_PASSWD` constant in `dh.h`.

[11] Ideally, before dispersal, each block must be transformed through the *IDA* algorithm [2], but at the time of writing this report, *IDA* was not implemented in *DFSTC*.

(4) If the specified entry is not found during (3), it is added to the local table

Once a file has been dispersed, from now on, it can be read from the sites. The first step in reading a file is to open it. This is done by issuing an Open command and providing the required parameters. Opening a file results in it, along with all the necessary parameters, being added to Table 3, the table of open files. First, a check is made to see if the file has already been opened. If not, the program dispatches N processes, in parallel, to query each site for the following two parameters:

(1) m: obtained from the local *.dfstc* file
(2) dispersal size: determined by obtaining the size of a local, previously dispersed block

These data are necessary for properly filling in the contents the a Table 3 entry. Parameter (2) is then multiplied by (1) to calculate the actual (undispersed) size of the file. Since all of the N sites will provide identical sets of parameters (1) and (2), it is only necessary for the program to record the first set that it receives and ignore the others. The remaining data are user-specified parameters entered during the issuing of Open. It is worthwhile to elaborate on the purpose of one of the user parameters: the availability level. This is a measure of the tolerance to site crashes that the user is willing to accept expressed in percentage. Recall that n sites will be called upon to provide their dispersed blocks in order to reconstruct the original file. Out of these n blocks, only m are necessary. By specifying an availability level, the user is able to specify the desired level of redundancy.

When reading a file, first D_{min} must be determined using the data in Table 2 and this is followed by evaluating λ. Having D_{min} and λ, *DFSTC* is now ready to calculate n based on equation (8), and it then provides the Read routine with n, along with the other necessary parameters. Read goes on to request a block (of dispersal size) from each site but only processes the first m that it receives since that is all it needs to reconstruct the original file. A new parameter that is a by-product of Read is the total communication delay of the system. This is the largest total communication delay among the first m sites to provide the program with their blocks, and is a measure of the time it takes for the system, viewed as a single entity to service a user request.

Closing a file is similar to updating remote sites, in the sense that, in addition to having the file removed from the local Table 3, each site must be updated from this change. The program does this by raising a flag to signal the removal of the entry to each of the N sites. Again, this is done by spawning N processes that (virtually) run in parallel. Once a file is closed, it must be re-opened for further Reads.

Finally, the program can be exited by issuing a "quit-shell" command, and if confirmed, the shell is aborted and all the data stored locally is lost. However the sites which are still alive contain their copies of the tables, and as an important further enhancement to the program, upon re-boot, *DFSTC* could have the ability to request each site for a copy of each table. Another adjustment is to add to the tables of open files, Table 3, at write-time, as opposed to open-time, to cut down on the number of requests sent to the remote sites. Since all the methods for Table 3 is provided, this should not prove to be time-consuming.

6 Performance Tests

In order to evaluate the effectiveness of *AIDA*[1], the algorithm which *DFSTC* is based on, three test cases were carried out, details of each is explained in the following subsections. All three of the cases have these parameters in common:

- Original file size, $|F|$: 114688 bytes
- m: 1
- N: 3; IP addresses of sites:
  - (1) csa.bu.edu (Boston University, Boston, MA)
  - (2) edson.ee.ualberta.ca (University of Alberta, Edmonton, Canada)
  - (3) spiderman.bu.edu (Boston University, Boston, MA)
- User-imposed timing constraint, T^F_{max}: none specified
- Availability level, x: 1.0 (100%)
- Notes:
  - - F was read 5 times for each case
  - - n was hard-coded and therefore T^F_{max} was not specified
  - - N/U: Not Used; refers to a site that was not contacted by *DFSTC*
  - - N/A: Not Applicable

Case 1: n = m (= 1)

Total Communication Delay (in seconds)				System Parameters (in seconds)		
Run	Site (1)	Site (2)	Site (3)	D_{min}	λ	System Delay
1	13	N/U	N/U	13	N/A	13
2	11	N/U	N/U	11	N/A	11
3	13	N/U	N/U	13	N/A	13
4	12	N/U	N/U	12	N/A	12
5	11	N/U	N/U	11	N/A	11

Case 2: m < n (= 2) < N

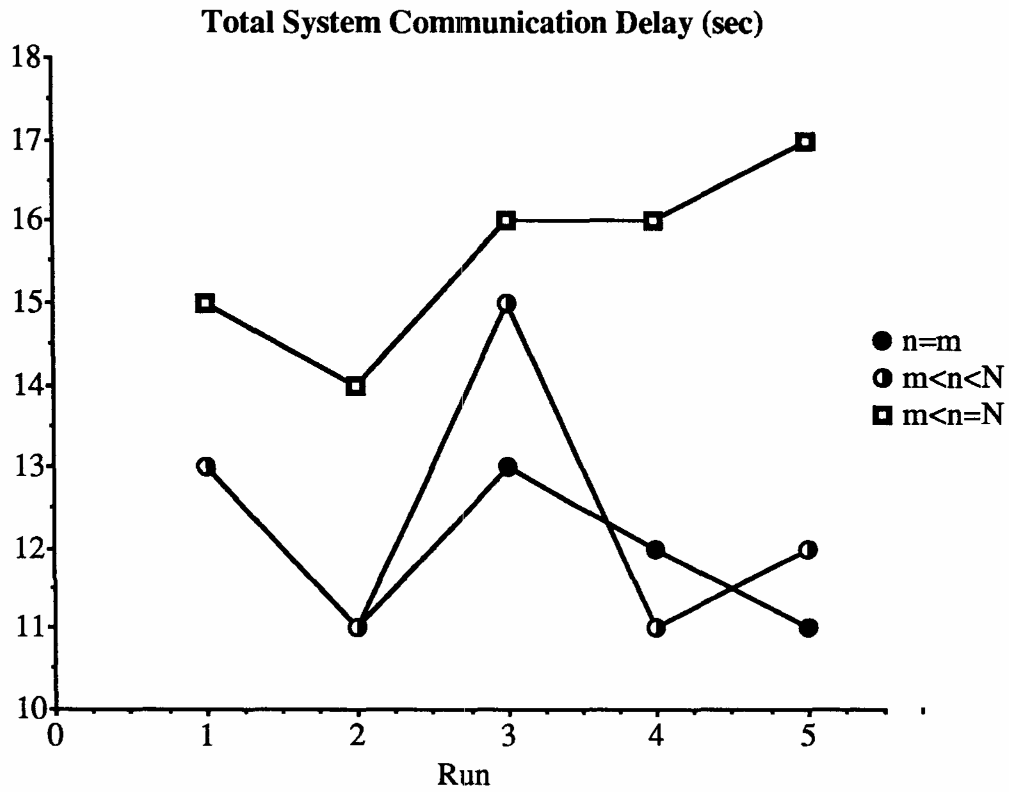
Total Communication Delay (in seconds)	System Parameters (in seconds)

Run	Site (1)	Site (2)	Site (3)	D_{min}	λ	System Delay
1	13	167	N/U	13	N/A	13
2	11	185	N/U	11	N/A	11
3	15	114	N/U	15	N/A	15
4	11	231	N/U	11	N/A	11
5	12	196	N/U	12	N/A	12

Case 3: m < n = N (= 3)

Total Communication Delay (in seconds)				System Parameters (in seconds)		
Run	Site (1)	Site (2)	Site (3)	D_{min}	λ	System Delay
1	21	198	15	15	78	15
2	14	179	17	14	70	14
3	16	191	19	16	75	16
4	16	362	17	16	132	16
5	17	416	18	17	150	17

A brief summary of the statistics on the system delay for the three cases is presented next. Notice the stability of the standard deviation of the system delay. It is desirable to increase the total number of sites, *N*, available for dispersal. In this experiment, there is a wide disparity between the delays of the first and third site (they are on the same internet) and that of the second site, which is accessed through a wide-area network. The analysis would be more telling if the number of remote sites was increased from 1 to 2, at least, to achieve a balance.



X_1 : n=m

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
12	1	.45	1	8.33	5

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
11	13	2	60	724	0

X_2 : m<n<N

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
12.4	1.67	.75	2.8	13.49	5

Minimum:	Maximum:	Range:	Sum:	Sum of Sqr.:	# Missing:
11	15	4	62	780	0

$X_3 : m<n=N$

Mean:	Std. Dev.:	Std. Error:	Variance:	Coef. Var.:	Count:
15.6	1.14	.51	1.3	7.31	5
Minimum:	**Maximum:**	**Range:**	**Sum:**	**Sum of Sqr.:**	**# Missing:**
14	17	3	78	1222	0

7 Files

The following is a list of the files that make up *DFSTC*. These can be copied from the path `~/makar/demo/` in the "bigbird" cluster at the department of Computer Science's Distributed Lab in Boston University. The RPC (Remote Procedure Call) specification file, `~/makar/demo/dfstc.x` must be processed by typing the command `rpcgen dfstc.x`. This results in the creation of a header file, `dfstc.h`, an XDR data conversion template file `dfstc_xdr.c`, the client stub `dfstc_clnt.c`, and the server stub `dfstc_svc.c`. The program can then be compiled by using the `make` facility by typing the command `make dfstc`, for *DFSTC* (the local segment), and `make server`, for the remote server segment. Make sure that the file `~/makar/demo/makefile` is located in the same directory as all the required source files.

- File name: `dc2.c`
- Contents: C source code for:
    - `write_data_1()`
    - `open_file_1()`
    - `read_data_1()`
    - `update_table_1()`
    - `update_table_2()`
    - `update_table_3()`

- File name: `dc3.c`
- Contents: C source code for:
    - `Password()`
    - `Command()`
    - `Get_Int_Param()`
    - `Get_Float_Param()`
    - `Cons_Sysad()`
    - `Write_Sysad()`
    - Table 1 methods:
        - `Add_Entry_T1()`
        - `Write_Entry_T1()`
        - `Del_Entry_T1()`
        - `Get_Entry_T1()`
        - `Update_Entry_T1()`

- File name: `dc4.c`
- Contents: C source code for:
  - Table 2 methods:
    - `Add_Entry_T2()`
    - `Write_Entry_T2()`
    - `Del_Entry_T2()`
    - `Get_Entry_T2()`
    - `Update_Entry_T2()`

- File name: `dc5.c`
- Contents: C source code for:
  - Table 3 methods:
    - `Add_Entry_T3()`
    - `Write_Entry_T3()`
    - `Del_Entry_T3()`
    - `Get_Entry_T3()`
    - `Update_Entry_T3()`

- File name: `dc6.c`
- Contents: C source code for:
  - `main()`

- File name: `dc7.c`
- Contents: C source code for:
  - `Write_Data()`
  - `Open_File()`

- File name: `dc8.c`
- Contents: C source code for:
  - `Cal_Dmin()`
  - `Cal_Lambda()`
  - `Cal_N()`
  - `Read_Data()`

- File name: `dc9.c`
- Contents: C source code for:
  - `Close_Remote()`
  - `Update_Remote()`

- File name: `dc10.c`
- Contents: C source code for:
  - `Print_Table_2()`
  - `Print_Table_3()`

- File name: `dh.h`

- Contents: This is a header file containing defined include files, defined constants and macros, and function prototypes.

- File name: `dfstc.x`
- Contents: This is the RPC specification file; see above for instructions.

- File name: `makefile`
- Contents: This is the file that the `make` facility uses in order to properly compile and manage *DFSTC*.

8 References

[1] Bestavros, Azer. "AIDA-based Communication for Distributed Time-Critical Applications". Computer Science Department, Boston University, Boston, MA. January 20, 1993: Technical Report 92-020.

[2] Rabin, Michael O. "Efficient dispersal of information for security, load balancing and fault tolerance." Journal of the Association for Computing Machinery. 36(2) (April 1989):335-348