Research Applications Program

National Center for Atmospheric research

RAP / NCAR

DIDSS

Data Ingest and Distributed Server System

High-level design

Mike Dixon

May 19, 1999

File /home/dixon/rap/libs/didss/doc/didss_design.mif

1. Overview.

This document describes the design of the Data Ingest and Distributed Server system (DIDSS) for data ingest and handling. DIDSS will be used by a number of projects at RAP for data management and access.

1.1. System components.

The system is made up of the following components:

- the **data store**, which comprises data sets stored on disk at locations relative to a well-known point in the file system, in much the same way as Web server data is organized;
- **clients** which either need to put data to the store or get data from the store;
- **servers** which provide clients access to the store;
- a **server manager** which starts servers in response to client requests, and provides information to clients on how to connect to the servers;
- a **data mapper** which keeps a list of available data in the store;
- a **scout** which searches through the store to keep the data mapper up to date;
- a **janitor** which cleans up the store, if required, compressing or removing files to keep disk space available for new data;
- a **system monitor** which presents diagnostic system status information to the user.

The use of servers allows (1) distributed access to data and (2) distributed intelligence, by hiding from the client unnecessary details about the data set.

Communication within the system is done either via the file system or TCP/IP sockets. The ‘hidden’ components which glue the system together are the function calls which the various components make to communicate with each other.

A data set is referenced by a Uniform Record Locator (URL) similar to those used for http and ftp access on the Web.

1.2. Shared data region.

A **shared data region** (SDR) is any networked region with sufficient bandwidth for efficient communication between distributed servers. Generally this means a fast LAN, (at least 10 mbit/s), although slower links may be included in the SDR if the performance across those links is satisfactory for the needs of the project.

The system may be contained on a single host or spread across multiple hosts.

In a single host system, the SDR encompasses only that single host. In such cases all data sets are local.

In multiple host systems, the SDR includes all hosts. A copy of the server manager, data mapper, scout and janitor runs on each host. Data sets and their servers are distributed across the SDR. All data mappers within the system are synchronized - when a mapper on a given host receives updated information about a data set it communicates that information to the other mappers in the SDR.

Figure 1 shows the components for a single-host system. Each of these components is

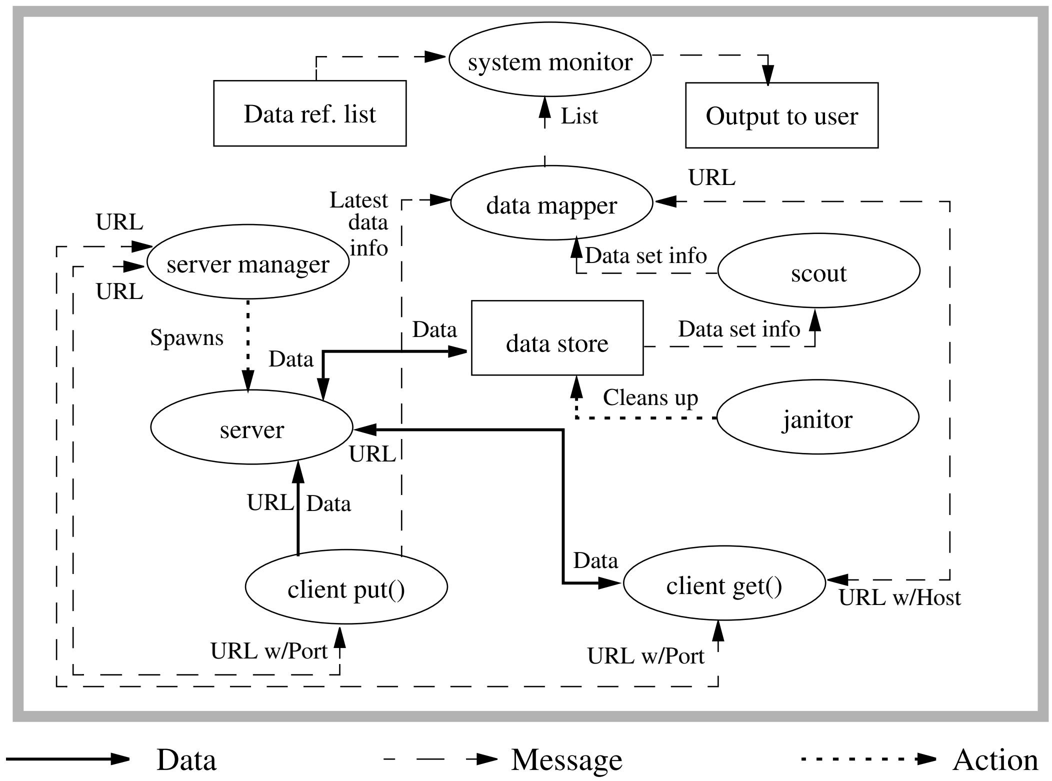

Figure 1. Components in a single-host system.

described in detail, along with their interactions, later in the document.

1.3. Byte-ordering.

All data in system is represented in big-endian (XDR) byte order. This applies to data in the store and to messages passed.

2. Uniform Record Locators - URLs.

Required data services are specified using a URL-style address.

The URL syntax is:

protocol:translator:params//host:port:file?args

The **protocol** is required. The **translator, params, host, port, file** and **?args** are optional. All delimiters except the **?** are required.

The optional **translator**, if present, defines the program to be used to translate the data into the desired protocol and form.

The optional **params**, if present, defines the param file to be used by the server or the translator. The parameter file must exist in the root directory for the data type. The parameter file name on disk is computed by concatenating the server name and the param name, separated by a period, with the entire name prefixed with an underscore (see example below).

If the **host** is missing, the data mapper must be contacted to determine the host name. If the **host** is local, the service may in some implementations be provided by reading to and writing from the disk directly.

If the **port** is missing, the server manager must be contacted to determine the port number. If the **port** is present, contact with the server is made via this specified port.

The **?args** is an open-ended string for passing extra information to the server API. It is not used frequently.

URL examples are:

- Visible satellite MDV data on virga, port to be determined from the server manager:

  `mdvp:://virga::sat/vis`

- IR satellite MDV data - host to be determined from the data mapper, port from the server manager:

  `mdvp:://::sat/ir`

- Visible satellite MDV data from virga using server on port 11000:

  `mdvp:://virga:11000:sat/vis`

- Colide boundaries SPDB data on babinet, port from the server manager:

  `spdbp:://babinet::boundaries/colide`

- TITAN tz30 SPDB data, on vil, port from the server manager, using the Titan2Symprod translation process executable with this name running under the "*$RAP_DATA_DIR/spdb/Titan2Symprod.mult_forecasts*" parameter file:

  `spdbp:Titan2Symprod:mult_forecasts//vil::titan/tz30`

3. The data store.

The data store is made up of files stored on disk.

3.1. Data types.

Any number of data types may be included in the system. The following data types have already been identified:

- mdv - gridded data in MDV format;
- spdb - symbolic product data in SPDB format;
- titan - storm track data in TITAN format;
- grib - gridded data in grib format;
- cedric - gridded data in Cedric format.

Other file types may be added as required.

3.2. Directory structure.

The top of the data directory tree is defined by the environment variable `$RAP_DATA_DIR`.

The various main data types are stored in subdirectories of `$RAP_DATA_DIR`. For example, you could expect to see the following subdirectories:

`$RAP_DATA_DIR/mdv` - all mdv data

`$RAP_DATA_DIR/spdb` - all spdb data

`$RAP_DATA_DIR/titan` - all titan data

`$RAP_DATA_DIR/grib` - all grib data

etc.

The various data sets would be stored in subdirectories of these main directories. For example:

`$RAP_DATA_DIR/mdv/sat/vis` - visible satellite data

`$RAP_DATA_DIR/mdv/radarCart` - Cartesian radar data

`$RAP_DATA_DIR/spdb/boundaries` - colide-derived boundaries

`$RAP_DATA_DIR/spdb/ltg` - lightning

`$RAP_DATA_DIR/spdb/surface/metar` - METARs

`$RAP_DATA_DIR/titan/tz30` - TITAN 30dBZ storm data

3.3. Data directory and file names.

The names of all special non-data files, such as parameter files, should start with an underscore. This will safeguard them from compression or removal by the janitor.

The directories and files within a data store may be named using any convention. However, in order to make optimum used of generic server and data library functions, the following conventions should be observed where possible.

3.3.1 One set of files per day.

In this case all of the files reside in the data set directory. The files should be named:

yyyymmdd.ext,

where *yyyy* is the year, *mm* the month, *dd* is the day, and *ext* is the appropriate extension.

For example, SPDB files are stored in this manner using *yyyymmdd.indx* for the index files and *yyyymmdd.data* for the data files.

3.3.2 Multiple files per day.

In this case the files for a particular day reside in a subdirectory, which should be named:

yyyymmdd/

Within that directory, the files should be named:

hhmmss.ext

where *hh* is the hour, *mm* the minute and *ss* the second.

For example, radar MDV files are normally stored in this manner. The file:

mdv/radarCart/19980623/152321.mdv

is a cartesian radar volume file for 15:23:21 UCT on 23 June 1998.

3.3.3 Multiple files per day, with multiple forecast times.

Some data sets, such as model data, contain forecasts at multiple lead times created at multiple generate times.

In this case the files generated on a particular day reside in a subdirectory, which should be named:

yyyymmdd/

Within that directory, there is a subdirectory for each generate time. The subdirectories should be named:

g_hhmmss/

where the '*g*' indicates that this is a generate time subdirectory.

Within that directory, the files should be named:

f_ssssssss.ext

where the '*f*' indicates that this is a forecast time file. The 8-digit numerical name indicates the forecast lead time in seconds, which is valid for lead times of up to 3 years.

For example, the file:

mdv/mm5/19980725/g_150000/f_00007200.mdv

is a 2-hour (7200 seconds) MM5 model forecast generated using data at 15:00:00 UCT on 25 July 1998.

3.3.4 Compressed files.

The janitor will compress files to save disk space. Therefore, any of the data files may be compressed using gzip, which will add the '.gz' extension to the name.

4. Clients.

A client is any program which requires service from DIDSS. The primary operations are (1) putting data to the data store and (2) getting data from the data store. The details of these functions are governed by the data type. Nevertheless a number of details apply to all such operations, regardless of data type.

4.1. The put function.

A client controls the put function using the URL. The following sequence is followed:

- **host**: if the host is not specified, the client contacts the local data mapper to resolve the host name. In most cases the host is known, since most components of the system put the data to a pre-determined location;
- **port**: if the port is not specified, the client contacts the server manager at the specified host to resolve the port number. The manager starts the relevant server if it not already running. It adds the port to the URL and returns the resolved URL to the client;
- **server**: the server is contacted and the put message is passed to the server for action.

If the host is local, the put implementation may choose to write directly to disk instead of contacting the server.

In real-time mode, the routine which writes the data to disk also contacts the data mapper and sends it the latest data information. If the server writes to disk, its write routine contacts the mapper. If the client performs a local write, this routine contacts the mapper.

Figure 2 shows the communication during a put using a server.

Figure 3 shows the communication during a put which writes directly to disk.

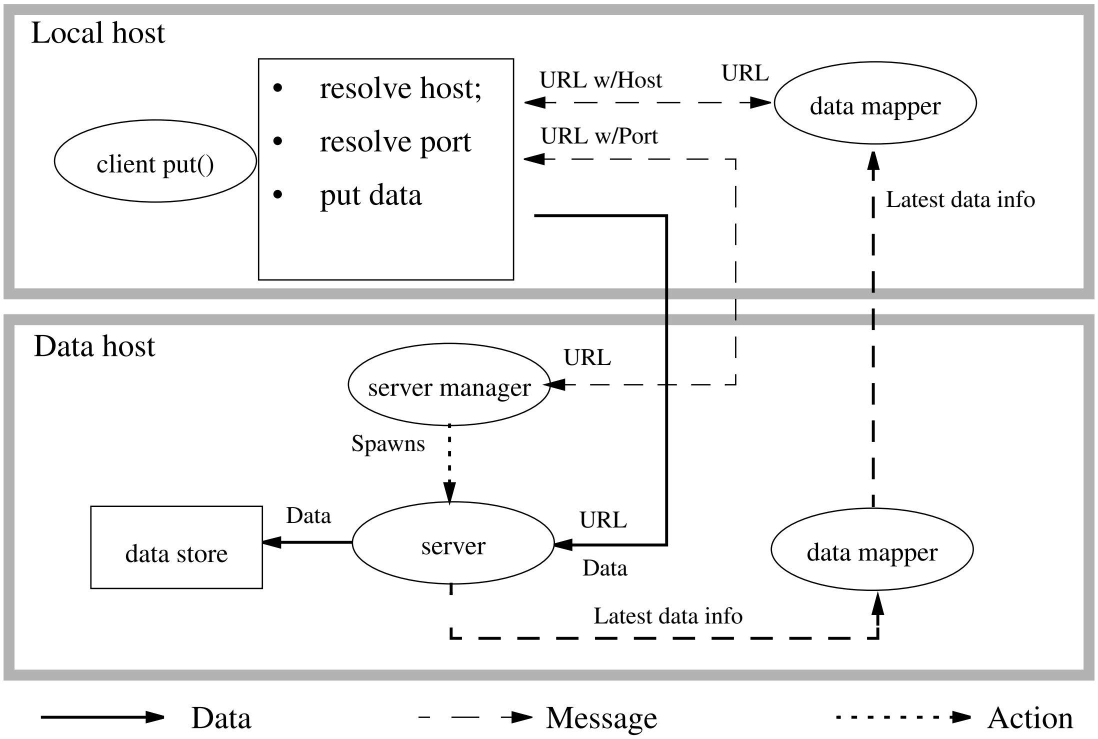

Figure 2. Communications during a put() using a server.

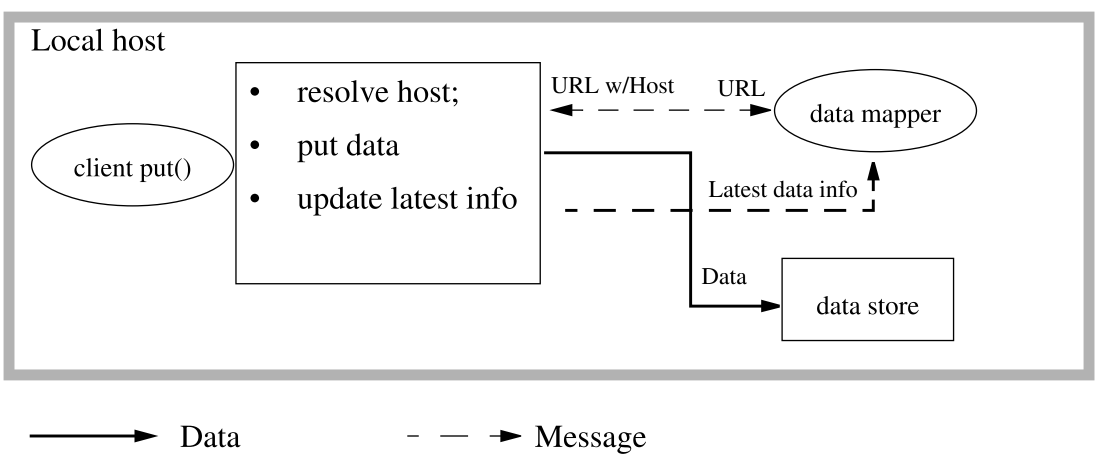

Figure 3. Communications during a put() which writes directly to disk.

4.2. The get function.

The get function involves similar steps:

- **host**: if the host is not specified, the client contacts the local data mapper to resolve the host name. The URL is passed to the data mapper which fills in the host and returns the resolved URL;
- **port**: if the port is not specified, the client contacts the server manager at the specified host to resolve the port number. The manager starts the relevant server if it not already running. It adds the port to the URL and returns the resolved URL to the client;
- **server**: the server is contacted, and the get message is passed to the server for action. The server carries out the request and returns the data to the client.

If the host is local, the get implementation may choose to read directly from disk instead of contacting the server.

Figure 4 shows the communication during a get using a server. Figure 5 shows the communication during a get which reads directly from disk.

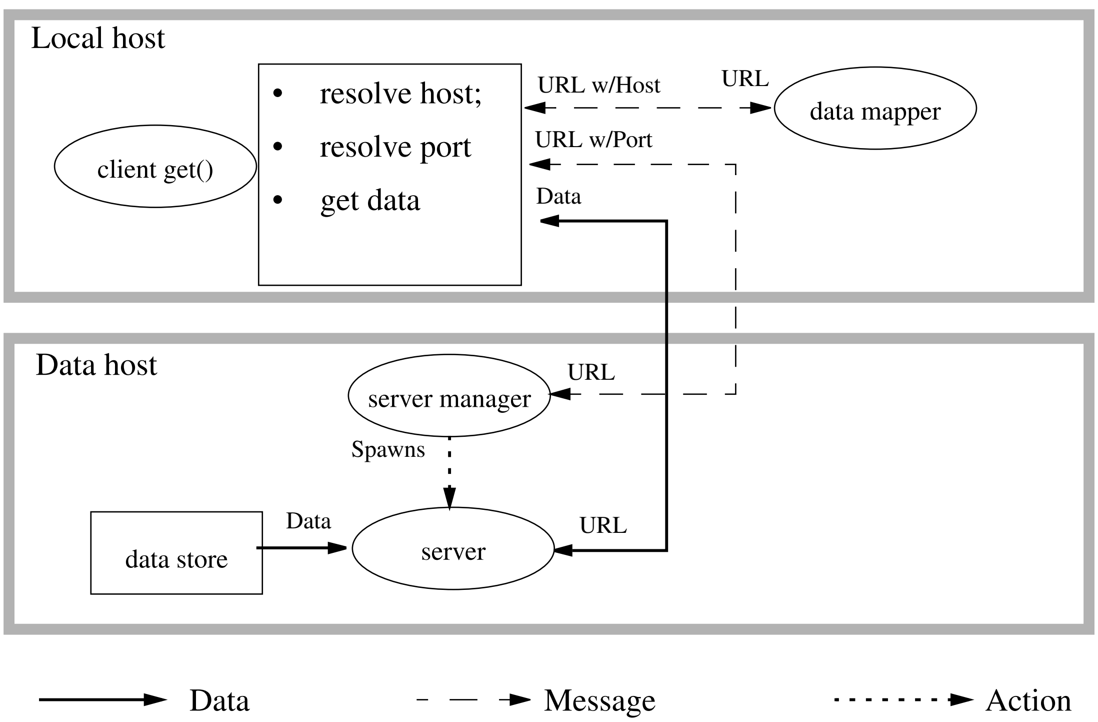

Figure 4. Communications during a get() using a server.

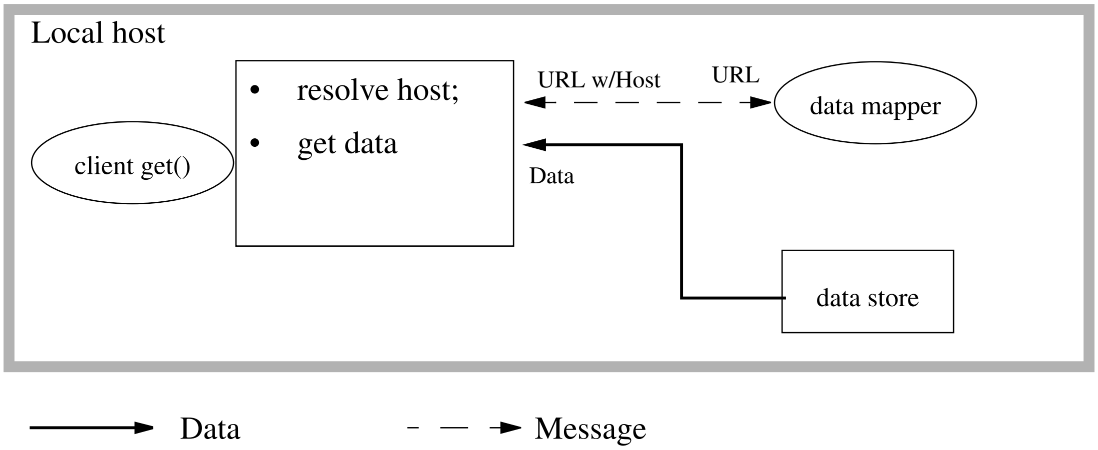

Figure 5. Communications during a get() which reads directly from disk.

4.3. Rules for resolving host and port.

For URL's with an unspecified hostname, the the data mapper is contacted to obtain the hostname. For URL's with an unspecified port number, the server manager is contacted to obtain the correct port.

Both the hostname and port in the URL may contain special key values which determine whether or not server communications is required.

4.3.1 Special behavior for URL hostName.

- if hostname = "localhost", *server_contact* = false.

4.3.2 Special behavior for URL port.

- if a non-zero port is specified, *server_contact* = true, use the user-specified port;
- if a port of zero is specified, *server_contact* = true, use the port returned from the server manager;
- if the port is left unspecified, *server_contact* is determined based on dataset location. If the dataset resides on the local host, *server_contact* = false, otherwise *server_contact* = true.

4.4. Caching the host and port information.

Contacting the data mapper and server manager every time for host and port information may incur unacceptable overhead in certain situations. In such cases, the put and get functions may be implemented to cache the host and port information, so that after the initial call all remaining calls contact the server directly. If the call using the cached information fails to obtain service the server manager is contacted to restore the service and provide updated port information.

4.5. Implementation.

When the data host is local, it may make sense to read from and write to the disk directly. The implementation of the put and get functions hides these details from the client. In such cases, after the host has been resolved, no server manager or server communication is necessary, since the reads and writes work directly on the file system.

For debugging purposes it may be useful to force server access during a put or a get. The following rule is followed: if the host name resolves to ‘localhost’ the routines may optionally access the disk directly. If the host name resolves to anything else, such as the actual host name, the routines must make their requests via a server.

4.6. Puts and gets in a multiple host system.

Figure 6 shows an example of the data and message flow during puts and gets in a system with multiple hosts.

5. Servers.

Servers respond to client requests for putting data to the store and getting it from the store. There are two classes of server (a) non-translating servers which serve out the data in native mode and (b) translating servers which translate the data before serving it out. Server requests may require the computation of derived fields before serving the data out - for example, a vertical cross-section in gridded data.

5.1. Deciding which parameters to use.

5.1.1 No parameter file specified at start-up.

A server may be started without specifying a parameter file, in which case it initializes using default parameters.

If such a server is servicing a request and finds a parameter file in the data set directory, it reads in the parameter file to override the default parameters prior to servicing the request. The parameter file name is determined by concatenating the server executable name and ‘params’, separated by a period and preceded by an underscore.

5.1.2 Parameter file specified at start-up.

If the URL which initiates the start-up of a server includes a parameter file name, this file is read to set the state after the server starts. The parameter file name is determined by concatenating the server executable name and the parameter name, separated by a period and preceded by an underscore. This file is located in the root directory for the

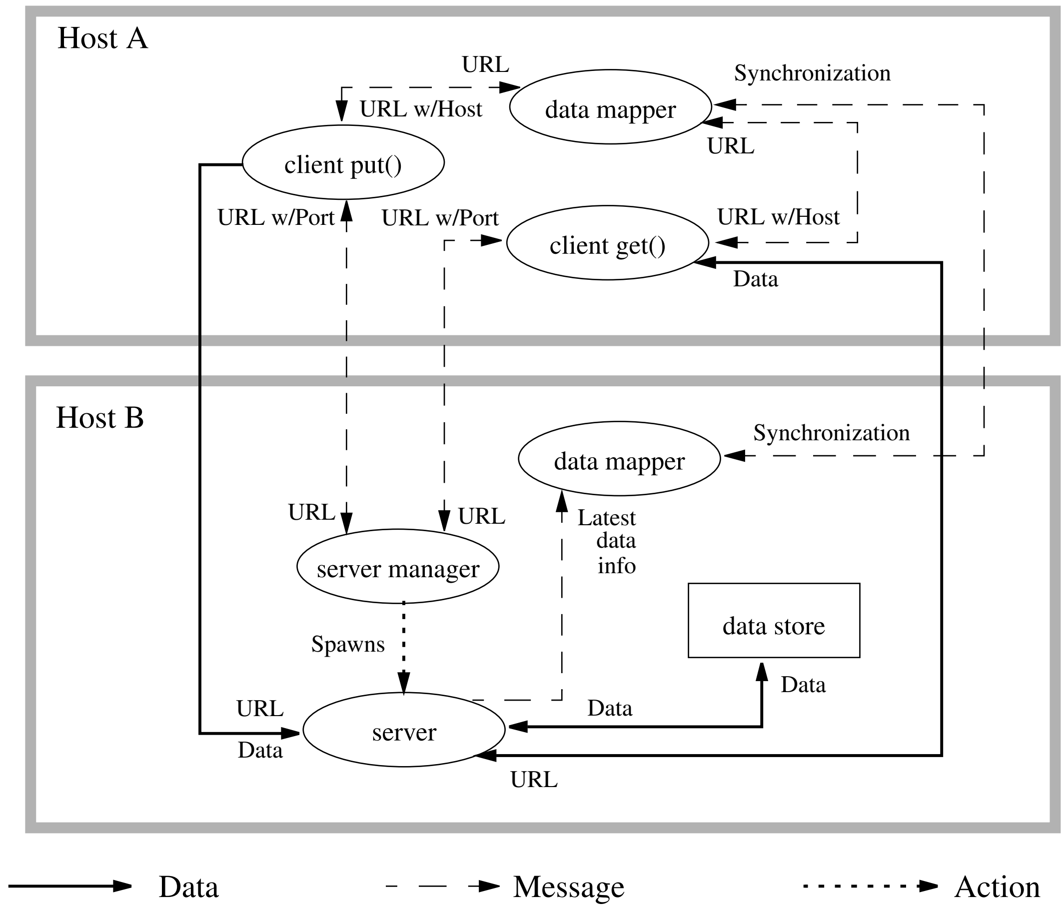

Figure 6. put() and get() messages and data flow across hosts in a multi-host system.

data type. In such a case, the server always runs with the start-up parameters and ignores any parameter files found in the data set directories.

5.1.3 Example of parameter files and their locations.

Consider the following directory structure:

mdv/_MdvServer.demo
mdv/sat/vis/_MdvServer.params

The ‘params’ parameter file would be used to override the parameters for an MdvServer which was started without a parameter file. The full path for this file would be *$RAP_DATA_DIR/mdv/sat/vis/_MdvServer.params*.

The ‘demo’ parameter file would be used at start-up for an MdvServer to satisfy a request for the following URL:

mdvp::demo//::radarCart

The full path for this file would be *$RAP_DATA_DIR/mdv/_MdvServer.demo*.

5.2. Handling a client request.

On receiving a client request a server decodes the URL to determine the directory for the required data set.

If the server was started without a param file, and a param file for that server exists in the data set directory, it reads in the param file to override the default parameters prior to servicing the request.

The server responds to the request by either writing to the specified data set directory, or reading from the directory, processing the data and passing it back to the client.

5.3. Requirements for interacting with the server manager.

In most cases the server is started and managed by the server manager. Therefore every server must provide a set of minimum capabilities.

When the server manager starts a server it must specify the port number for listening, and qmax, the maximum quiescent time, which is the maximum time, in seconds, which the server waits between connections before exiting.

Therefore the command line handling must allow for the following:

- **-port** to specify port number;
- **-qmax** to specify the maximum quiescent time in seconds.
- **-params** for specifying a parameter file name.

If the port is not specified, a default port must be used.

If -qmax is specified on the command line, the server must exit gracefully (with return code 0) if no client requests have been made over the past qmax seconds. The server manager reaps the child (using the wait3 function) and given a 0 return code it does not restart the server until the next request arrives for that service. This allows the manager to keep control of the number of running servers.

The server manager starts all servers using the -qmax command line argument. The value of qmax is specified when the server manager is started.

If -params is not specified, the server is initialized using the default parameters.

5.4. Server executable names.

Since the server manager must start the particular server required for a requested service, a logical naming convention is essential. Server names are chosen to match the data types. For example, MdvServer for MDV data, SpdbServer for SPDB data and TitanServer for TITAN data.

5.5. Server Protocols.

Data is served out using protocols which are specific to the data type. The protocol need only be understood by the server and its clients.

The server protocols names are derived from the data types, with ‘p’ appended. Therefore using the types above we have:

- mdvp: MDV data server protocol
- spdbp: SPDB data server protocol
- titanp: TITAN data server protocol

Data is served out using a specific protocol. For example, MDV data, which is gridded, is served out using the mdvp protocol. Other gridded data types, such as grib, may be transformed into MDV format and served out using the mdvp protocol. In the latter case, a translating server is used to transform the data format.

6. Server manager.

The server manager is responsible for starting servers as required, and providing server port information to clients.

6.1. Handling requests for service.

When the server manager receives a request for service, it does the following:

- decodes the first portion of the URL;
- decides which server is appropriate to handle the request;
- checks its server list - if the appropriate server is not running, it starts it on an available port and adds the server information to its server list. The desired port and qmax information is communicated to the server at start-up via the command line;
- fills out the URL with the port number and returns the resolved URL to the client.

6.2. Configuration.

The server manager reads a parameter file which contains a table relating server executable names to information in the URLs. It uses this table to determine which executable is relevant for each requested service.

6.3. Handling requests for the complete server list.

Some clients (such as the system monitor) may require the entire server list. If the server manager receives such a request, it places the entire list in a message and writes the message out to the client.

6.4. Maintaining the server list.

The servers are programmed to exit gracefully (return code 0) if they are quiescent (receive no connections) for a certain period (qmax seconds). The server manager, as the parent process, must reap the children if they die. If the return code is 0, the server manager simply removes the server from the list. If the error code is not 0, the manager logs that information. If a particular server dies continually with errors the server manager may elect not to restart it and deny client access to that service by returning an error code to any client which requests the service.

7. Data mapper.

The data mapper receives, stores and serves out information on data sets within a shared data region. It listens for clients on a well-known port.

7.1. Data set list.

The data mapper maintains a list of the following information for each data set:

`host`: host on which the data resides.

`data_type`: major data type, e.g. mdv.

`dir`: subdirectory of data store, relative to `$RAP_DATA_DIR`.

`latest_time`: the time of the latest data put to the store. This is not necessarily the most recent data in the store, because in playback mode the latest data may precede data already on disk.

`start_time`: start time of data in the store.

`end_time`: end time of data in the store.

7.2. Data mapper clients.

The data mapper handles messages from a number of different clients.

7.2.1 Registration of data set information.

- in real-time mode, the put functions contact the data mapper to convey the time of latest data put to the store;
- the scout determines the start and end times for each data set, and conveys this information to the data mapper.

7.2.2 Requests for data set information.

- if a get function receives a URL which does not include the host name, it passes the URL to the data mapper for resolution of the host. The mapper fills out the URL with the host name and returns it to the client. If the data set is distributed across hosts, an array of resolved URLs is returned in priority order - the mapper puts the URL most likely to satisfy the request at the top of the list, and so on. The clients can then try the URLs in order.

- system monitor processes contact the data mapper to obtain the entire active data set list. The active list is compared to an expected list and the comparison results are displayed to the user.

7.3. Synchronizing data mappers within a shared data region.

The data set lists for all data mappers within an SDR are identical. This is achieved in the following way. Each data mapper routinely scans a file, in $RAP_DATA_DIR, which lists the hosts in the SDR. When a registration message is received, this indicates that the data store has changed in some way. The data mapper forwards this message to all the other mappers in the SDR. It also receives forwarded messages from the other mappers. This keeps the data set lists on all hosts synchronized.

7.4. Implementation.

The data mapper stores the data set list in a file in $RAP_DATA_DIR. This allows certain functions, such as the client get(), to access the data set list directly via a file read instead of via a socket call. This improves efficiency and simplicity.

8. The janitor.

The janitor is responsible for maintaining free space on the disks. It achieves this by performing the following functions:

- data compression;
- data scrubbing.

The janitor routinely traverses the data tree, compressing files which have not been recently used and removing old files as required. The actions of the janitor depend upon:

- the time since a file was last accessed - a file which has not been used for a certain period may be compressed;
- the time since a file was last modified - a file which is older than a certain age may be removed;
- an event list - event lists protect the data for given events, so that all other data is scrubbed before data from the events. An event is a time period of interest for analysis, so the data should be saved if possible. Event lists are created and installed manually by users;
- the free space on the disk - as a disk fills, the file scrubbing age may optionally be reduced in an effort to keep a certain minimum free space on the disk. The maximum allowable age before deletion is calculated by linear interpolation based on the percentage of disk usage. Beyond that the disk is allowed to fill, since there

is no point in removing very recent files which are likely to be required by the system.

The janitor ignores all files with names beginning with an underscore. Therefore, the names of all special files such as parameter files and access control files should begin with '_'.

These behavior of the janitor is configurable via a master '*_janitor*' parameter file, to be found at the top of the data tree, and on secondary *_janitor* files at other points in the tree. The janitor moves recursively through the tree. When it finds a *_janitor* file at a given level this is used to override the current parameters from that level downwards. When it moves up to a previous level the overrides are lost and the previous parameter settings apply.

9. The scout.

The scout has the responsibility of supplying the data mapper with details of the data sets in the store. The data store may be modified in the following ways:

- a client writes to disk via a put function, which may in turn use a server. In either case the data mapper is informed directly by the put function or server, therefore the actions of the scout are secondary;
- data is added to the disk in some other way, for example when a user copies files from some other location;
- the janitor removes data through scrubbing;
- a user deletes files.

The scout moves through the data tree, locating the data sets and computing the start and end time limits for the data sets. It then communicates this information to the data mapper.

10. System monitor.

The system monitor provides a mechanism for summarizing the status of the system and presenting that summary to the user in a clear manner.

The monitor reads a static file which contains information about system configuration and an expected data set list. It also periodically contacts the data mapper to obtain the current data set list. It compares this list with the expected list and identifies any discrepancies such as late or missing data.

The data set list and discrepancies are displayed to the user.

11. Providing redundancy.

For some projects, high reliability is required. In such cases some or all of the system may be duplicated to provide redundancy. A distributed system is an excellent way to provide redundancy.

11.1. Redundant data mappers.

The data mappers in a standard distributed system are already redundant since their data lists are copies of each other.

11.2. Redundant data stream.

Running redundant data streams requires that the following needs be handled:

- splitting a stream into two;
- joining two streams;
- allowing a client to pick the best store for a given data set.

Figure 7 shows an example of a system in which part of the data flow is duplicated in redundant streams.

Splitting a stream is simple - a put() function puts the data to two locations as depicted at point A in the figure.

Joining two streams requires an arbitration process. Consider the two puts at D and E in the figure. Each put() acts alone, and the server must decide whether to store the data from the second put() over the first or to ignore the second put() request.

A client must decide where to get the data. It contacts the data mapper and receives an array of URLs, one for each data store location, B and C. It tries the first URL, and if that fails it tries the second one.

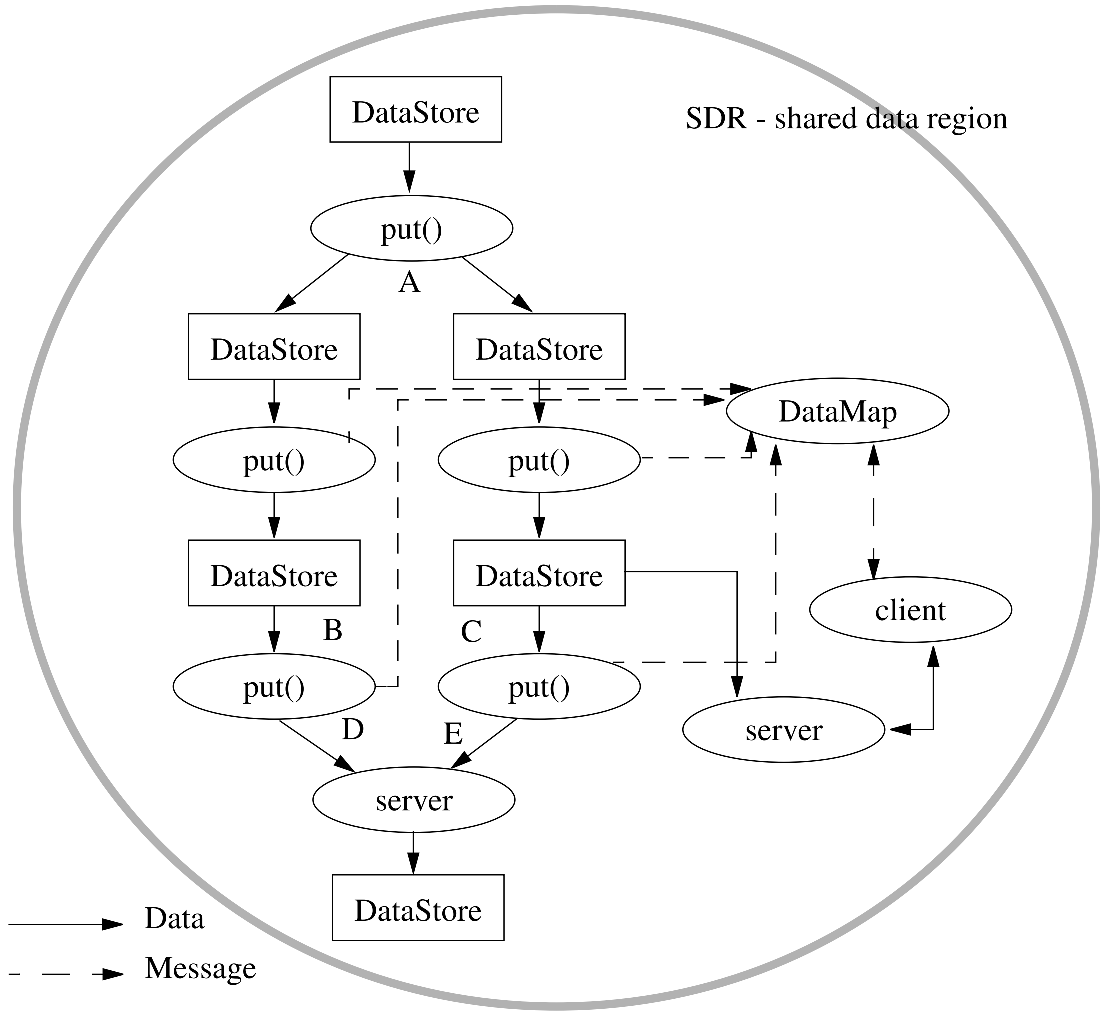

Figure 7. Example of redundant data streams.