

NotificationServer Implementation

[Gerry Murray](#)

April 28, 1998

Note: This document describes the state of the notification server software midway through Build 4.2 development. Several changes were made in the details of the server's operation. An update of this document is pending.

The notification server's primary purpose is to receive and gather messages from decoder processes and other data suppliers signifying that new data has arrived and is now ready to display or process. It then determines which depictables are interested in those data notifications, and when they want to be notified. Finally, the server matches the notification times with corresponding inventory times, and sends those inventory times to processes that are currently using those depictables.

Now that we have an overview of what the notification server does, here is an overview of the implementation, categorized by functionality. This document only describes the implementation of version 4.2 to be released in August, 1998. Previous versions of the server have the same functionality but the implementation may differ substantially. This document discusses the implementation of the following functionality:

- [Client Registration for Depictable Notifications](#)
- [Notification of the Latest Inventory Time Upon Depictable Registration](#)
- [Receiving Data Notifications](#)
- [Converting Data Notifications into Depictable Notifications](#)
- [Sending Depictable Notifications](#)
- [Fault Tolerant Inventory Retrieval and Caching](#)
- [Restoring State after a Restart](#)
- [Statistical Reporting](#)
- [Notification Trace Logging](#)

Client Registration for Depictable Notifications

When a client like the *IGC_Process* loads a product and decides that the product needs to auto-update, the IGC will send an IPC message to the *notificationServer* process. Our process uses a "named target" which means that its IPC address is easily found by other processes. Most of the real work is done in the *Notification Server* singleton (one per process) object and its subsidiary objects. This singleton object is also a receiver (subclass of *DataMgmtReceiver*); thus all the *NotificationServer::receive** methods are for receiving and interpreting incoming IPC messages.

NotificationServer::receiveDepictRegistration() handles a request from a client process to register for one or more depictables. *NotificationServer::receiveDepictCancellation()* performs the inverse service. Both methods delegate the real work to the *NotificationServer::_depictRegistrations* object. This singleton manages all the depictable registrations. It provides two important mappings:

- Depictable to IPC addresses of interested processes.
- Data access key to associated depictables.

It also contains some useful attributes for the registered depictables such as how often the depictable wants to be updated, or whether notification times for this depictable should match exactly with the inventory times. Registration and cancellation requests are ultimately handled by this singleton so that the internal data structures can be updated.

Notification of the Latest Inventory Time Upon Depictable Registration

Since the UI process has enough to do at initialization, it doesn't want to be obtaining inventories just to get the latest time for display on a product button. That is why a registering process can ask the notification server to send back the latest time for the depictable it is registering for.

When the notification server is asked to do this, it is usually registering many depictables with a single request from the client process. The server maintains an inventory cache for various depictables. For more about the inventory cache, [check out the section describing inventory retrieval and caching](#). *NotificationServer::receiveDepictRegistration()* builds a list of all registered depictables that have a valid, cached inventory and uses the list to send a large IPC message back to the client containing those depictables and their latest times. Of course, if the cache is empty or small, very few times are sent back to the client.

Sometimes, the client will request a latest time for an inventory that has not been fetched yet. In this case, we defer this request since fetching the inventory can be time consuming and we don't want to spend excessive time while handling an incoming IPC message. This request is deferred by adding the depictable key and client's IPC address to the data structure:

NotificationServer::_unsentRegisterNotifies(). The server's flow of control calls *NotificationServer::sendRegistrationNotification()* when there isn't more important tasks to do. This method finds the latest time of one of the depictables in the data structure and sends that time to all interested client processes. If the send failed, all of the client's depictable registrations are automatically cancelled.

Receiving Data Notifications

Messages signifying new data arrive asynchronously via the IPC mechanism. They are handled by *NotificationServer::receiveDataNotification()* and consist of a data access key and a *DataTime* object. The time usually corresponds to a time on the inventory but it doesn't have to! The method asks the *DepictRegistrations* singleton if there are any registered depictables that are associated with the data key. If not, the data message is ignored. If so, the key and time is added to a dictionary of *DataTime* sets (*NotificationServer::_newDataList*). Each time set is sorted so that the latest time is the last set element. This is useful for validating a cached inventory in some cases which will be discussed [in the section covering the validation of cached inventories](#) .

Converting Data Notifications into Depictable Notifications

The notification server buffers all data notifications that arrive during a fixed interval of time (currently hard-coded to 20 seconds). We do this because data notifications come in very frequently and are often redundant. Inventories take much longer to get; if we sent out a depict notification every time a data notification came in, we would not be very responsive to incoming IPC, possibly causing socket buffers to overflow.

At the end of a data collection interval, we convert data notifications into depictable notifications by calling *NotificationServer::generateDepictNotifications()*. Its job is to loop through every data notification that has been placed in the new data list by *NotificationServer::receiveDataNotification()*. Each data notification is converted into one or more depictable notifications depending on how many depictables are associated with that data key. Depictable notifications consist of a depict key and a set of *DataTime* objects. It computes the depictable notification time set by merging all the associated data notification sets. Most depictable notifications are sent out by the next time interval, but some are deferred and can be around for several intervals. It is possible for a data notification to add a new time to an existing depictable notification. Once we loop through all the data notifications, we clear the data structure, *NotificationServer::_newDataList* since all the data notifications have been converted and are no longer needed. At this point, we also increment our statistical counters.

Deferring Certain Depictable Notifications

Depictable notifications are either placed at the back of the pending notification queue (*NotificationServer::_pendingDepictNotifies*) or placed in the deferred notification table (*NotificationServer::_deferredDepictNotifies*), depending on how frequently this depictable is to update. The frequency is part of the depictable's meta-data that is stored in the *DepictRegistrations* singleton object. This object gets this information from a *DM_DepictableInfo* object at depictable registration. Notifications are deferred with frequencies greater than 20 seconds. If it is a deferred notification, the expiration time (current time + frequency) and the depictable key is placed in a dictionary (*NotificationServer::_delayExpirations*) that maps expiration times to depict key with the times sorted from earliest to latest. Purpose of this data structure is so *NotificationServer::sendNotification()* will know which deferred notification to process next.

Sending Depictable Notifications

As mentioned earlier, our first priority is to be responsive to incoming IPC. The next important priority is to send out the depictable notifications that we have generated. Notifications are sent out one at a time while there are no incoming IPC messages to receive. This is repeated until there are no notifications to send. At this point, more depictable notifications can be generated (if at least 20 seconds have elapsed) or some registration notifications can be sent.

A single depictable notification is sent to all interested processes using the following algorithm:

1. Select a depictable notification to send. It can be the earliest one on the deferred notification list whose time is earlier than the current time. Or it can be the first notification on the pending notification queue. In either case, the notification is removed from its corresponding data structure. A depictable notification consists of a depictable key and a set of `DataTime` objects representing the notification times.
2. Get an inventory from the cache. The inventory cache decides whether to calculate a new inventory using the set of notification times or to just use the cached inventory.
3. Use `::inInventory()` to determine which inventory times match the notification times. An exact match or a closest match strategy can be used depending on the depictable. Like the frequency attribute, this attribute is obtained when the depictable is registered by using a ***DM_DepictableInfo*** object. An exact match algorithm returns the index of the inventory time that matches exactly with the specified time; i.e. either analysis times or reference times are equal and both forecast times are equal. A closest match algorithm returns the index of the earliest inventory time that is later than the specified time. If the specified time is later than the latest inventory time, then the index for the latest time is returned.
4. Construct an IPC message for each matched inventory time. The message includes the key, the matched inventory time, and the entire inventory.
5. Send this message to each process associated with the depictable. If the send failed, cancel all of the client's depictable registrations.

Fault Tolerant Inventory Retrieval and Caching

The *NotificationServer* singleton uses a helper singleton object, *NotificationServer::_inventoryCache* to do inventory retrieval and caching. The cache singleton works with another helper singleton, *NotificationServer::_ourSigHandler* to provide resilience when computing inventories.

Retrieving and Validating a Cached Inventory

The inventory cache is simply a dictionary of pointers to inventories (list of *DataTime* objects) indexed by depictable key. All inventory requests are made through the method *InventoryCache::retrieve()* with the caller specifying the depictable key and a set of times used to validate the cached inventory. An empty set of times may be used indicating that the cached inventory should be returned without validation (if there is one to return). There are two ways to validate an inventory, so the caller also has to specify which approach to use.

Retrieving an inventory involves checking to see if the inventory is available on the cache. If so, it is validated using one of the two methods described below. If valid, then it is returned. If the inventory is invalid or not in the cache, we have to obtain the inventory in a potentially arduous manner described in the next section. But before doing so, we free the memory allocated for the invalid cached inventory.

The first method of cache validation ensures that every validation time exactly matches a time on the inventory. If not, the cached inventory is considered invalid.

The second method is more forgiving than the first. If the latest time on the validation set is earlier than the latest time on the inventory, the inventory is valid. Otherwise, it is not.

Computing an Inventory

To compute an inventory, we use the static method, *DepictableInventory::allTimes()* which may execute a heterogeneous assortment of algorithms depending on the depictable and the data accessor supporting that depictable. Some are extremely robust and efficient while others are not. For that reason, we decided to allow the notification server to continue even if a run-time exception or fault is encountered while obtaining the inventory.

The fault tolerance requirement is implemented with the Unix jump mechanism. The run-time environment is saved with the Unix routine, *setjmp()* just before *DepictableInventory::allTimes()* is called. If an exception is thrown, the signal catcher will pass it to the singleton object, *NotificationServer::_ourSigHandler* which will call *longjmp()* and return us back to the place where *setjmp()* was called. At that point, we know that we can not get a valid inventory, so we return immediately from *InventoryCache::retrieve()* passing the caller a null pointer.

If *allTimes()* succeeds, we insert the pointer to the inventory into the cache table (*InventoryCache::_cache*) and return the pointer to the caller of *InventoryCache::retrieve()*. The memory for the inventory is allocated by *allTimes()* but the responsibility for deleting the memory is the sole responsibility of the cache singleton object. Other NotificationServer methods can access the inventory for read only, but should never destroy the inventory.

Restoring State after a Restart

An important design requirement is that restarting of the notification server process should be as transparent as possible to the users of the display workstations. Of course, a few notifications may be dropped on the floor while the server is down. However once the server is up and running again, notifications should resume without having to restart the workstations.

Saving State

As stated earlier, *NotificationServer::_depictRegistrations* object handles the mapping between registered depictables, processes, and data keys. Another important responsibility is the saving and restoration of state. In order to do that, this object maintains a set of IPC_Target objects (*DepictRegistrations::_currentClients*). Whenever a client registers or cancels a depictable registration, we add or remove an IPC target from the set only if no other depictables are associated with that client. Whenever, the set of addresses is modified, the method: *DepictRegistrations::saveClientState()* is invoked. This method simply writes an ASCII representation of each IPC address in the set to the file:

`$FXA_DATA/workFiles/notificationServerClientListState.txt`

Recovering the Saved State

Client state is restored by *DepictRegistrations::restoreClientState()* which is called at process initialization. This method extracts each IPC address from the client state file, and sends a message to each client process requesting the client to register again for all the depictables that it is interested in. If we can not send this message to a process, we don't worry about it. It just means that the client process has gone down while we were down. The *DepictRegistrations::_currentClients* data structure is modified to contain the addresses of the processes that we were able to send to. In addition, the client state file is written in order to reflect the current state of the client set.

Statistical Reporting

Periodically, (currently hardcoded to be once an hour), a statistical report is logged via the logStream category: DIAG by the method: *NotificationServer::showServerStats()*. Statistics include the following:

- Registered depictables and their associated clients.
- Data notifications received and generated into depict notifications since the last report.
- Depictable notifications that were successfully and unsuccessfully sent since the last report.
- Inventory cache including access times and size information.
- IPC usage. See `src/threadIPC/SocketConnection.C` for details.

A statistic report can be also produced by an outside stimulus, asynchronously, by using the executable, *testNotificationServer* with an argument of 30003.

Notification Trace Logging

As you might imagine, if we logged all the notifications received and sent continuously, the size of the log files would quickly become unmanageable. However, there are times while debugging that it is useful to change to a mode of logging all activity without having to restart the server. That is why there is a mechanism to trigger and un-trigger a verbose logging state by an outside stimulus.

A user can provide that stimulus by running the *testNotificationServer* executable on any host on the same network as the server with an argument of 30004. This will send an IPC message to the server which will toggle the value of the boolean flag: *NotificationServer::_notificationTrace*. With *_notificationTrace* set to true, the following situations are logged with the event category instead of the verbose category:

- Receiving data notifications
- Constructing depict notifications
- Sending depictable and registration notifications
- Registering and cancelling Depictable registrations