Meta Change Queue: Tracking Changes to People, Places and Things

Jon Finke – Rensselaer Polytechnic Institute

ABSTRACT

Managing information flow between different parts of the enterprise information infrastructure can be a daunting task. We have grown too large to send the complete lists around anymore, instead we need to send just the changes of interest to the systems that want them. In addition, we wanted to eliminate ''sneaker net'' and have the systems communicate directly without human intervention. Some of our applications required real time updates, and for all cases, we needed to respect the ''business rules'' of the destination systems when entering information. This paper describes a general method for propagating changes of information while respecting the needs of the target systems.

Introduction

At LISA 2002, I presented a paper *Embracing and Extending Windows 2000* [4] that described how we kept our Windows 2000 environment, as well as our LDAP directory services synchronised with our Unix account space. These feeds quickly grew to carry more than just Unix account information to include directory and other status information. Well, we were a victim of our own success. Other systems needed access to the same or similar change feeds, and other data streams were becoming available, and a more general architecture was needed. In addition, we found that we had to interface with vendor supplied systems and it became important to provide a clear demarcation between our systems and the vendor's systems and provide a clear place to implement their business rules with our data.

At LISA 96 in Chicago, I gave an invited talk *Manage People, not Userids* that demonstrated the importance of managing the more general information about people, and from that, managing their computer accounts. In addition, in a paper at the same conference, (*White Pages as a Problem in Systems Administration* [3]), I again showed how tools for systems administration could benefit other areas and that many areas for code and tool re-use exist. As our friends in the JAVA community (and other object oriented languages) are fond of telling us, solve the problem once, and re-use the solution to solve other problems. Thus, we wanted a general mechanism to move different types of changes to different systems.

At our site, many of our systems[1] are vendor supplied packages running on an Oracle or other relational database. In addition, we were also feeding information to non relational database systems such as our LDAP directory servers and the Windows 2000 domain controllers. To further complicate matters, we have many different data elements available, and not all systems wanted all data elements, we needed ways to pick and choose which data elements went to which system. We also needed to be able to accommodate different operating schedules and data latency requirements. Some data elements change very slowly (such as adding a new building) where a daily update feed is more than adequate, while other data elements need to move much faster (such as a password change, or email forwarding.) We wanted to retain the low processing costs we achieved in earlier implementations, while making it easier to add new ''listeners'' to a feed. Lastly, although we wanted changes to propagate quickly, we needed to avoid blocking an operation on one system because a downstream system was not reachable.

[1] Student Records, Human Resources, ID Card, Dining Services, Space Management, Telephone Billing, Help Desk, etc.

Interfaces and Business Rules

The first aspect of this project, is the interface model we use to actually get the changes into the destination system. While many applications have procedures to import a CSV file, these require manual activity and our objective is to fully automate the process. Some applications and systems provide an API that we can call to insert and update records; this is our preferred method. But other systems don't provide that and for at least database based systems, we need to muck about directly in the vendor database tables. We wanted a clear demarkation between our systems, and the interface code that needs to understand how the target system works. For the systems without an API, our approach is to insert the changed records into a import table and have that trigger the appropriate processing. We have used this model as well as the API model successfully.

Assuming that we have some sort of interface, we still need to face the classic system admin issue of pushing in changes from the central server, versus pulling in changes from the client. The answer here is ''it depends.'' In general, I have taken a very pragmatic

approach. For destination systems that do not have "aggressive administration,"[2] I prefer the push model from the central server. This allows me to monitor the connections and updates and become aware of problems (and hopefully resolve them) before the end users. This also allows me to adjust the schedule and timing of updates as needed. For systems with aggressive administration, we can negotiate the "best" approach (more efficient, least work, etc.).

Procedural API

At the heart of the Meta Change Queue package is the Get_Changes routine (Figure 1) which provides all of the changes for the specified listener in order. This is called with the processor (queue) name, and an optional table name within that queue. This will return a record with a number of fields of interest (Table 1). When the record has been processed, the Ack_Change routine is called. This cycle is repeated until the Change_Type field in the record is null. This indicates that there are no more changes that need to be processed.

```
Function Get_Changes(
      Proc_Name in varchar2,
      tname in varchar2)
return rec;

procedure Ack_Change(R in Rec);
```

Figure 1: Get_Changes definition.

When an application is processing a change, it examines the change record, and based on the Tname and subtype (and other fields), determines what record had changed and gets the current value of that record from the database. This is a very important issue to understand, we do not record what the change was, only that something had changed. We need to be able to move the final state for a record, without having to step through intermediate steps. If I change my phone number twice, the only thing that matters is the final number. Other aspects of our systems may maintain history and change logs, but not this one. Here we only indicate that something changed. The application must be able to apply the same change twice without harm, i.e., "set quota to 100" is ok to repeat, "increase quota by 50" is not.

There is another set of routines that given a change record, will return the desired information (directory, status, etc.) to applications that can then update the target system. This model has worked well with our interfaces to LDAP and Active Directory where we have written a program in Java or C#, that gets the queued changes and updates the target. These applications apply all the changes in the queue, acknowledging them as they go. Once it reaches the end of the queue, it sleeps for a short time and looks for changes again. These programs will retry if they loose the network connection and will eventually catch up once they can reconnect. This automatic restart has proven very handy and reliable.

The Get_Changes interface also provides a handy hook for our process monitoring system [5]. The applications that are polling via the Get_Changes routine often just sleep for a short time; maybe a second. Unlike the calls to Get_Changes which puts a very small load on the database, calls to the Mark_Process routine results in a write (or update) to the database, and frequent calls will impact performance and transaction logs. So we typically wrap the call to Mark_Process in code that skips the actual call until at least five minutes has elapsed since the last call. This will still give us good notification when one of these processes dies. We usually catch one that has died every three or four months.

Import Table

Our second interface method is by using an import table to receive records. When a record in inserted into the import table, a database trigger[3] fires which will then process whatever business rules that

[2] An administrative team who is constantly monitoring the system and is able and willing to set up cron jobs or the equivalent.

[3] A database trigger is a stored procedure in the database that will be executed whenever there is an insert, update or delete on a row in a database table [1].

Field	Type	Description
Tname	varchar2(32)	The name of the table that had the change.
Change_Type	varchar2(8)	One of "Insert," "Delete," or "Update."
rrowid	rowid	Oracle row identifier of base table record.
proc_name	varchar2(32)	The processor (or queue) name.
subtype	varchar2(32)	More detailed information about what specifically changed about the target object.
person_id	number	The internal person identifier if the object is defined in the "people" table.
Pkey_String	varchar2(32)	The primary key (identifier) of the object (if not a person) as a character string.
pkey_number	number	The primary key of the object when that is a numeric value (not a character string).
aux_string	varchar2(255)	An optional extra character field to identify the change. Often used for membership changes where two keys are needed.
entry_date	date	The time and date this change was made.

Table 1: Change record definition.

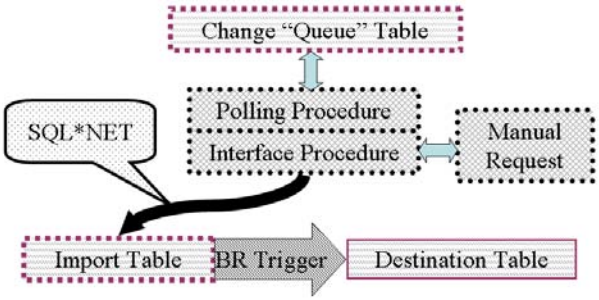

Figure 2: Interface and business rules.

are required. This appears as the bottom row of elements in Figure 2. We have used this successfully with several different vendor applications.[4] In cooperation with the vendor engineers, we define an import table, and then the vendor engineer writes a database trigger that processes each insertion as it happens and makes the appropriate changes in their own tables. This allows us to feed in the changes in a controlled manner and isolates our code from vendor changes. The vendor does need to update their triggers when they make a change. We had originally intended to queue the records in the import table, and then the vendor would have a process that looks for pending records (much like how we did the Meta Change Queue project), but we found it easier to just write the trigger and avoid writing the polling application.

One example of this is seen in Figure 3 which is a database trigger written by a vendor engineer. In this case, for each new entry in the Simon_Person_Import table, it first checks to see if the entry has already been made in the vendor table SA_PERSON, and if not, it inserts the person. If the person is already in the table, it checks to see if the person has a status.[5] If they don't have a status, see if they did, and if so, change it to "Former-" whatever and then update the person's record. The vendor application did not a have a field for the "Status" of a person, and although we could

[4] BEST – ID Card and Access Control, FAMIS – Physical plant trouble ticketing, INSITE – Space Management

[5] Maintaining "status" values for every person is a topic for another paper.

```
CREATE OR REPLACE TRIGGER T_SIMON_PERSON_IMPORT
BEFORE INSERT ON SIMON_PERSON_IMPORT FOR EACH ROW
declare
    Cursor Get_rec (pn number) is
    Select person_number,person_last_name,person_first_name,
           person_text1, person_location, person_memo, rowid
      from SA_Person where person_number = pn;
    R           Get_Rec%RowType;
    new_status  varchar2(48);
begin
    Open Get_Rec(:new.spriden_id);
    Fetch Get_Rec into R;
    if Get_Rec%NotFound    -- No existing record, insert a new one
    then
        new_status := nvl(:new.status,'No Status');
        INSERT INTO SA_PERSON
            (PERSON_ID, ENTERED_DATE, ENTERED_BY, PERSON_NUMBER, PERSON_LAST_NAME,
             PERSON_FIRST_NAME, PERSON_TEXT1, PERSON_LOCATION, PERSON_MEMO)
        VALUES
            (PERSON_ID_SEQ.NEXTVAL, SYSDATE, USER, :new.spriden_id, :new.lastname,
             :new.firstname, orgn, substr(new_status,1,24), :new.title);
    else
        if :new.status is null  -- If no current status, save what they were
       then
            if substr(R.person_location,1,7) = 'Former-'
            then new_status := r.person_location;
            else new_status := 'Former-' || R.Person_Location;
            end if;
        else
           new_status := :new.status;
        end if;
        Update SA_Person
             set Updated_Date = sysdate, Updated_By = user,
                 Person_Last_Name = :new.lastname, Person_First_Name = :new.firstname,
                 Person_Location = substr(new_status,1,24),
                 person_text1 = orgn, person_memo = :new.title
           where rowid = r.rowid;
    end if;
end;
```

Figure 3: Insite Trigger.

have added one to their table (like we did with the Person_Number, we did not want to change all of the display screens, so we took over the Person_Location field, and store and display the status there.

We don't actually care about the contents of the Simon_Person_Import table. Once the trigger fires and completes, all of the work is done. We periodically flush the import table. If there is a problem with the trigger, perhaps some integrity constraint (unique usernames, etc.) is violated, the trigger throws an exception and the insert fails. This exception propagates back to the system attempting the insert and appropriate error handling can take place there.

This approach has the additional advantage of allowing us real time updates for applications that needed it. For example, we have a secure web page that is used by our Human Resources department to mark when a new employee has signed their I9 form (and is now allowed to start work). This web form updates the person's status, and immediately pushes that change to the ID card system. By the time the new employee has made it to the ID desk, they have already been loaded in and can have their ID card photo taken right away. This has made both the HR staff and the ID desk staff happy (HR is happy because they can now control when someone is issued a staff ID card, and the ID Desk staff is happy because they don't need to call HR to verify each new hire.)

Not all changes need to happen in real time. Many changes happen as the result of other automated processes and batch jobs. We have a simple PL/SQL program that uses the Get_Changes routine to find out what has changed for a given queue, and then loads the appropriate records into the import table. If the target system is down, the changes will wait in the queue until the next run. Since we are using the process monitor to ensure that this happens, we know when the scheduled jobs does not complete successfully. In the new employee case, we have already loaded the employee via the HR web page, but the repeat load in the next batch run doesn't hurt anything.

We can combine the use of the queuing support described in the previous section, with the insert trigger based code, to come up with a catch up routine like the one in Figure 4. This simply looks for changes for the 'Insite' queue, and passes them to the Insite system via the Push_Person routine we described earlier. Once we get to the end of the list, we record the fact we finished and terminate. This process is called once a day by a cron job.

Manual Entries

When we bring a new system on line, it is generally empty of our data. Rather than loading it via CSV files or other bulk import tools, we use the Meta Change Queue interface to load them up. In the cases where there is a program calling the Get_Changes routine directly, we simply manually insert records in the queue for that service, and watch what happens. If we like what we see, we write a simple script to load all objects of interest into the queue. From that point on, things run automatically, and the interface has been well tested, as the entire system load has been processed via the new interface. This also makes it easy to reload if we decided to flush and start over.

In Figure 5, we have an example of a PL/SQL script that will select all transfer students from the Fall of 2002, and "refresh" their entries in any listener that

```
procedure Push_Queue(stopcount in number)
is
    R       Meta_Change_Access.Rec;
begin
    loop
        R := Meta_Change_Access.Get_Changes('Insite', Null);
        exit when R.Tname is null;
        Push_Person(R.Person_Id);
        Meta_Change_Access.Ack_Change(R);
    end loop;
    Process_Monitor_Record.Mark_Proc('Insite-Push_People');
end Push_Queue;
```

Figure 4: Push_Queue procedure.

```
declare
    Cursor Lrec is Select Username,Source, owner, unixuid, rowid
      from logins where admit_cohort='TR200209';
begin
    for L in lrec
    loop
        Meta_Change_Rtn.Log_Update('LOGINS',l.rowid, person_id => l.owner,
             pkey_string => l.username, pkey_number => l.unixuid);
    end loop;
end;
```

Figure 5: Manual refresh via queue.

is interested in changes to the LOGINS table.[6] You will note that several of the parameters specified in the call to LOG_UPDATE correspond with fields in the change record (Table 1).

In the cases where we use the import (trigger) table, we generally have written a routine like Insite_Interface.Push_Person (Figure 6) that will look up the appropriate information and do the appropriate insert (via SQL*NET). This routine can be called by hand for testing, and later on via scripts to bulk load the entire population. In Figure 7, we have an example of PL/SQL script that will load all current employees and faculty into the INSITE (space management) system via the Insite_Interface.Push_Person routine. This routine calls routines in the Meta_Change_Data package to get the data elements that are needed, and then inserts that into the import table Simon_Person_Import on the Insite machine using sql*net.

Tables and Listeners

The second aspect of the project, is how we detect changes, queue them, and finally deliver those changes in a timely manner.

Defining Tables

The original concept was to track changes in a particular database table, but in the actual implementation, this proved to be limiting. Instead of looking at the details of the source systems tables, we looked at the data requirements of the destination system. For example, one system might just want general information on a person such as name and status, while another system would want that as well as directory information. Since the transfer model was to give them a complete record of all desired information about a person, a facility to pick and choose what information about a person, was desired. Instead, we defined the table to be the source of the primary key, and added a sub type to indicate what about the base object changed. For example, a telephone number change would be marked as the PEOPLE table and the Telephone sub type. We currently have 16 table and sub type combinations defined (Table 2).

To detect these changes, we set up a database trigger (Figure 8) which records whenever a telephone number is changed. There are similar triggers to handle new telephone numbers (inserts) and deleted telephone numbers. Since this was done with a database trigger, we did not have to change any of the applications that had been previously developed to make changes. It also ensures that we don't miss any changes.

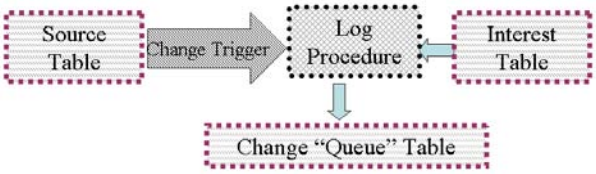

Figure 9: Detecting changes with triggers.

While database triggers can be very handy for integrating existing applications, they can sometimes get complicated. We often have changes to a table that are ''housekeeping'' in nature. Something in the table changed, but that change is not of interest to any downstream systems. You can with a trigger be more selective about what columns you look for changes in, but that makes the trigger more complex. Triggers are also challenging from a maintenance prospective, as they are sort of split conceptually between the table

[6] The query has been edited for space, but the concept is still valid.

```
Meta_Change_Data.Person(Person_Id, Lname, Fname, Mname,
    PFN, Rin, Iso, DOB, Gender, Ssn, Pidm);
Meta_Change_Data.Person_Department(Person_Id, Department, Division,
    Portfolio, Insite_Name, Orgn_Code;
Meta_Change_Data.Person_Status(Person_Id, Category, ID_Card,);
Meta_Change_Data.Person_Directory(Person_Id, Title, Camp_Add,
    Camp_Phone, Camp_Fax, Mailstop);

Insert into OPS$INSITESYS.SIMON_PERSON_IMPORT@insite
    (Spriden_Id, Lastname, Firstname, Orgn_Code, Status, Title)
Values (Rin, upper(substr(lname,1,24)),upper(substr(nvl(pfn,fname),1,16)),
    Orgn_Code, upper(ID_Card), upper(Title));
```

Figure 6: Insite_Interface.Push_Person .

```
declare
   Cursor Emp_List is
   Select person_id,spriden_id,lastname
     from people
     where id_card_status in ('Employee','Faculty');
begin
    for R in Emp_List loop
        Insite_Interface.Push_Person(R.person_Id);
    end loop;
end;
```

Figure 7: Direct refresh (trigger table).

definition (DDL) which is usually set at the start of the project and the interface code (PL/SQL Packages). New projects allow for closer integration of the change queue requirements with the interface code.

We recently installed a unified messaging system.[7] Although this system was supposed to use our existing Exchange server (the one discussed in *Embracing and Extending Windows 2000* [4]), our initial deployment required a second Exchange server, and in fact, its own Windows 2000 domain. An obvious step was to set up another listener in parallel to the one used for our primary Windows 2000 domain. However, along with the LOGINS information needed, we also needed voice mail specific information.

Since this was a new project, we were able to design the system so that all access to the ‘‘voice mail’’ tables was via single interface package. This allowed us to call the Meta_Change_Rtn.Log_XXX routines directly as needed. This gave us much greater flexibility in what we send to the Unity system for processing. For example, we have two ‘‘owners’’ for many objects. We have the ‘‘Unity Owner’’ which controls some access on the Unity system itself, and ‘‘System owner,’’ which controls administrative access on the central database. For operational reasons, these often different entities. A voice mail tree will be administratively owned by a department, while on the Unity system, it will be ‘‘owned’’ by a group of administrators. We often need to change the administrative owner, but there is no need to send any changes to Unity. By having the single interface, this can be handled properly in the interface package.

In order to manage what tables are available to the listeners, we defined another database table, Meta_Change_Tables (Table 3) to hold that information. The primary purpose of this table is to document what is available. Most of this information is set when the table is defined, but one aspect is collected automatically. The first time a change record is logged for a specific Tname SubType pair, the PL/SQL call stack is saved to this table. This is a traceback of what procedures and packages called the logging routine. This can be very handy when tracing odd entries. This value will get refreshed if the Stack_Date value is cleared. This table also provides a handy selection list of possible tables when setting up listeners.

Defining Listeners

It generally doesn’t do any good to talk, if no one is listening. There are three parts to each listener, an entry in the Meta_Change_Listeners table (Table 4), a listener specific interface package (such as the Insite_Interface package mentioned previously) and the actually interface application, be it an import table or a custom application. Like the Meta_Change_Tables, we also record the call stack of whoever calls for this listener. Although we have some concept of role based access control built in for each queue, in all of our deployments so far, we have written a specific interface package which provides the access control we need.

[7] Cisco Unity – voice messages and email are co-mingled on an Exchange server, with access to both via both the telephone and Outlook or other email agents

Table	sub type	Description	Count
BUILDINGS		Buildings (from INSITE Space Management)	258
DEPARTMENT		Departments from the phone directory	3442
GROUPS		Unix Groups	12
GROUP_MEM		Members of Unix Groups	871
INSITE_FLOOR		Floors within buildings (from Insite)	46
INSITE_SITE		Campuses (from Insite)	1
LOCATIONS		Rooms within Buildings (from Insite)	11890
LOGINS		Computer accounts (email)	61632
PERSON	Address	Address information	209016
PERSON	Dir_Orgn	Departmenal affiliation from directory	406
PERSON	Merge	Database cleanup – really ugly	224
PERSON	PEOPLE	Basic person information, Name, DOB, ID Numbers	160850
PERSON	Status	Current status for a person (Student, Employee, etc.)	95468
PERSON	Telephone	Telephone number (home, campus, etc.)	78960
PERSON	UDI	*User Directory Information*: Class Year, web page, email address	5725
UNITY_VMAIL		Command for Unity Voice Messaging System.	6072

Table 2: Tables and sub type.

```
Create or Replace Trigger Directory_Telephone_Trig_Upd
 after update on Directory_Telephone for each row
begin
   Meta_Change_Rtn.Log_Update( tname => 'PERSON',
        subtype => 'Telephone', rrowid => :new.rowid,
        person_id => :new.Person_Id, Aux_String => :new.tele_type);
end  Directory_Telephone_Trig_Upd;
```

Figure 8: Telephone change trigger.

We currently have seven listeners defined (Table 5). Of those, three are ‘‘real time,’’ polling for changes frequently, and the others get once a day updates. In addition, both BEST and CMMS have interactive tools available to push through individual records on demand.

Linking Listeners with Tables

The last part of the puzzle is the Meta_Change_Interests table (Table 6) which defines which table and subtype pairs any given listener is interested in. This mapping is maintained with a web based tool, making it very easy to maintain these relationships. This tool also allows you to display pending and processed change counts, flush pending records (handy during development), as well as the call stacks for tables and listeners.

When a call is made to one of the Meta_Change_Rtn.Log_XXX routines, it takes the Tname and Subtype parameters, looks for listeners in the Meta_Change_Interests table (Table 6) that are interested,

Field	Type	Description
TNAME	varchar2(32)	Primary Key(1) – The table we are monitoring. Matches Meta_Change_Queue.Tname.
SUBTYPE	varchar2(32)	Primary Key(2) – An optional subtype of the table.
COMMENTS	varchar2(255)	A short description of what we are logging. Intended to help developers.
CALL_STACK	varchar2(2000)	The formatted ‘‘call stack’’ that made the a log entry. This is set when Stack_Date is null.
STACK_DATE	date	The date when the latest call stack was recorded. This will trigger refresh of the call stack data.
PERSON_ID	varchar2(65)	The source (if any) of the person_id value.
PKEY_STRING	varchar2(255)	The source of the pkey_string. This may be a composite value.
PKEY_NUMBER	varchar2(255)	The source, if any for the pkey_number. These values are generally not person_id values.
AUX_STRING	varchar2(255)	The source of the aux_string. This may be a composite value.

Table 3: Meta_Change_Tables definition.

Field	Type	Description
PROC_NAME	varchar2(8)	Primary Key(1) – The name of the valid listener. Used in the Get_Changes function call.
COMMENTS	varchar2(1024)	A description of what this listener is.
ROLE	varchar2(32)	An optional Oracle role needed to access this queue.
OWNER	number	The simon.people.id of who ‘‘owns’’ this queue.
CALL_STACK	varchar2(2000)	The formatted ‘‘call stack’’ that made the a log entry. This is set when Stack_Date is null.
STACK_DATE	date	The date when the latest call stack was recorded. This will trigger refresh of the call stack data.

Table 4: Meta_Change_Listeners definition.

Listener	Count	Frequency	Description
ADSI	25566	5 Sec	Active Directory – our primary windows 2000 domain.
Applix	148401	Daily	The trouble ticketing system for the computer center.
BEST	52633	Daily	ID card and physical access control system
CMMS	162520	Daily	Physical plant trouble ticket and payroll system.
Insite	52407	Daily	Space Management system (OFMS)
LDAP	176774	5 Sec	Directory service
Unity	16573	3 Min	Unified voice and email messaging system

Table 5: Current listeners.

Field	Type	Description
PROC_NAME	varchar2(8)	The name of the listener.
TNAME	varchar2(32)	The table that the listener (Proc_Name) is interested in.
Subtype	Varchar(32)	The subtype if applicable.
COMMENTS	varchar2(1024)	Maybe a reason WHY it is interested.

Table 6: Meta_Change_Interests definition.

and for each one, makes an entry in the Meta_Change_Queue table (Table 7). The name of the listener is set in both the Queue_Name and Proc_Name fields. When a record is processed, the Queue_Name column will be set to null. By putting an index on this field, and clearing it when it has been processed, the calls to Get_Changes can be done very quickly and efficiently.

Conclusions

At present, we have seven distinct "listeners" waiting for changes in one or more of 16 defined tables and sub types. To date, this system has processed over a half million changes. The three "real time" polling processes do not appear to put any noticeable load on the database, and in fact we have several other similar polling processors handling password changes, and they also do not noticeably load our database server. The approach of using an index on a key column that is cleared when the record has been processed works very effectively, and we will continue to use that here and with other processes, such as our password synchronization for our "single signon." We recently modified our password processing (described in [4]) to re-encrypt a password change for additional authentication realms (LDAP, Kerberos version 5, and our second Active Directory domain for the Unity Voice mail system.)

The import table/trigger approach has been very handy in providing interactive response to some of our processes and will likely be our interface method of choice when dealing with new Oracle based vendor applications, as well as internally developed applications where we want to maintain that clear demarcation between systems.

Futures

This Meta Change Queue system is fully operational and well integrated with our environment. I don't currently plan any major changes to it, but we will be making minor changes as new systems come along.

XML Output

Currently, each new listener required a listener specific interface package to be written. One area that may be worth exploring is a generic listener that generates XML. This will most likely happen when we

Field	Type	Description
TNAME	varchar2(32)	The name of the table (real or conceptual) that has been changed.
SUBTYPE	varchar2(32)	The subtype of this table – if any.
RROWID	rowid	The rowid of the record that was changed. This may be useful in speeding processing.
CHANGE_TYPE	varchar2(1)	The type of change; "I" – Insert, "C" – Change, "D" – Deletion. Some indication of what happened to the record.
PERSON_ID	number	The Simon.People.Id (if any). This is often a primary key for Simon tables.
PKEY_STRING	varchar2(32)	A varchar2 primary key value, for tables that do not use Person_Id as their key. This is optional.
PKEY_NUMBER	number	A numeric primary key value, similar to Pkey_String, only numeric rather than varchar2.
AUX_STRING	varchar2(255)	An optional extra value that might be useful the receiving system. This might be the old name.
ENTRY_DATE	date	The sysdate value when this change entry was made.
ENTRY_NUMBER	number	An ever increasing sequence number. This can be used to order changes.
PROC_DATE	date	The date when this record was processed and could be cleared.
HOLD_UNTIL	date	The time and date when this record should again be made available for processing. This can be used by other systems that can't process an event now, but want to get it eventually. Some other process will need to requeue these entries.
QUEUE_NAME	varchar2(8)	The name of the listener who is waiting for this record. This is the trigger value for pending entries. This column is indexed, and once a record is processed, this should be set to null. This will keep the index small and fast, allowing for low overhead and frequent polls.
PROC_NAME	varchar2(8)	The name of the listener. Initially, it is the same as the Queue_Name, but Queue_Name will be cleared after processing, this helps us track which listener got this record.
RETRY_COUNT	number	The number of times that this record was "put back" by the listener. This can help identify problem records and allow for back off options using the hold_until feature.

Table 7: Meta_Change_Queue definition.

get a new system that can accept an update stream in a format like that. Given the existing examples and support code, development of these interface packages has not been a problem. They are generally pretty simple and straightforward.

Status Reporting

We now have listeners automatically tied into to our process monitoring system, which will report on overall system problems. However, we have not done much with record level feedback and error reporting. In general, problem records don't get processed and cycle around for a while until someone notices them and takes appropriate action. This hasn't been much of a problem, but is something we need to look at more closely.

One of the objectives of my division, is to provide metrics for our activities. We are currently logging some periodic summaries of changes, and more formal analysis and reporting would be desirable.

Other Listeners

New listeners are generally prompted by the arrival of new systems, and as these systems are generally from other divisions, is not easy to predict what and when. We do have some existing systems that could benefit from the Meta Change Queue approach, and we will be exploring these areas. Some of them include:

- DNS configuration – providing end user tools for DNS changes, with immediate changes going via the MCQ.
- DHCP configuration – this has proven to be a "growth area" as we need to implement ways of rapidly change DHCP configuration as the result of virus scans, abuse investigations and so on.

Bulk Priority Queue

I will be adding a low priority queue, that will allow bulk entries to flow when "real time" requests are not pending. This has become an issue when mass create jobs "lock up" a listener for a long time and interactive users are trying to work. This change will be done entirely within the Get_Changes routine and none of the listeners will need to be changed.

References and Availability

Some of the examples in this paper have been edited for publication, frequently, some of the error handling code has been removed. While this should not impact your understanding of how this works, if you are going to implement something like this, I would suggest looking at the actual source code to see some of the special cases that we had to deal with. Some are very site specific, but will give you some idea of some of the details we had to handle.

This project is part of (but not dependent on) the Simon system, an Oracle based system used to assist in the management of our computer accounts [4], enterprise white pages [3], printing configuration [2], All source code for the Simon system, is available on the web. See http://www.rpi.edu/campus/rpi/simon/README.simon for details. In addition, all of the Oracle table definitions as well as PL/SQL package source are available at http://www.rpi.edu/campus/rpi/simon/misc/Tables/simon.Index.html .

Acknowledgements

I would like to thank Andy Mondore for reviewing this paper. Special thanks also go to Alan Powell, Mike Douglass, Rich Bogart and Chet Burzynski all of RPI and also Lance Holloway of BEST Access Systems and Megan Whyman of OFMS for their contributions to this project. I also want to thank Rob Kolstad for his excellent (as usual) job of typesetting this paper.

Author Biography

Jon Finke graduated from Rensselaer in 1983 with a BS-ECSE. After stints doing communications programming for PCs and later general networking development on the mainframe, he then inherited the Simon project, which has been his primary focus for the past 13 years. He is currently a Senior Systems Programmer in the Networking and Telecommunications department at Rensselaer, where he continues integrating Simon with the rest of the Institute information systems. More recently, Jon has taken on support of the Telecommunications billing system,[8] and providing data and interfaces for Unity Voice Messaging and some Voice over IP projects. When not playing with computers, you can often find him merging a pair of adjacent row houses into one, or inventing new methods of double entry accounting as treasurer for Habitat for Humanity of Rensselaer County. Reach him via USMail at RPI; VCC 319; 110 8th St; Troy, NY 12180-3590. Reach him electronically at finkej@rpi.edu. Find out more via http://www.rpi.edu/~finkej.

References

[1] Armstrong, Eric, Steve Bobrowski, John Frazzini, Brian Linden, and Maria Pratt, *Oracle 7 Server Application Developer's Guide*, Chapter 8, Oracle Corporation, pp. 1-29, December, 1992.

[2] Finke, Jon, "Automating Printing Configuration," *USENIX Systems Administration (LISA VIII) Conference Proceedings*, USENIX, pp. 175-184, September, 1994.

[3] Finke, Jon, "Institute White Pages as a System Administration Problem," *The Tenth Systems Administration Conference (LISA 96) Proceedings*, pp. 233-240, USENIX, October, 1996.

[4] Finke, Jon, "Embracing and Extending Windows 2000," *The Sixteenth Systems Administration Conference (LISA 2002)*, USENIX, November, 2002.

[5] Finke, Jon, "Process Monitor: Detecting Events That Didn't Happen," *The Sixteenth Systems Administration Conference (LISA 2002)*, pp. 145-153, USENIX, November, 2002.

[8] AXIS – Pinnacle CMS by Paetec