Awakening the 20,000 Megahertz Productivity Giant: The Batch Problem Queue Pattern

Bruce Lombardi
Vice President, Engineering
BehavHeuristics, Inc.

Introduction

Would you like to have a 20,000 MHz machine capable of performing like a mainframe sitting on your desktop? Of course you would, but even with today's rapid acceleration in microprocessor speeds 20,000 MHz is still a long way off. You may however, have 20,000 MHz of processing power sitting on "your" desktop if "your" is taken to mean your organization as a whole, and the organization has two hundred 100 MHz person computers (PCs). The problem is harnessing all of this processing power in such a way that it performs like one 20,000 MHz processor; the solution is the Batch Problem Queue pattern. This pattern, used in a client/server architecture, has successfully harnessed the collective processing power in systems with as few as three PCs to as many as 200(the latter providing the 20,000 MHz power alluded to above). The pattern has allowed client/server systems to perform tasks usually performed on a mainframe.

Despite the popularity of client/server systems and the many applications being developed using such an architecture, one very useful area of application is still being relegated to centralized mainframes -- batch processing. The crunching through of a large volume of work in unattended mode, often in the middle of the night, seems to be solely the purview of mainframes. These machines, with their high processing speed and their operating systems geared toward balancing the priorities of a number of batch jobs, are well suited to the task. However, mainframes come in only one size (big) and only one cost (high) and therefore are usually a centralized corporate resource that is not always suitable to departmental needs. The needs of a department for batch type processing, however, can be accommodated using the Batch Problem Queue pattern on a client/server system. The pattern also allows the batch process to scale up without changing the software as the needs for processing power grow.

The Batch Problem Queue pattern was developed and refined over several projects of various sizes and culminated in its use on one very large project. On one hand, an "experience report," such as those given at OOPSLA, of the largest project would have been informative and easy to write, however, it might have buried the general nature of the ideas amid the project details. Presenting the information as a Design Pattern will avoid that problem. On the other hand, many of the details discovered over the course of those projects have a direct bearing on both the ease of implementation and the performance of a system using the pattern. It would be a disservice to readers not to provide information that would let other implementors of the pattern avoid some of the pitfalls that we have discovered. Fortunately, both can be included by using the Design Pattern description style introduced in the book *Design Patterns: Elements of Reusable Object-Oriented Software* (Gamma et. al., 1995) that includes a section where implementation details and trade-offs can be discussed.. The Gamma et al. form explicitly classifies many of the elements of a design pattern and should become a standard for describing patterns that are ready to be fully documented. The body of this paper will follow the Gamma et al. form.

Pattern Name and Classification

Batch Problem Queue - Client-Server Architectural

Intent

Allow CPU intensive processing to be distributed among clients in a client-server based system thereby reducing the cost of the system as a whole and allowing otherwise unmanageably large volumes of data to be processed.

Motivation

Consider a system where large volumes of data need to be collected from diverse sources and processed in a computationally intense manner . The data from each source is related and can not be processed until it is all gathered together in one place. Assume also that the data can only start to be gathered after midnight, and that the processing needs to be done quickly so that the results are available the next morning. The CPU intensive nature of the processing is such that the processing power of a mainframe class machine would seem to be necessary to complete the job in the time frame allotted. A mainframe, however, is far too expensive for a single department to buy and the corporate mainframe is already struggling with the processing load of its current applications and has no room to provide additional services. Couple all of this with the fact that in order for the output of the processing to be of most use, it needs to be reviewed by humans, preferably in a graphical mode that allows them to easily navigate through the results and pick out the important information in a time efficient manner.

A concrete example of these circumstances occurs in Airline Revenue Management systems, where data on seats sold during a given day needs to be analyzed by the next morning so that the best decisions regarding future seat inventory can be made. In the Revenue Management system, information can often be gathered only after midnight and comes from several diverse sources, such as reservations systems, frequent flyer systems, and pricing systems. The gathered information is used for forecasting demand and no-shows, for calculating optimal seat allocations among different fare categories and for projecting expected revenues. All of these steps are CPU intensive. (In one such system, the requirements were to process 100,000 flights in a three hour window with each problem requiring multiple forecasts and optimizations resulting in a total of over 11 million forecasts in the time period). The results are then reviewed by human analysts that must be able to select and look through flights rapidly, a need that requires a graphic interface.

Given that such a system requires a high degree of processing power usually only found on a mainframe, it would seem that only project's whose benefits could justify such a large hardware investment would be approved; an unlikely event given the cost of mainframes and the staff required to run them. On further consideration, however, a solution to the problem that allows it to be solved with a departmental level mini-computer/database server becomes clear. Consider for a moment the last mentioned requirement of a graphical user interface (GUI) to review the results. On first encounter, this requirement seems to place an even greater burden on the processing power of the system, but in fact, it holds the key to the solution of the problem. The key is this: The CPU intensive part of the processing occurs overnight, and the GUI part of the processing occurs during the day. If powerful PCs are used to provide the GUI, and if the power of the PCs could be used for the CPU intensive processing at night when they are otherwise idle, then the problem can be solved without requiring a mainframe. This solution provides four benefits:

1) It allows the problem to be solved without a mainframe.

2) It makes use of valuable corporate resources that are usually wasted in the off-hours. (The 200 or so PCs that would normally be turned off, or more likely would be endlessly exercising the screen saver program all night).[1]

3) If done as suggested below, it provides a fault tolerant solution that inherently balances the processing load across all available PCs. In fact, PCs can be put in or taken out of the overnight processing operation at will, without affecting the outcome (except for the fact that more or less PCs are doing the processing and the processing will be that much faster or slower).

4) The same processing can be run throughout the day on any PCs that are available to use for that purpose. Thus it can be useful for handling extra processing loads. Extra available PCs can be designed into the system if required, but in large organizations with large numbers of PCs there are often enough PCs available from people who are absent, in meetings, or at lunch, to provide all the processing power that is needed. As long as the users of those PCs allow the application to run while they are away from their desks, those PCs will become part of the corporate processing pool..

Given a database server that houses the to-be-processed information, and X number of PCs to process the data, there are a large number of ways the processing can be divided among the PCs. For example, each problem to be processed could be numbered and each PC assigned a range of numbers to process. Given 100 problems and 10 PCs, PC1 could be assigned problem 1-10, PC2 problems 11-20 and so on until PC 10 was assigned problems 91-100. This seems to work fine, but what if on the next day there are 120 problems? Okay, we could provide an algorithm that redefines the division of problems among PCs and then somehow communicates that to each PC. But then, what happens if PC5 has a hard disk failure in the middle of processing its assigned problems? What happens if Problems 1-50 are relatively simple problems, but problems 51-100 are more difficult. PCs 1-5 finish their problems and remain idle while PCs 6-10 struggle to finish theirs, thereby wasting both time and computer power and delaying the completion of processing.

The key to solving the major processing problem is to distribute the processing across multiple PCs. The key to achieving benefit number three (fault tolerance and load balancing) is to setup the database as a centrally shared FIFO queue. This FIFO is the queue in the name of the pattern.

Applicability

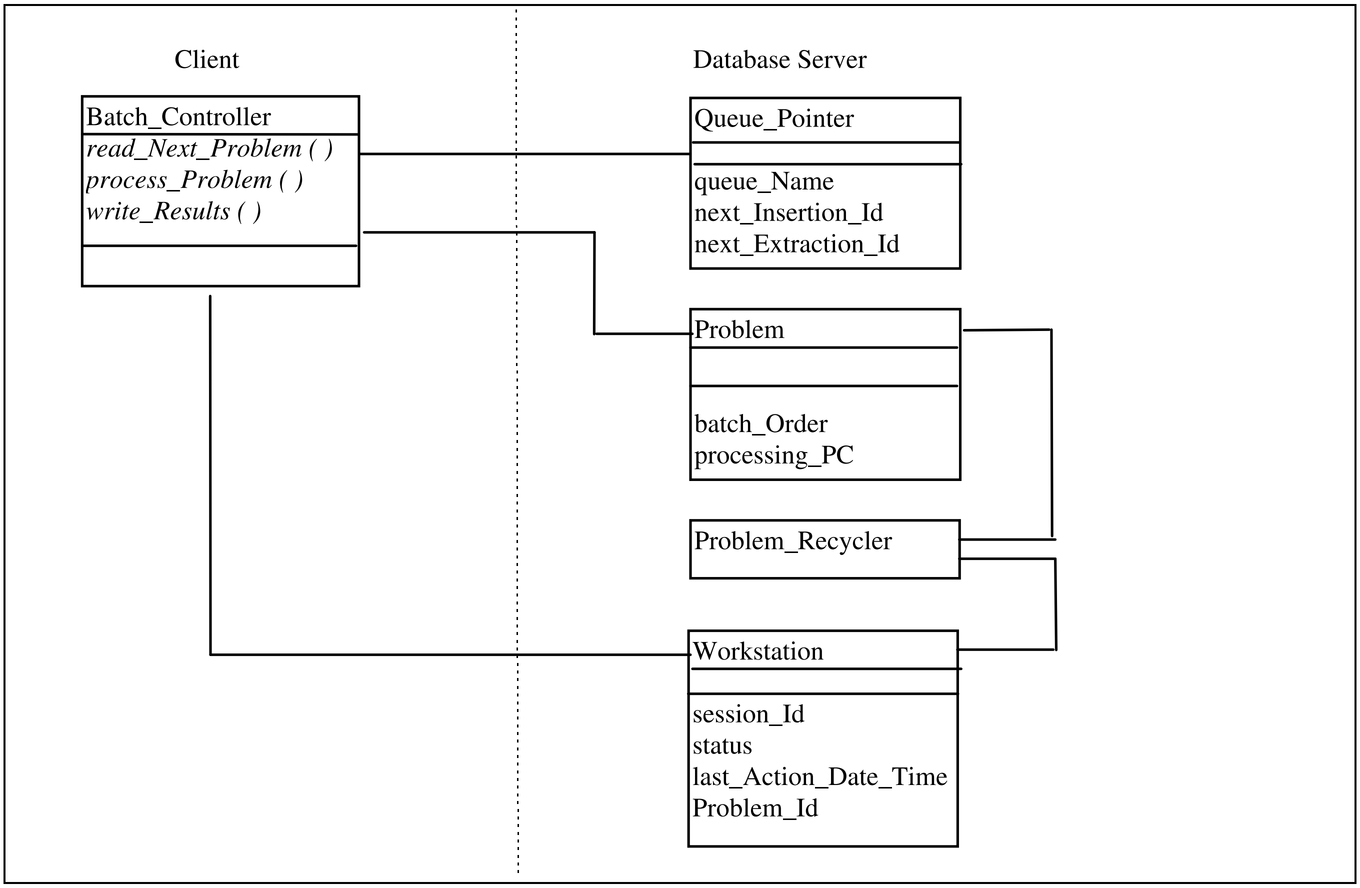
The pattern can be applied whenever a large volume of data needs to be processed and the bottleneck in processing is due to computationally complexity. Even if the pattern does not seem applicable on the basis of the time to process one “problem” if problems can be grouped together and then given out to PCs to be processed as one “chunk” it may still be possible to save a significant amount of time by using the pattern (see the discussion of “convoys” in the Implementation Section).

Structure

The following are the structural elements of the Batch Problem Queue pattern represented as an Object Diagram in OMT form (Rumbaugh et. al. 1991). The top section of each box shows the object’s name, the next section shows key operations (operations in italics are abstract operations), and the final section shows

[1] Corporations pay a lot of money to run screen savers.. In the old days, computer resources were considered too valuable to waste. Now, the costs of computing power is admittedly much lower, and the cost is more distributed across the organization, but the collective cost of the computing power in an organization is still not cheap and I see no reason not to take advantage of the sleeping giant. It doesn’t eat any more when it is awake and working but it is far more productive.

attributes. The lines shows the associations among the object which are discussed in the collaborations section.



Participants

Batch_Controller

The batch controller is an object that controls retrieving data from the database, processing the data, and storing the results back in the database. It is helpful to implement the batch controller as an abstract class and allow concrete objects to inherit the structure, which is why the operations are shown in italics. The concrete subclasses can implement the behavior of reading, processing, and writing differently depending on the context of the problem without limiting the generality of the controller. For example, in the airline revenue management problem described earlier, some airlines control inventory by "legs" which are the origin and destination airports between which planes fly, and other airlines control inventory by "segments" which are the boarding and getting-off points for passengers. A flight, with a single flight number, that flies from New York's JFK to Texas' DFW to California's LAX has two legs: JFK-DFW and DFW-LAX, but three segments: JFK-DFW, JFK-LAX and DFW-LAX. The differences in processing for these two types of inventory control system do not affect the Batch_Controller which always gathers together the entire problem (i.e., the whole flight). The concrete subclasses of the controller can be used to implement any special processing for variations on the problems such as the difference in handling leg versus segment based controls.

Queue_Pointer

The Queue_Pointer object has only three attributes (or fields). The queue_Name, which allows multiple queues to be defined, the next_Extraction_Id, which holds the batch_Order number of the next problem to extract for processing, and the next_Insertion_Id which holds the number that should be put into the batch_Order_Number of the next problem inserted into Problem to be processed.

Problem

Problem holds the values (or some of the values) that are of interest to the application. Problem is where the main control mechanism for the Batch Problem Queue pattern is found. The batch_Order number is used to select the problem(s) to be processed. The processing_PC field contains an identifier of the PC that is processing the problem.

Workstation

Each workstation object holds information about individual PCs participating in the processing. The workstation_Id field is used to uniquely identify a PC. The status field contains the state of a PC, either "waiting", "processing", or "dead". The last_Action_Date_Time field holds time-stamps of when the status last changed.

Problem_Recycler

The Problem_Recycler is run every few minutes to check for PCs that are not processing their assigned problems. When PCs are found in this condition, the problems assigned to them are re-queued for processing by other PCs.

Collaborations

Prior to batch operations, data is loaded into the database server. At that time, the batch_Order field of each Problem must be set. Usually, the batch_Order is just set incrementally for each problem based on the order in which the problems are loaded. Each problem does not have to have a unique batch_Order, and it may be better to group certain problems together by giving them all the same batch_Order. This may be done either for performance reasons or because of dependencies among the problems (see the consequences section for detailed analysis).

The Queue_Pointer fields must then be initialized. Both the extraction pointer and the next_Insertion_Id are set to "1" and the next_insertion_ID is incremented as the batch_Order numbers are filled in.

When a PC starts processing problems, the following sequence occurs:

- The PC's Id is inserted into a list of active PCs on the database server.
- The batch_Order number of a problem (or group of problems) is obtained by selecting the next_Extraction_Id from Queue_Pointer. (In one uninterruptable operation, the next_Extraction_Id value must be incremented by one to prevent another PC from obtaining the same batch_Order and processing the same problem).
- When a problem is checked out , the information in Workstation is updated. The problem_Id field is updated with the batch_Order of the current problem (or group of problems) it is processing, the last_Action_Date_Time is updated with a time-stamp of when the problem was checked out, and the status is set to "processing".

(Note: If the pattern is implemented in an environment where the database server can have active objects, then the responsibility for performing the above steps should reside in an object on the database server. In most implementation environments, the database server is a passive repository of data, and behavior must reside in the clients. In this case, it is the responsibility of the client's Batch_Controller to perform these functions).

The batch_Order number that the PC's Batch_Controller object obtained is used to select problems with a matching batch_Order. The Batch_Controller processes all problems returned.

Every few minutes, the Problem_Recycler should run and check for PCs that have checked out problems but have not finished processing them in a specified amount of time. It does this by looking at the time-stamp of when the PCs checked out the problems for processing and checking against the current system time. If it finds a PC that is not processing, it will set the status in Workstation to “dead” and will re-queue the problems that the PC was working on when it crashed. (It is also a good idea to kill any database sessions for the PC if possible to prevent the PC from “coming alive” later and writing back data). It does the re-queuing by first obtaining and incrementing the current next_Insertion_Id value in Queue_Pointer and then simply updating the batch_Order field in the Problem table with the new next_Insertion_Id number. This has the effect of logically moving the problem back onto the end of the queue for processing by other PCs.

Consequences

The Batch Problem Queue pattern has the following consequences:

1. It provides a mechanism for processing large amounts of data on a departmental size machine that might otherwise require a mainframe.
2. It allows valuable corporate resources to be used during off hours when they would otherwise be unused.
3. It provides fault tolerance, so that if one PC fails, others will process the problems. (The server, however, is a single point of failure unless redundant systems are used, but the server usually runs on an UPS in a very controlled and monitored environment and is not usually a source of failure).
4. It allows scalablity. This pattern has been used with as few as three PCs attached to a Server and as many as 200.
5. Setting the batch_Order numbers in the problems before starting the batch controllers allows the scheme for assigning those numbers to be confined to one place and allows those numbers to be changed at will without changing any code in the clients.
6. By allowing multiple rows of the database to have the same batch_Order, the pattern allows for multiple problems to be brought back for processing at once. For example, if groups of five problems get the same batch_Order, then when the Batch_Controller selects problems to process based on batch_Order it will get back 5 problems at a time. This can be an advantage for performance (see discussion in the Implementation section).
7. Additional groups of problems can be processed during the day as long as PC are available. Adding a few PCs specifically for this purpose is beneficial, but running the application in batch mode whenever a machine is unattended may be sufficient if only a small number of problems need to be processed.
8. The order in which problems are processed is not fixed. This is the only slightly negative consequence to this pattern. It means that there can be no dependencies among the problems that require them to be processed in a specific order. However, dependencies among small groups of problems could be handled through grouping them together for processing by giving them the same batch_Order number. The client side process, of course, would have to be aware of the dependencies.

Implementation

Implementation of the Batch Problem Queue pattern can be done in many ways depending on the processing environment. The pattern is general enough to be implemented with any language and any database whether or not they are object oriented. As stated above, if the processing environment allows objects to perform behavior on the database side (as does, for example, Gemstone), then some of the responsibilities of the Batch_Controller may be moved to a database object. The Batch Problem Queue pattern, however, was conceived and developed in an environment where the server was a relational database and behavior could not easily be put on the database side. As a consequence, the implementation section deals with implementation issues that arise mainly in that environment.

If you are using a relational database, all of the server side objects will be relational tables, and often, all of the information needed to process a problem will not reside in the Problem table. To put it all there would usually violate the rules of good database design (e.g., data normalization rules). This leaves the option of executing an SQL statement to join the data in diverse tables together or inventing a different approach. Joining tables tends to be slow and use a lot of resource on the server, a different strategy of retrieving a record from the Problem table (using the batch_Order number) and then using the key fields contained in the problem record to retrieve the related records from other tables is recommended. A potential problem with this strategy was that a lot of time could be spent issuing queries and waiting for data to come back from each table (in our application, the data were spread across five tables). Consider what would happen if each problem had a unique batch_Order number. Processing five problems would mean 25 "round-trips" to the database with a wait for the completion of each query and all the incumbent communication overhead. Thus a server resource problem is solved, but another performance problem is introduced.

The solution to the second performance problem is to group the problems so that the first query returns multiple records, and the keys for those records are then used to select multiple records from the other related tables. In this way processing five problems only requires 5 trips to the database, one for each table. (Since this pattern was developed during the design of and airline revenue management system in which flight information was being retrieved from the database, this grouping became known as a "convoy" of flights). The size of a convoy that is most efficient depends on a number of factors including the type of problem, the number of tables the information is spread across, and whether or not the information for an entire convoy fits into a single block (see below). The bad news is that the best convoy size is hard to predict ahead of time, but the good news is that convoy size is simply a function of how batch_Order numbers are assigned a process that is totally controlled at data loading time on the server side. It is therefore trivial to vary the convoy size and do batch runs until the best performance is found.

If you are using an Oracle database you may experience a problem with locking. Locking seems to sometimes occur at the level of the block instead of the row (this was true at least for one version of Oracle 6.0 on one machine , a Sequent 750 running Dynix/PTX, but may not be true on newer versions of Oracle or on different operating system). The problem with locking at the level of the block occurs when multiple PCs are attempting to update information stored in the same block as when they are writing back the results of processing. The symptom of this condition is much slower performance. The solution is to organize the data so that separate blocks are being accessed at any one time, and the key to organizing the data is to group the problems the right way. This can also be done by assigning their batch_Order numbers so that no group of problems extends over more than one database block.

Also, if Oracle is used, consider taking advantage of sequence number generators instead of using a Queue_Pointer table to track the next insertion and extraction Id. This Oracle feature allows clients to make select statements that retrieve a number and automatically increment the number in the same step. This guarantees unique numbers will be given to each client without the necessity of locking and unlocking any data. The sequence number generators are faster and easier to use than the queue pointer table, but they are not without problems. The main problem that arises is that **every** time a selection is made the number will be incremented. There is no way to just check what a sequence number is without incrementing it. This means that even after the queue is empty the next_extraction_Id will increment each time the queue is checked making it difficult to keep the insertion Ids and extraction Ids synchronized when processing approaches the end of the queue. Some clever techniques can be used to get around this problem, but their description is beyond the scope of this article. The sequence number approach is, of course, the easiest to use if you are only putting problems on to the queue at startup time (i.e., no Problem_Recycler type recovery is required and no additional problems are added during processing).

As a final implementation issue in a relational database environment, consider the use of stored procedures. The implementation shown in the sample code does not use stored procedures, but most relational database vendors now support them, and some improvement in performance may be obtained by the fact that the queries in a stored procedure are only compiled once and then executed using parameters passed in from the clients.

Sample Code

The following code fragments provide a feeling for how the pattern is set up in a relational database environment, and also shows techniques for avoiding some implementation pitfalls.. The code fragments use standard SQL, but some also rely on Oracle's PL/SLQ. It is hoped that providing a concrete example in PL/SQL will be helpful even for readers working in other relational databases (e.g., Sybase, SQL Server) which do not support PL/SQL. Most modern relational databases support some type of procedural additions to SQL and many of them now support those additions as stored procedures. It should be easier to translate from the concrete PL/SQL example than to develop the code from scratch. The sample code will demonstrate how to setup the Batch Problem Queue and batch_Order and extract records. Sample code for inserting new problems onto the queue dynamically, handling crashed PCs, and reinserting flights that were being processed by them will not be presented.

Problem Queue Setup

```
/* The following code creates and initializes the Queue_Pointer table */
create table Queue_Pointer (
        queue_Name char (32)
        next_Extraction_Id number (8,0)
        next_Insertion_Id number (8,0));

insert into Queue_Pointer values (‘Problem_Queue’, 1,1);

/* The following code would be executed after loading data and is designed specifically to
   set the batch_Order numbers without incurring a problem with locking on update */
DECLARE
        b           number      := 0;
        blkid       char(12);
        blkcnt      number;
        c           number;
        convoy      number      := 5;
        oldblk      char(12)    := 'xxxxxxxxxxxx';
        r           number;
 --
 cursor prbcur is
     select substr(rowid,15)||substr(rowid,1,8) blkid
     from   problem
     order  by 1, rowid
     for update of batch_order;
 --
BEGIN
 select count( distinct substr(rowid,15)||substr(rowid,1,8) )
 into   blkcnt
 from   problem;

 for f in prbcur loop
   if oldblk <> f.blkid then
     oldblk := f.blkid;
     b      := b + 1;
     c      := 0;
     r      := 0;
   end if;
   --
   if r >= convoy then
     c := c + 1;
     r := 0;
   end if;
   --
   update problem
     set batch_order = b + blkcnt*c
    where current of prbcur;
   --
   r := r + 1;
 end loop;
 commit;
END;
```

Queue Extraction

```
/* The following is an Oracle PL/SQL block used to find the next problem to process and to update
   the Workstation table */
DECLARE
        thisExtractionValue Number;
        nextExtractionValue Number;
        BEGIN
<<Try Again>>
        /* Select insertion and extraction pointer if not at end of queue.
                Signal exception if queue is empty */
        SELECT next_Extractiion_Id, next_Insertion_Id
                into thisExtractionValue, nextInsertionValue
                from Queue_Pointer
                where queue_Name = 'ProblemQueue'
                and next_Insertion_Id > next_Extraction_Id
                for update of next__Insertion_Id, next_Extraction_Id;
        UPDATE Queue_Pointer
                set next_Extraction_Id = thisExtractionValue + 1
                where queue_Name = 'ProblemQueue';
        commit;

        UPDATE Workstation
                set status = 'processing',
                problem_Id = thisExtractionValue,
                last_Action_Date_Time = sysdate
                where workstation_Id= 196936 /* Oracle session id for this PC's session */
        UPDATE Problem
                set processing_PC = 196936 /* Oracle session id for this PC's session */
        where batch_Order = thisExtractionValue;
        IF SQL%NOTFOUND then
                rollback;
                GOTO TryAgain;
                END IF;
          COMMIT; /* To stabilize PC for recovery purposes */
EXCEPTION /* Here if queue is empty */
        WHEN NO_DATA_FOUND THEN begin
                rollback;
                update Workstation
                        set status = 'waiting'
                        problem_Id = 0,
                        last_Action_Date_Time = sysdate
                where workstation_Id= 196936 /* Oracle session id for this workstation's
session */
                commit;
        end;
END;
```

Retrieving Problems

When the above statement has completed, the problems marked for retrieval can be retrieved with the following statement. (The DECODE statement in Oracle works like a case statement in this case it is used to ensures that when the queue is empty we don't retrieve problem_Id = 0):

```
Select * from Problem where batch_Order = (
        select DECODE ( problem_Id, 0, -9898)
        from workstation where workstation_Id= 196936 /* Oracle session id for this
                PC's session */
```

Known Uses

The only know use of this pattern is in BehavHeuristics Inc.'s airline revenue management system which uses the pattern to forecast and set optimal booking limits on discount and full fare seats for airlines. The revenue management Problems involve three processing steps: a forecasting step that makes use of a unique and proprietary neural network architecture, an optimization step which uses techniques from Operations Research where the marginal value of making one more seat available at a discount as opposed to holding out for a full fare passenger is considered, and a booking simulation step which uses probability distributions and makes some assumption about how discount versus full fare passengers will materialize. Forecasts of demand and no-shows and estimates of confidence in those forecasts, optimal booking limits, and projected revenue numbers are written back to the database for each booking class on each flight segment. At BehavHeuristics' largest client, over 100,.000 flights are process overnight in a three hour time window. These flight all have 10 booking classes and many of the flight have multiple segments so that over 11 million forecasts are made during the time window. The processing is distributed across 150-200 PCs, and a Sequent 750 with 10 CPU houses the data (a large machine but still much less expensive than a mainframe). A smaller client has used this system with only seven PCs and a single processor Pentium based machine as the server.

Currently there are no other known uses of this pattern, however, when I have informally explained the pattern to technical people, many of them became excited over the possibility of using such a pattern to problems in their organization. One discussion with an individual involved in processing bulk orders for magazine subscribers felt that the pattern could be useful in keeping the accounts up to date for his large corporate clients.

Related Patterns

A written pattern is currently under development which takes the ideas concerning grouping of problems mentioned in the implementation section as "convoys" and elaborates upon them. The Stars pattern described by Peterson in the PLOP book (Coplien and Schmidt, 1995, p. 163) describes a pattern for developing a schema that is optimized for information retrieval in client/server based decision support systems. That pattern would be useful in organizing problem information for the Batch Problem Queue Pattern. Also, in the same book (p. 51), Wolf and Liu present a pattern language that provides a framework for dealing with many of the issues faced when implementing object oriented workstations that are communicating with non-object oriented systems.

Conclusion and Summary

This paper has presented an architectural pattern that has proven useful in an application that has been scaled to address the needs of organizations and customers vastly different in size without any code changes.

I strongly believe that the Batch Problem Queue pattern shows the usefulness of having Client/Server systems where a lot of processing power is available on the clients. There has been some talk lately about converting Smalltalk to a system where much of the processing would be done by a "headless" (i.e., without a user interface) image running on a server and the clients would just provide the user interface (much like an X station). While I could name many good uses for a headless Smalltalk., I believe that a wide variety of options for distributing processing should be available and power should not be confined to a central server. Such a restriction would make the intelligent use of computational resources as shown in the Batch Problem Queue pattern impossible to implement.

Acknowledgments

Sam Youngman of Relational Analysis Inc. and Alan Purdy of ReUsable Solutions Inc. refined and improved on the implementation of the pattern over several projects. Alan Purdy was also responsible for the concept referred to here as “convoys” which are very important to performance. Early implementation of Batch Controllers were done by Ward Cunningham and Don Berkeley (both at Digitalk at the time), later refinements were added by Ken Otwell at BehavHeuristics and Wolfgang Franzen at ReUsable Solution.

The author would like to thank Natalie Kharlaf , and Sharon Hormby of BehavHeuristics for their comments on the manuscript, and in particular would like to thank Paul Keefer of BehavHeuristics for his helpful suggestions in converting the manuscript from an experience report style to pattern style.

References

Gamma, E., Helm, R., Johnson, R., Vlissides, J. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, 1995.

Peterson, S Stars: A Pattern Language for Query-Optimized Schemas. In Coplien, J. O., and Smith, D. C. Eds. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.

Rumbaugh, J., Blaha, M., Premerlani, W., Eddy, F., Lorenson, W. *Object-Oriented Modeling and Design. Prentice Hall, Englewood Cliffs, NJ, 1991.*

Wolf, K. and Liu, C. New Clients with Old Servers: A Pattern Language for Client/Server Frameworks. In Coplien, J. O., and Smith, D. C. Eds. *Pattern Languages of Program Design*. Addison-Wesley, Reading, MA, 1995.