## 1 Slide pst_vgrf_0115 – PIA Data Handling

PIA expects large volumes of data (terabytes, petabytes, exabytes and beyond) as a matter of course. The entire design of the CORBA-served implementation was predicated upon this realization, so much so that the topic is generally not even mentioned. Because of this presumption, every kind of object with which PIA is built is based on a highly-extensible, distributed, data handling infrastructure.

To begin this process, PIA adopts an extremely accommodating standard for the identification of individual objects. This identification consists of

1. The IP address of the creating machine (which is coded to support IP V6 addresses when those come into widespread use),
2. The process identifier of the serving program within the machine,
3. The time of object creation (expressed as 64-bit count of seconds since midnight, January 1st, 1970 in univeral, coordinated time),
4. A 64-bit count unique within the context of the serving program, and
5. The type of the created object.

All of this identification combines to create what is, in effect, a very generous "virtual address space" for PIA objects. In simple terms, the PIA object identification space would allow every machine on the (eventual Vesion 6) Internet to create 16 billion billion objects of the same kind every second for the next 500 billion years before an object identification was ever duplicated.

Obviously, no single server or cluster of servers could reasonably expect to serve so many objects, or even a tiny fraction of such an object space, at one time. Because of this, the PIA infrastructure defines the concept of an active object – that is, an object actually doing something useful at the present time – as opposed to an inactive object. Active objects are served while inactive objects are identified, deactivated, and their internal state stored to disk (or other) storage, a process called *etheralization* in CORBA parlance. This unburdens any individual server by requiring it to instantiate only those objects presently doing something. Past PIA prototype experience has suggested that this strategy will be quite effective in reducing the difficulties associated with large volumes of data.

The amount of data stored in any given object is, of course, a matter left to the kind of the object. Some objects may simply be organizational in nature, storing only references to other objects, while others may contain large amounts of data, for instance entire geographical survey maps and the like. The key consideration here, though, is one of overhead costs balanced against ease of service. Objects having only a very small amount of useful information will, inevitably, pay a considerable overhead cost in infrastructural information to contain their few bytes. On larger objects, beginning perhaps in the range of tens of kilobytes, that overhead will begin to shrink to acceptable levels. Very large objects, on the order of perhaps ten or one hundred megabytes, will of course incur no perceptible overhead costs; however, on smaller machines such objects may become cumbersome to handle, requiring for example the machine to find one hundred contiguous megabytes of memory space in which to place the instantiated object.

To further ease the difficulties of extreme data volumes, the CORBA-served implementation of PIA has been designed from the beginning to be an inherently distributed system. Any given PIA application or information resource has the option of being served not by just a single machine, but by a cluster of machines cooperating as a unified whole. While not yet completely implemented, basic infrastructural "hooks" have been put in place that will eventually allow any given object to be served by several members of such a cluster. This will not only increase data availability in the event that members of a cluster must be paused for maintenance or other purposes, but is also planned to be a part of load-leveling strategies that attempt to evenly distribute the current burdens over the members of the cluster.

Finally, PIA supports a diverse set of mechanisms for storing the persistent state of inactive objects. A server, or a member of a server cluster, is of course expected to have access to a local storage server or storage server farm for this purpose. Rotation among multiple members of a storage farm is an implicit part of PIA's utilization of such a facility. Additionally, PIA can explicitly store persistent information to multiple locations so that, should a particular storage server fail, the desired information can still be obtained from a backup source. Also, the PIA storage infrastructure can utilize the facilities of CORBA to store persistent object states to remote storage servers reachable through the Internet. Finally, this remote storage server mechanism can be used to allow a client to store proprietary information only on his own storage servers even when he is using the PIA-served information processing resources of other parties.

## 2 Slide pst_vgrf_0116 – PIA Speed of Data Access

When dealing in such huge data volumes as PIA anticipates, no system can reasonably expect to be characterized as "fast" in the sense that most people understand that term in the context of computing; however, PIA applies tools that are expected to allow expeditious access to such volumes of data.

The first requirement of expeditious data access is that the data must be sorted in some manner so that a search for a desired item can be directed toward that item. Exhaustive searches, in which every item is picked up and examined for the desired characteristic, will never be efficient by any standard and, when extended to the exa-item (that is 2 to the 60th power or approximately a billion billion items) range, will become truly exhausting.

In a great deal of modern programming the sorting method of choice is the hash table because of the hash table's order *1* search performance. It must be remembered, though, that this search performance is based upon the hash table's size being some modest fraction of $n$, the sorted item count. Frequently, implementors of hash table solutions go to the additional effort of allowing the hash table to dynamically re-size itself as the number of sorted items changes. Given the speed of the hash table, the size and additional complication of dynamic re-sizing are equitable trade-offs.

The hash table was an untenable choice for PIA, though, given the fundamental presumptions of data volume. To sort a billion billion items effectively would require a hash table of perhaps 10 or 100 million billion entries. Creating a hash table so large is simply impractical and the cost of re-sizing such a table as data volume changes is nearly unthinkable.

As a result of the impracticality of pursuing hash-table technology to the desired level, PIA settled upon the balanced, binary tree as its primary sorting mechanism. The balanced, binary tree provides a reliable, scalable order *log n* search performance. This means that a search for a particular item out of an exa-item tree will require about 60 key comparisons. (To obtain similar performance, a hash table would have to have about ten million billion entries.)

Because the PIA balanced, binary tree sorting system is built upon PIA's distributed, active/inactive object infrastructure, the burden of serving an exa-item tree can be spread over a cluster (indeed, most probably a large cluster) of server machines and storage farms. This requires no special invokations or incantations, but merely falls out of the nearly unavoidable act of building with the tools supplied by PIA.

There is another aspect, though, to the speed of data access: the speed at which the information of a particular object can be obtained when that object has been deactivated and its state stored on some secondary storage device. Since file systems typically resort to simple linear searches to find a file of a particular name, simply dumping all the billion billion files holding the internal states of a billion billion inactive objects into a single directory is untenable – it would result in an average search of one-half billion billion file names to find any particular file.

Again, PIA resorts to a tree structure to reduce the magnitude of this problem. While a matter of server configuration, typically a *16-way* directory tree structure with a depth appropriate to the anticipated file load is configured. For example, a *16-way* directory tree 15 layers deep can accommodate 16 billion billion files with an average lookup of only 128 filename comparisons (an average of 8 comparisons for each of the 15 directory layers plus a final 8 in the file-containing layer) to locate any particular file.

Now we can consider the composite speed of these two systems. Consider a PIA-implemented tree sorting information for a billion billion items. Suppose that all the objects of that tree are initially deactivated and that the files containing the internal states of those objects exist within a PIA directory structure configured to handle the 16 billion billion files discussed above. How long does it take to identify one particular item for which we have the sorting key? To begin with, we will have to examine, and therefore instantiate, an average of approximately 60 different objects to perform the necessary key comparison in order to navigate across the tree to the desired object. In order to instantiate each of those objects, we will have to find their associated state files from among the 16 billion billion files in the storage server, a process that takes on average 128 file name comparisons. So, overall, we will have to do approximately 7,680 file name operations (60 objects times 128 file name comparisons for each object) to locate the desired data.

The question of access speed now becomes one of determining how fast we can perform the 7,680 file name comparison and the less-significant 60 file open operations. This, of course, depends on many factors: how busy is the storage server, how fast is the connection between the storage server and the object-serving machine, so on and so forth. Let us just grab a number out of the air: let us suppose that we can perform a file name comparison in about 10 milliseconds. (This is about the speed at which a modern disk drive can move its heads and about two orders of magnitude greater than the rotational latency of such a disk drive.) At this speed, we can estimate the time to do our 7,680 file name comparison operations as being about 77 seconds – about a minute and one quarter.

Waiting something over a minute for the piece of information you want to come up may not seem like a staggering performance at first; it is diffcult to generate a real conceptualization of what finding one particular piece out of a billion billion items really means. Consider this analogy, though: suppose that the budget of the United States is one trillion dollars – that's $1,000,000,000,000 – per year and is held flat at that value for the next million years. Now suppose that all that money is printed up in one dollar bills and stacked around you. That would be a cube of dollar bills a little less than 8 miles on each side; about 500 cubic miles of dollar bills in all; it would be a cube that would pretty much cover downtown Cleveland, Ohio, and reach up to where the trans-continetal airliners fly. Let us suppose that as you stacked up all these dollar bills, you took notes or sorted them by serial number somehow. How long do you suppose it will take you to find a particular one dollar bill, by serial number, out of all those mountains of money? A little over a minute, perhaps? That is what it means for PIA to find one particular data item out of a billion billion such items in a little over a minute.

## 3 Slide pst_vgrf_0117 – Impact of Self-Revealing Semantic Infusion Technology

Most traditional database tools rely upon a record structure technology in which each of the fields of any given record are relatively fixed. The content of each such field is usually specified at the time the database is designed and consumers of that database must generally be built with a knowledge of that specification. This makes the problem of consuming information from different database designs more difficult because the consumer must be built with knowledge of the specification applicable to each database to be utilized.

A further burden of flexibility is placed upon the consumer of multiple database applications in that it must track the source of each obtained record so as to know what additional information might exist in that record as opposed to a record obtained from another source. For example, some personnel databases may contain Social Security Numbers while others may not. A consuming application spanning such different databases must be aware which will provide such information and which will not so as to provide appropriate protections when the information is available (since records are generally read as a whole and coming into possession of the number is unavoidable), and to avoid inappropriate operations when it is not.

PIA is not built upon the principals of fixity and fore-knowledge, but is instead based upon the technology of self-revelation. The kind of information available in any given application or information resource and, indeed, in any particular configuration of that application or resource, must be discovered by the consumer as it is found and need not be known in advance. While this does lead to some additional complexity on the part of the consuming application, it is a complexity universally applicable to all encountered applications. Inquiry can be made of each information source as its found as to the kinds of information it offers. If the desired kinds do exist, then a search can be made for “records” having the desired values. There need be no concern for the record format for there is no record format: only a set of discrete items of information grouped together into a whole. There is also no concern for what other information might be present: if it is not needed, it is simply not accessed; if it is needed, it can be searched for and discovered by its kind.

Algorithms that wish to apply themselves to found “records” of information have the facilities through the technology of self-revelation to determine whether or not the conditions for their operation do, in fact, exist. For example, an application that must have a Social Security Number for use as a taxpayer identification can determine whether or not that item of information exists in the presented record. The algorithm need not be built with fore-knowledge as to the specific record formats it can successfully operate on, but instead examines each record as operation upon it is proposed. The seamless consumption of information is furthered.

Furthering these capabilities is PIA's use of semantic infusion through class derivation: information "knows" about itself and is, in a small sense, self-aware as well as self-revealing. For example, the encapsulated financial value of a transaction knows in what units of currency it was made – dollars, yen, pounds sterling – and, depending upon the situation, it may know the exchange rates in effect at the time of the transaction, or it may be able of its own volition to obtain the current exchange rates. Such an item of information might even be encoded with a knowledge of present value analysis so that even though it was a payment made then, it might be willing to tell its consumer what that is worth now. The effect is that consumers of information are further unburdened; they do not have to know that records from this database record payments made in dollars while records from that database record those payments in yen.

## 4 Slide pst_vgrf_0118 – Areas of Anticipated Improvement

PIA was originally conceived not as a database technology, but as a technology for combining technical applications into an integrated, functional whole for the complex, multi-fidelity, multi-disciplinary analysis of systems. As such, a number of inital design choices were based upon presumptions not entirely appropriate to extreme database applications.

One such presumption was that, while iterations upon sets were contemplated, those sets were considered to be both manageable in size and inherently dynamic in nature. These expectations gave rise to the design of the position iterator which, at present, encapsulates the entirety of a set iteration into a single iterator instance.

The difficulty with the present positional iterator is that it will become unmanagably large as the set to be iterated over approaches a practical infinity. The iterator instance is coded to hold a reference to every member of the iterative set and that accumulated set of references in a single instance may simply burst the capacities of a single server of that instance.

To alleviate this difficulty with the size of a single iterator, it is expected that an implicitly-segmented iterator scheme can be devised within the presenting framework. When the number of references added to the apparent iterator instance exceeds some threshold, that instance will create additional iterator instances and segment the whole of the iteration across the collection of implicitly-created individuals, while still acting as the apparent central focus of the iteration. This will allow PIA's inherent technologies of distribution to be brought into play.

A possible advantage of implicit segmentation is foreseen: it may be possible to apply multiple threads of execution to such segmented iteration, thus bringing additional processing power to bear upon the processing of a near-infinite set under traversal. This presumes the applicability of such concurrent processing techniques; a cumulative algorithm may not be able to use such features conveniently.

Refinements may also be introduced to the set iteration support provided by some PIA information structures. Currently, set iteration necessarily involves all the members of the information structure. It would certainly be possible to add certain fixed forms limiting this set to members of interest. For example, an iterative initialization might select only those elements having a key value greater than one given value, $A$, and less than another given value, $B$. More flexible, adaptive forms might also be devised.

Another fundamental presumption of set iteration previously mentioned was that sets were inherently dynamic; that is, that the members of the set could and would be changing while other processes were traversing that set. Because of this presumption, the iterator instance is connected to the set on which it is iterating and is informed of those changes. Moreover, this presumption disallowed the implementation of more traditional methods of literal set traversal; that is, traditional functions in the manner of **GetNext** were not implemented because of the inherent unreliability in the face of a dynamically-changing set and their limited ability to inform an iterating process that the set had, in fact, changed during the iteration. Such functions, though, are not in the least difficult to implement and, given an environment in which inherently static sets may be more frequently encountered, may well be worthy of implementation at this later time.

## 5 Slide pst_vgrf_0119 – Expected Growth in the CORBA-Served Implementation

The original C++ prototype implementation of PIA was just that: a prototype designed to demonstrate the feasibility and mechanisms of basic concepts upon which PIA was to be built, those being principally the technologies of self-revelation and semantic infusion through class derivation. It has always been expected that the migration of PIA to a CORBA-served form would offer the opportunity to greatly expand on these foundational concepts.

One of those expected conceptual expansions is the application of semantic infusion through class derivation to offer various kinds of applications. The exact hierarchy of such a derivational tree is not entirely settled as yet; however, it would seem clear that the concept of a "database" (or information resource) application whose primary purpose is the offering up a volume of information for inspection would be one of those natural kinds of applications.

One of the key elements of the semantic infusion technology is the ability to add functionality appropriate to the semantics of the encapsulated kind. For a database application support for various standard query mechanisms, for example Structured Query Language (SQL), would be a natural functional addition. To support PIA's self-revealing nature, other query functionality to reveal the kind of information actually contained by a given database application might also be added.

One key distinction that may result in different kinds of database applications is the location of the actual data. One kind of database application may offer a portal to data managed by a traditional database tool while another kind of database may contain the data itself. Both kinds of database application have their advantages and disadvantages.

The first database application kind, acting as a portal to data managed by a traditional database tool, has the disadvantage that PIA's support for information extending to a practical infinity is set aside. While all the mechanisms for such volume are still in place in the PIA application, the limit on data volume is set by the database tool being exposed by the PIA portal. The advantage of the portal application, though, is that the semantics of the parameter configuration tree are relatively natural: the tree can follow the path of queries pursued by the consumer of the database application, producing an auditable record of what was searched for, what results were returned, what paths of investigation were not pursued, and the like, much as the PIA application conceptual design intended.

The second database application kind, in which the PIA application supplants traditional database tools and becomes the holder of the data, again has the advantage that PIA's support for information volume extending to a practical infinity can now be utilized – there is no traditional tool imposing any limitation. While perhaps not so much a disadvantage as an uncertainty, the use of the PIA parameter configuration tree in such an application is now less clear. In traditional database tools, all records are peers; there is no hierarchy. On the other hand, the *n-ary* tree of the PIA parameter configuration tree has a definite parent/offspring hierarchy. Some database applications might find the parameter configuration tree hierarchy very natural to the structure of the data, some might not.

A further consideration of the parameter configuration tree issue in the second database kind is that some applications might find the *n-ary* tree presumption unnatural to their kind. Specifically, geneological databases would fit much more naturally into a parameter configuration graph (in which each node of the graph represented a specific person) than into the restrictive *n-ary* specialization of this tree. Since the parameter configuration tree is constructed using an interface that is, in fact, a directed graph node interface, it might be expected that a further derivation of the second kind of database application, designated a geneological database application, might be defined in which the *n-ary* tree restriction of the parameter configuration tree is turned off, with the result that parameter configurations become a full-fledged graph. Exactly what import this has for other PIA-wrapped consumers of information is unclear; however, the technology of PIA self-revelation allows such consumers to inquire, determine that parameter configurations are in a full, directed-graph form, and act appropriately.