COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

PROGRESS SOFTWARE BUSINESS ALLIANCE
ISV Partner

Summary Use Case for Multi-threading Progress Sessions

Thomas Mercer-Hursh
9 January 2006

Use Case:	Summarizes multiple use cases relating to the implementation of multi-threading capabilities in a Progress ABL session including OERA Architectural Patterns – Data Access Objects, OERA Architectural Patterns – Services, Model-View-Controller Separation, Long Running Task, Real Time Application, Handling Events, RMI Response, Handling Multiple Socket Connections, and Dump and Load.
Product Area:	4GL Language
OS:	All
Description:	**Major Architectural Uses** The OO capabilities introduced in OE10.1A lead one to want to use established OO architectural principles (e.g., Fowler's Patterns of Enterprise Application Architecture) to implement the OpenEdge Reference Architecture. A frequent core characteristic of these patterns includes multiple components executing simultaneously and interacting closely. For major OO languages, this typically means multiple processes operating within a shared "space" in which they can interact efficiently along with multiple threads within each process. In terms of the current Progress "session" (see below), this implies a need for both new mechanisms by which multiple processes (sessions) can interact, which is not covered in this discussion, and adding the ability for a single session to be multi-threaded. In many ways, this latter capability is the more essential since it is the one which is likely to provide adequate performance for fine grained interaction between objects. Some examples of the important architectural uses include: OERA_Architectural Patterns – Data Access Objects: A standard approach in OO architectures is to provide a data object for a particular table and to have many processes share the use of that object as their source of the data from that table. This approach provides close control over access to the table, encapsulates table access in a minimum number of classes, enables intelligent caching of smaller tables, and allows implementing sophisticated data access features such as providing optimistic locking within the object without the need for multiple trips to the database. Among caching possibilities for code tables and other such rarely changed data like a table of states, a shared data object could cache a copy of the table and provide it to all processes needing that data until the machine was rebooted without having to ever access the database more than once. A read-only copy of this object can be shared with other sessions or services. This pattern cannot be effectively implemented within a session without multi-threading. OERA Architectural Patterns – Services: A classic model for a Service in a Service Oriented Architecture is a lightweight façade object which instantiates necessary component objects such as data sources, and then listens for service requests, spawning an independent object to handle each service request. This allows the façade object to remain blocked for very, very short periods of time and thus remain highly responsive. This pattern is only effective if the request handling objects can execute independently, utilizing shared data and other objects as needed, and returning status and results to the façade object when available. Multiple processes are not really appropriate to this pattern because of the need for fine-grained interaction between the objects and for the need for this interaction to be highly performant.

Therefore, the most effective implementation of this pattern would be a multi-threaded session.

One desirable architecture would use a single session per service and one service would be considered the "master" for any one set of data, being responsible for obtaining the data from the data store and handling any updates. Every other service needing that data would typically obtain a read-only copy from the master service.

Model-View-Controller Separation: One of the most classic OO patterns is the Model-View-Controller pattern for defining the structure of any user interaction as a separation into Model (domain or business logic), View (user interface), and Controller (flow control). In any circumstance in which the Model component involves a long running process, one needs multiple simultaneous lines of execution, i.e., multi-threading, in order to retain a responsive interface. A trivial, but very common, example is the desirability of displaying a "progress bar" during an extended inquiry or computation. Similarly, the ability to interrupt an on-going process of this type requires, at a minimum, that the View and Controller remain responsive.

While a complete response to these patterns implies both multi-threading and good inter-process communication, the current discussion is focused only on the multi-threading aspect. It is believed that PSC already has an understanding of the need for better inter-process communication. These two capabilities are complementary with inter-process communication being appropriate for "coarse-grained" interactions such as might occur between Services and multi-threading appropriate for "fine-grained" interactions such as might occur within a Service.

Other Uses

Long Running Task: As noted in the discussion of Model-View-Controller, when a session has a need to execute a long running task, there is a need to display a progress bar, provide potential interruption, and to process other events while the long running task executes. This need exists beyond the specific context described above, but rather is a general need that exists in non-OO programming and programming with no UI. A special case of this need is breaking down a large query into sections, returning partial products to the main thread for display or action while additional records are collected in the background.

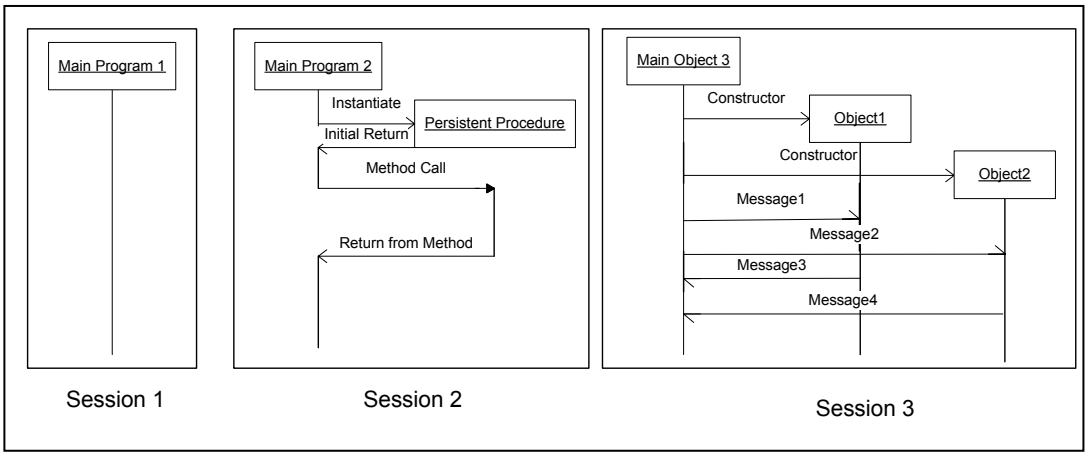
Real-time Applications: Current P4GL capabilities are not equal to the task of interfacing to systems requiring true real-time response because the application is blocked from handling new events while handling the prior event. Multiple threads would allow a thread to be spawned for each event, leaving the event responder free to continue to process new events.

Handling Events: This is related to the issue in Real-time Applications, but refers to the desirability of launching a thread to handle one or more particular events, thus not blocking the main thread from processing while waiting for the response, and retaining an appropriate state for handling the event.

RMI Response: In a similar fashion as for Handling Events, separate threads can be used for handling RMI requests, particularly synchronous ones, so that the potentially lengthy process need not block the entire session from other work.

Handling Multiple Socket Connections: At present, it is not feasible to write a general purpose socket server because this implies the ability to spawn a new thread for each socket connection.

Dump And Load: By spinning off separate threads for each table, up to some specified maximum number of threads, considerably greater throughput could be achieved without the

	current need to start multiple separate processes. In terms of priorities, the architectural patterns are of much more sweeping impact and importance than these latter examples, if only because of their scope.
Actors:	Because the use cases considered in this discussion are actually patterns in modern software development, the "actors" are primarily the programmers seeking to develop such applications, although it is certainly the case that the users of applications developed with these capabilities are likely to experience a substantially more desirable user interaction because of the use of these capabilities. In particular, users would have more responsive applications generally and would not experience long periods of "locked" sessions while work was being done in the background. Similarly, while not tied to any particular actor, the benefits of a Service-Oriented Architecture are well understood and full exploitation of this pattern requires multi-threading.
Assumptions:	The following definitions are used in this discussion: Session: A single Progress process as launched by mpro or similar command. Process: A separate sequence of execution whose activity is managed by the operating system and are typically independent, carry considerable state information, have separate address spaces, and interact only through system-provided inter-process communication mechanisms. Thread: A separate sequence of execution whose activity is managed within a process and which shares some degree of context with other threads in the same process and share memory and other resources directly. Context switching between threads in the same process is typically faster than context switching between processes. See http://en.wikipedia.org/wiki/Thread_(computing)#Threads_compared_with_processes for a discussion. In the following diagram there are three sessions:  Each session consists of a single process, which is allocated processing time according to the scheduling algorithm of the particular OS. In the first session a simple linear execution sequence is shown, i.e., the type of execution one would expect of a procedural model program. In the second, a main program instantiates a persistent procedure, continues processing, calls a method in the PP, which in turn does some execution and then returns to the main procedure. Only one of these two possible lines of execution is active at any given time. In the third session, the object representing the main line of execution instantiates two additional objects with their own thread of execution. Each thread may or may not be active at any given time, but can execute in parallel with other threads in the same process when it has work to do. It is possible for one thread to execute a method in another thread synchronously, i.e., wait for the response, or it may execute asynchronously and respond to messages or

	returns from other threads as they occur. While any given thread may execute for a while and then go idle, it is also possible for all threads to be executing "simultaneously" relative to the process switching provided by the operating system. It is this ability, i.e, for a single Progress session to process multiple threads "simultaneously" that is covered by this use case.
Issues:	Forking: Multi-threading implies the ability of any process to fork off new threads, which then execute asynchronously and a mechanism by which such forked threads can return to the parent thread. It is not clear that there should be a pre-defined limit on the number of simultaneous threads since the number might be large if the threads are being used to implement a Service in a Service-Oriented Architecture, but there are probably practical limits such that the overhead from handling additional threads reduces the potential advantages of separate threads. Note, however, that many uses might include a relatively large number of threads in an idle state so the total number of threads could be correspondingly higher. The forking process should be similar to the existing mechanisms for running persistent procedures (pre-10.1) or instantiating new objects (10.1+). In the former case, it might be accomplished by a relatively simple indication that the new process is to run asynchronously and this would imply a new thread. In the latter case, it seems desirable to create a thread object and manipulate it in the fashion of other OO languages. The later mechanism should have priority over the former and it is possible that the control provided by the OO paradigm cannot be adequately duplicated for persistent procedures without excessive development work. Synchronous and Asynchronous Execution: In general, there is no benefit from executing a thread synchronously when compared with running a procedure or instantiating an object within the current thread. However, execution of internal procedures in a persistent procedure in a different thread or method calls on an object in a different thread should be capable of either synchronous or asynchronous execution. Thread Communication: Communication methods should include: • Run an internal procedure in a persistent procedure running in a separate thread asynchronously or synchronously; • Return values from a procedure execution to a callback procedure; • Terminate an asynchronous procedure when complete or continue idle with current state ready for a new procedure execution; • PUB/SUB exchanges between threads; • Notify() and NotifyAll() mechanism which merely wakes a sleeping thread so that it can check if it is ready to resume processing; • Instantiate an object to run in a separate thread and make method calls on that object as if local; • Agent/event signaling between threads. These are arranged roughly in reverse order of priority, Note that these contrast with the communication methods available between sessions where the opportunities are currently limited to: • Communication via the database or filesystem, which is unsatisfactory for many purposes; • Socket connections between processes, which requires a very rigid structure and has many issues of scalability; • Sonic, JMS, or MQSeries type MOM connections, which are highly suitable for coarse-grained, service level connections, but which carry excessive overhead for fine grained messages and which require significant deployment complexity and cost. If and when, there is some form of intersession PUB/SUB, intersession communication would be more similar to intrasession communication, but this would not obviate the need for the

fine-grained and direct communication appropriate to threads.

Database Access: In an OO design for a SQL database, one would typically expect to find a database connection object shared by the threads of a process. This suggests that it might be acceptable for a single database connection to be shared by all threads. While this might be an acceptable implementation, it seems that it would be undesirable in at least some cases where a long running database access would block use of the connection by another thread that might have a very quick database request. Good encapsulation suggests that some threads would need database access while others would not, thereby limiting the number of independent connections needed, so it would be desirable if the connection were capable of some kind of pooling in order to minimize the potential for conflicts. The number of independent connections in this pool might be a start-up parameter.

A possible simple approach could be to restrict the database connection to a single thread, but this would have the same difficulties as having the threads compete for a single database connection and would be difficult to enforce. Having them compete would be conceptually simpler, although having an explicit connection object might help this be more explicit. An explicit connection object that was not shared might have its own database connection which did not compete with other such objects in the same process. This would imply that one did not start the session with database connection information, but rather that the connection was made through this connection object. This seems the most desirable form as one could instantiate as many connections were needed for a particular session, concentrating everything in one thread and one connection when that was appropriate or using multiple connected threads when needed. The key facility missing in the current language to provide such a connection object is the requirement that one cannot use a connection at the level at which the connection is created. One would have to provide a mechanism for the object to create the connection and return, but to have that connection then be available for the thread.

Interrupt: It is desirable to provide some level of ability to interrupt a running thread without waiting for it to complete its current operation. This might take two forms. In one form one would merely post an event to the thread and leave it to the thread to notice this event and terminate gracefully. In the other form the thread would be terminated immediately as if it had encountered an error condition. While the first form provides a more graceful and safe operation, it seems likely that the second form would be required to accomplish things like aborting a long database query.

Transactions: It seems essential that transactions can propagate from one thread to another. As this presents a risk of deadlock, there will be a requirement for careful programming techniques to avoid unwanted transactions impacting large blocks of code.

Thread Shared Information: While good encapsulation will generally dictate that a thread have its own local context separate from other threads in the same process, the fact that these threads are tightly coupled, not separate processes, dictates that some amount of information should be shareable. One of the necessary pieces of shared context are parent-child relationships between threads, but it is less clear what other information should be shareable. It seems inappropriate to have any global or shared variables. Although some OO languages do support threadlocal variables, it seems preferable for such values to be obtained via accessor methods on singleton objects.

If threads are processing asynchronously, then there is an inherent difficulty associated with access to shared entities, whether direct access to data members or via accessor methods because multiple methods modifying the same set of related values without control might leave them in an inconsistent state. In Java this issue is addressed by the use of *synchronized*

	methods which effectively lock the object until the method completes, comparable to an exclusive lock on a database record. In some form, this seems like a necessary construct for any multi-threading implementation. This might alternatively take the place of named mutex locks that were obtained and checked by methods that needed to be thread-safe. It seems undesirable to share database buffers other than via synchronized methods. There is some temptation to want to share temp-tables and prodatasets in order to avoid having to pass potentially large entities as parameters. However, it is not clear that it is not preferable to simply encapsulate these entities in objects and pass the object *reference* in order to provide sharing via accessor methods on that object. This implies, of course, the necessity of using OO structures to fully exploit multi-threading, but this might be considered an acceptable limitation. Of course, some would prefer to have these capabilities available with minimum changes to existing code, but there seem to be many perils in utilizing threads without appropriate controls. Access to User Interface: While it might be best practice in most cases to only have one thread updating the UI, there is a question of how it is that this would be enforced or managed. One possibility is that it could be managed via streams if there were some slight change in current handling of default streams such that it was possible for the parent thread, which would normally have the default streams open, could effectively close those and allow one of the threads to open one or both. The alternative would seem to be only allowing the parent thread to do UI, which would be unfortunate. Comparison with Processes: Many other major contemporary languages include both the ability to run multiple processes within the same "space" (e.g., JVM) and the ability for each process to have multiple threads, where a thread is lighter weight that a full process, but does provide independent execution and local context. Two processes within the same "space" can communicate with each other using very lightweight, efficient, and highly performant mechanisms. While connections between ABL sessions using a transport such as Sonic are possible, this interface is suitable for the coarse-grained interfaces typical of a Service-Oriented Architecture, not for the fine-grained interface appropriate to objects interacting within a non-remote environment. This facility is more comparable to Remote Method Invocation where the processes are potentially on different computers. It is believed that PSC has indicated an understanding of the need to provide for efficient communications between sessions in a future release, so the current discussion is focused on multi-threading within a session. To some extent either feature can imitate or substitute for the other, i.e., good interprocess communication could allow multiple sessions to act in a coordinated way as if they were part of some "uber-session" or multi-threading could be used to bundle multiple functions within a single session, but both of these approaches have limits, so both capabilities are ultimately needed. For non-OO ABL, the syntax for this facility could be similar to that which currently exists for AppServer and WebService executions. For OOABL, some new syntax will be required and it is probably best based on the Agent/Event structure proposed in 4.12.10 of the Object Oriented Language Initiative document. I would expect the interruption mechanism to probably use APPLY or something from the Agent/Event and a callback structure. Communication for non-OO ABL might be through publish-subscribe. For OOABL, a stronger, type-validated construct would be desired.
Workarounds:	1) Instantiation of multiple sessions connected by some signaling mechanism such as sockets. This is far less attractive than the proposed functionality because it requires complex setup and it is rigid, not dynamic. In fact, it is a current limitation of the ABL socket system that it is single-threaded, limiting the extent to which a single server can reasonably expect to support multiple clients. At best, sockets are appropriate for coarse-grained interfaces

	among limited endpoints. 2) Use of Sonic, JMS, MQ Series, etc. to send messages between sessions. Again, this requires significant superstructure to implement and requires instantiating predefined sessions, not dynamic ones. There is also a concern about the performance overhead. I.e., it is a valuable technique, but not the same technique. It is a technique that is highly suitable for the coarse-grained interface between services, but not for the tight, fine-grained interface within a service. 3) Cross-session publish/subscribe. This would be a very valuable addition to the language, but, it is not currently available in ABL, and has the same rigidity issues as the other workarounds. Again, it is suitable for a coarse-grained interface, not a fine-grained one.
Best Practices:	In a good multi-threaded design, some of the appropriate best practices would include: 1) Two threads in the same process should not access the same table. Instead, access to the table should be encapsulated in a data access object in one thread and objects related to the table passed to other threads as needed or its properties accessed with accessor methods. 2) Only one thread in a session should access the UI. This is consistent with clean Model-View-Controller separation. 3) One of the design options available with multi-threading is to create a service-oriented architecture in which only one service "owns" the right to update any one particular table. All other services that need to know about that table would obtain a copy of what they needed from the owning service. In a great many cases they can request a read-only copy and it can be forgotten about, although one could also create a subscription for a change event on that record for as long as one cared about what was in it. When it was read with intent to update, the data access object could cache the record to provide a baseline and notice if more than one request was open on the same record at the same time. If an update is received when more than one update is pending, the other process could be notified of a change event and refresh to the changed version. This is a very elegant approach to optimistic locking which one can't do now. 4) Isolation of database access to data access objects that are shared by the threads in a process can limit the necessary scope of transactions, as long as it is acceptable to lose work not committed. There is no question that multi-threaded code is more complex to write than single-threaded and presents many opportunities to create locking issues and the like. However, many of these hazards are substantially reduced by means of best practices, just as they are in minimizing conflict between sessions. Just because there is a potential for difficulty is no reason to withhold capability.
Relates To:	Enhancement Request System entries: • 2931 is a request for multi-threading and has a status of DONE, but the solution described is simply an asynchronous call to the AppServer, i.e., two single-threaded sessions communicating with each other. While better than requiring all communication to be synchronous, this is not the same capability. • 1123 is a request for a multi-threaded client and has a status of NPCLOSED based on it being a “monumental task”, which does not really address its importance.1016 is a request for timer events and has a status of NPCLOSED, but provides no solution for non-Windows platforms. This capability is needed on server platforms. At best, even the Windows functionality is a trivially minimal implementation of the needed functionality. • 1122 is a request for interprocess communication and has a status of DONE based on the request containing a reference to TCP/IP and the availability of sockets, which was not the primary request. • 2600 is a request for intermachine PUB/SUB and has a status of DONE based on the availability of SonicMQ. 1122 was request for interprocess, not necessarily intermachine communication and, as discussed above under workarounds, is not an equivalent solution. • 2370 is a request for implicit PROCESS-EVENTS and has a status of ACTIVE. I do not

	feel this should be implicit and this addresses only a very small part of the needed functionality.