PARABON® COMPUTATION

Computing Outside the Box®

FRONTIER® GRID PLATFORM

Frontier® Software Development Kit

Frontier® API White Paper

Foundation

Frontier® is a massively scalable, grid computing platform that allows applications to easily utilize network-aggregated computational power atop many different operating systems. Specifically, Frontier draws on the otherwise unused computational capacity of relatively low-power, unreliable, high-latency nodes in the form of Internet-connected desktop computers running in parallel to create a coherent, high-power, reliable whole.

Behind this platform is the Frontier Application Programming Interface (API), the framework that makes creating, launching, monitoring, and controlling enormous compute-intensive jobs from an average desktop computer possible. Consisting of two primary components-the *Task Runtime API* and the *Client API*, the Frontier API enables developers to quickly build new applications or port existing applications to run on Frontier.

This document describes the Frontier API and presents how it can be used to develop applications to utilize the unprecedented power of Frontier. Specifically, we'll first take a look at the architecture of the Frontier platform, discuss the components of the Frontier API, examine the basic elements and design considerations that go into writing a Frontier application, and provide a technical overview of the Frontier API itself.

The Frontier Platform

The Frontier platform is made up of three main components, each of which plays a significant role in performing computational work on Frontier:

- *The Client Application*. Executed from a single computer, a *client application* is a domain specific application configured to execute compute-intensive jobs via communication with a Frontier server. This communication occurs within the context of a session, during which jobs can be launched, monitored, and terminated.
- *The Frontier Compute Engine*. The Frontier compute engine is a desktop application that

utilizes the spare computational power of an Internet-connected machine to process small units of computational work called *tasks* during idle time. Individuals who install and run the Frontier compute engine to contribute their computers' spare power to a Frontier grid are called *providers*.

- *The Frontier Server*. This is the central hub of the Frontier platform, which communicates with both the client application and multiple Frontier compute engines. It coordinates the scheduling and distribution of tasks; maintains records identifying all provider nodes, client sessions, and tasks; and ensures the platform's consistency and reliability.

Frontier Basics

Jobs and Tasks

Computational work to be performed on Frontier is grouped into a single, relatively isolated unit called a *job*. Within a job, work is divided into an arbitrary set of individual *tasks*, with each task being executed independently on a single node. Each job is defined by a set of elements (which we'll describe in the next section) and a set of tasks, while each task is defined by a set of elements it contains, a set of elements it requires, a list of parameters, and an entry point in the form of a Java class name.

Tasks running on Frontier have the following characteristics, each of which will be discussed in more detail later within this document:

- Tasks cannot communicate with other running tasks.
- All communication *to* a task takes place at the instantiation of the task according to its specification and associated elements.
- All communication *from* a task (e.g., the reporting of results) occurs in the form of status reports and, implicitly, checkpoints.
- Tasks can employ job-level and global elements when correctly referenced.

Although tasks within a job tend to be homogeneous and launched simultaneously, this behavior is not mandated. Tasks can be completely unrelated and launched at different times over the lifetime of a job. Once a job or task is submitted to the Frontier server, it remains resident until explicitly removed. For a task, this includes all results, regardless of whether the task has yet to run, is running, or is completed. For a job, this includes all job elements and tasks.

Data and Executable Elements

An *element* is the mechanism used to efficiently transport relatively large chunks of binary data required to execute a task. Data and executable elements may be associated with a single job or task.

They are sent from a client application to the server and directed to computational nodes as required before the execution of a task is initiated. A task may refer to elements that exist either as part of that task or as part of the job that contains it. Any other reference will not resolve correctly and will result in a malformed task.

Executable elements provide the Java bytecode instructions necessary to run a task. Except for the runtime environment, all classes required to run a given task must exist in one of the executable elements explicitly referenced in the task specification. Each executable element is formatted as a Java jar file and is made available by the engine to the Java Virtual Machine (JVM) in which the task is executed.

Data elements are black-box chunks of data that are referred to in a task's parameter list, and hence are implicitly required for running a task. All data elements thus referenced are sent to a compute engine before a task's execution begins and are made available as streams to the running task via requests to the task context. A task can handle these streams of data however it deems appropriate.

Task Definition

A task is defined by the job of which it is a part, the elements it contains, a list of executable elements it requires, a list of parameters, and an 'entry-point' class name. The latter two are described in more detail below.

The *parameter list* is the most important component of a task definition. It is possible-though not necessary-for a job to contain a set of tasks that differ only by the contents of their parameter lists. Put simply, a parameter list is a set of name <-> value pairs. Each name is a short, case-sensitive string composed of alphanumeric characters and underscores, and each value is one of a set of the following predefined types of parameter values: boolean, integer, scalar, string, date, data element reference, binary data, array of parameter values, or structure. As a structure is defined as a set of name <-> value pairs with the same form as the parameter list itself, parameters can actually contain entire hierarchies of information.

The *entry-point class name* specifies the name of a Java class that is included in the executable elements referenced by a task. This class must be public and must implement the `Task` interface as described in The Task Runtime API section.

Task Status

Task status reports are the primary method tasks use to communicate with the client application. These reports include run mode, results or exceptions, and progress, as well as other pieces of information that the Frontier compute engine and server may include such as computational work performed, etc. Run mode can include unstarted, running, and complete, as well as the more exceptional modes of paused and aborted. Results can be arbitrary and are generated by the task using the same name <-&gt value mapping mechanism as employed for task parameters. Exceptions, composed of codes and a textual description, are generated either for tasks that complete by throwing exceptions or that are 'malformed' and cannot be started, or for tasks that cannot be run on available engines due to resource limitations. Since the former type of exception would occur no matter where a task is run, it is considered the natural result of running a task, and the task generating the exception is marked complete. The latter set of exceptions cause the task to be marked aborted.

Tasks announce results and progress to the engine at arbitrary intervals-generally, whenever it is convenient for the task to do so. After each report, the Frontier compute engine may save and/or report results to the Frontier server to be persisted and routed to applicable client sessions. The last such status report before normal termination of a task is considered the final result and is consistently sent back to the server and maintained as such. Each new report of results replaces any and all previous reports.

Progress information may also optionally be attached to results, reported alone, or attached to checkpoints (described in the next section). Progress is a strictly monotonically nondecreasing, positive scalar over the lifetime of a task (including across checkpoint restarts). If reported, at any given stage of the execution of a task, a task's 'progress' is considered the last progress reported. Any new progress reported must be greater than or equal to the last.

Checkpoints

The tasks comprising a job may not always run from start to finish in a single session. The most common reason for this is temporary interruption on the provider node, such as when a provider moves his mouse when a task is only halfway complete. If such an event occurred, all the work that had been done prior to the mouse move would simply be discarded, and the task would have to start over from the beginning when the machine was once again idle. In fact, long running tasks would possibly *never* complete-for example, if a task ran for more than a day and a provider checked her mail every morning, the task would never have a chance to run from start to finish. A better approach would be to start the task from where it left off. Another, albeit less frequent, situation in which a task had to be stopped in the middle would occur if the Frontier server reassigned it to another node, for instance, from a slow machine to a faster one.

In these situations, before a task is stopped, a snapshot of its state must be saved, from which it can be restarted. Frontier uses a mechanism called *checkpoints* to achieve this capture of a task's state. Basically, a task takes a snapshot of *itself* whenever convenient, packages it up, and ships it off to the engine to be saved to disk. A task can then be stopped quickly and easily at a moment's notice, and only the work done since the last time it logged a checkpoint will be lost. The engine can use the most recent checkpoint to restart a task when it's allowed to once again start doing work, or send it to the server when requested. This mechanism is easy for tasks to use, since tasks themselves decide when they can most easily package up their state. Its real beauty, though, is that it even works when a task is shut down abruptly: the engine can stop computation *fast* when interrupted by the provider, and the mechanism even works when the computer is unexpectedly rebooted.

Checkpoints themselves are very similar in format to the set of name <-> value pairs used for task parameters. In fact, what a task actually reports as a checkpoint is simply a set of parameters that should be used in place of the original parameters to start a new task from something close to the task's current state. So, when a task is started, the parameters it is given may be either the initial parameters specified by the client application or some set of parameters logged as a checkpoint by an earlier execution of the same task. The task should continue executing based on these parameters, behaving as closely as possible to how it would have behaved had no interruption ever occurred.

For a trivial example of checkpoints, let's consider a toy task that sums all the integers from `N1` to `N2`. The two most obvious parameters would be `N1` and `N2`; however, we'll introduce a third parameter, `CurrentTotal`. If we wanted to sum 1 through 100, we'd set the following parameters: `N1` = 1, `N2` = 100, `CurrentTotal` = 0. The task would start looping from `N1` to `N2`, logging a checkpoint after each iteration. After the first iteration, the checkpoint might look like `N1` = 2, `N2` = 100, `CurrentTotal` = 1. After the second iteration, `N1` = 3, `N2` = 100, `CurrentTotal` = 3. After the third, `N1` = 4, `N2` = 100, `CurrentTotal` = 6. Let's say the power goes off during the fourth iteration. When the machine comes back up, the task will be restarted using this last checkpoint instead of the initial parameters. Conceptually, it will now compute the sum of the integers from 4 to 100, adding 6 to the result. Hence, the answer sent back will be the same as would have been reported had the task never been interrupted, without having to re-compute the sum from 1 through 3.

Design Considerations

When developing applications to take advantage of the Frontier platform, one must take into account several considerations that would not necessarily be encountered when dealing with more traditional, localized platforms. This section explains several of these issues and discusses how they affect application design and implementation.

Use of Java for Task Executable Code

The security model for an Internet-based grid computing platform requires that neither users nor providers be considered trusted entities. Unlike most local systems, users do not own or directly control the computational nodes that comprise a Frontier grid. Because Frontier allows arbitrary tasks to be executed using resources owned and operated by independent providers, engines must be able to execute unreviewed, third-party user code on demand, without risk of violating the security and integrity of a provider's computer, data, and privacy. Writing task executable code in the Java programming language ensures that this objective is achieved.

Frontier supports native tasks (written, for example, in C/C++ or Fortran), however, Frontier's security policy limits their execution to nodes explicitly designated for such. Users can configure nodes under their authority into a so-called *virtual private grid* for exclusive execution of their native tasks. Alternatively, special arrangements can be made with a provider of native-enabled engines. A tutorial for the creation of native tasks is provided in a separate document.

Java was specifically chosen over other programming languages for its inherent security features, namely, its JVM technology, which provides a 'sandbox' inside which an engine can securely process tasks on a provider's computer. The JVM is arguably the most flexible, robust, and well-established modern sandbox technology. Its sandbox is given a restrictive security policy that prohibits, among other things, direct network access, disk access, and access to native methods. The only route through which tasks can access the outside world is through interfaces specifically designed for this purpose. Tasks can communicate only directly with the Frontier compute engine and back to the Frontier server, employing disk storage within carefully established limits. This ensures that arbitrary code can be used to execute tasks on providers' machines with no manual intervention on behalf of either providers or Parabon. Thus, providers can rest assured that tasks cannot harm their machines or violate their privacy.

As tasks are run within the JVM, the code sent to engines and used to run tasks must be in the form of valid Java bytecode. Generally, Java bytecode is produced by compiling source written in the Java programming language; however, Frontier does not mandate the use of the Java language. Any language that can be compiled to Java bytecode can be used, as long as it can interface with the Frontier Runtime API. Beyond this, the client application need not be written in Java. In fact, aside from task code itself, the only portion of an application that must use Java is the piece that communicates with the Frontier server via the Client API. For example, a client application could be written almost entirely in C++, with some Java Native Interface (JNI) glue to talk to the client library, plus tasks written in Java.

Division of Jobs into Tasks

A job to be processed on Frontier must be explicitly broken up into a set of relatively small tasks. Each of these tasks is given some piece of data and is sent off to an engine to run independently and report back results. The results of these tasks are then gathered by the client application and assembled like pieces of a puzzle to form a coherent image. This requires that the amount of work a task performs be small enough both to be processed effectively given the resources (i.e., memory, disk storage, etc.) available on a provider node and to return a final result within a relatively short time-generally, a few minutes or a few hours.

Further, as nodes are inherently high-latency, the time required for a round trip between tasks running on two different nodes can be quite long-often on the order of several seconds and possibly as long as minutes or even hours. Thus, frequent communication between tasks is not feasible. Though inter-task communication may be more feasible in the future, this functionality is not supported in the current version of Frontier.

High Compute-to-Data Ratio

The individual machines that provide Frontier's computational power often have relatively low bandwidth connections to the Frontier server. Further, the central server must communicate with many of these nodes simultaneously, meaning that communications bandwidth on the server side is at a premium as well. This means that sending large amounts of data to nodes and returning large results can take significant amounts of time.

However, after a task's data has been sent to a node and before its results are sent back, the node can efficiently crunch away on a task for minutes or hours. Adding all this together, we see that tasks run most efficiently if they have large amounts of computation to perform and relatively small pieces of data required and results to report. This is known as a high 'compute-to-data ratio'.

For comparison, let's consider two extreme cases. In a best-case scenario, a task with a very high inherent compute-to-data ratio would be compute-limited and would tend to scale and run as well on

Frontier as it would on a traditional cluster of machines. Thus, it would use Frontier's power at an efficiency approaching 100 percent. On the other hand, in a worst-case scenario, a task with a very low compute-to-data ratio would be bandwidth-limited and would spend most of its time transferring data back and forth to and from the server. Such a task could, in fact, take longer to complete on Frontier than on a single machine.

The Launch-and-Listen Paradigm

For most applications, the lifetime of the actual computation maps directly to a subset of the lifetime of an application. That is, computation begins sometime after the application starts, ends sometime before the application exits, and runs only while the application is active. Frontier follows a significantly different paradigm. Applications utilizing Frontier tend to use a two-stage methodology: launching and listening.

Both stages can occur within a single instance of an application, two or more separate invocations of the application, or even via two or more completely separate applications. When launching a job, tasks are created and sent to the server for processing. Listening involves gathering results and status updates or removing tasks from Frontier. Launching and listening can intermingle in a single session or occur over the course of several sessions with the server. The precise paradigms for establishing, observing, and terminating jobs are described in The Client API.

Unpredictable Task Execution

Nodes in Frontier are by nature unreliable, in terms of whether or not they can successfully complete a given task within available resources, what their effective power over the course of a given task is, and whether they still exist and can communicate with the server by the time they have completed a task. While the Frontier server can help mitigate the first and completely hide the last by reassigning tasks, task execution time may become even more unpredictable. As a result, the expected time to completion for a given task follows a complex probabilistic distribution determined by a number of factors.

The most significant of these factors is the actual amount of computational work required to complete a given task. Although Frontier can even help mitigate this through techniques such as redundancy and selective node assignment, applications that take this distribution into account implicitly tend to behave most satisfactorily. For instance, applications that compute intermediate results can give hints as to how well a computation is performing and quickly provide feedback as to the expected final results. Some algorithms-for instance, Monte Carlo analysis-even behave quite well in the presence of only a random subset of task results, with the only penalty being additional variance in the 'partial' result.

Erroneous Results

The Frontier server can employ a number of safeguards to protect against compute engines returning invalid results, either accidentally or maliciously, including full task redundancy and result comparison. However, it is possible that an invalid result could go undetected. Hence, it is considered good practice for client applications to validate results whenever possible via internal consistency checks, discarding of outliers, etc.

Intellectual Property Protection

Often, the parameters, data, or executable elements of a task may represent a user's intellectual property. To protect this sensitive information, Frontier employs a number of mechanisms, including:

- Strong encryption of data sent between the server, compute engines, and client applications.
- Removal of all information external to actual vital task contents that might identify a user to a provider.
- Strong obfuscation of task elements and specifications on the provider disk.

Moreover, a given task-even if cracked-represents only a tiny fraction of an entire job: one piece of a large puzzle, as it were. To provide even more protection, users can implement additional safeguards within their applications beyond those Frontier provides, such as obfuscating executable bytecode, excluding intellectual property and identifying marks from task parameters and interface, and obfuscating data and results.

The Task Runtime API

The Task Runtime API is the portion of the Frontier API used to create tasks to run on a provider node. While running on a provider node, tasks are restricted to a Java sandbox with a security policy that denies access to such items as the network, the disk, and the graphical display and other I/O. The only communication tasks are permitted to make with the outside world is via the Task Runtime API. Through this API, task execution is initiated, task parameters are set, data elements are accessed, status and results are reported, and checkpoints are logged.

This section describes the role the Task Runtime API serves in the execution of a task, outlines its various components and interfaces, and explains how to write tasks that use the API correctly and efficiently to take full advantage of the Frontier platform. This includes what the format and contents of task parameters, results, and checkpoints are, as well as how to use the `Task` and `TaskContext` interfaces to access and manipulate them.

The Task Lifecycle

Tasks undergo a relatively complex lifecycle that can be described from two perspectives: that of the Frontier user and that of the provider. This section discusses these viewpoints.

Frontier User Perspective

A Frontier user initiates tasks via the Client API (which is described in more detail in The Client API section). Tasks are submitted to the Frontier server for execution, after which client applications can be used to observe status and obtain the results of tasks. In the meantime, however, several activities occur behind the scenes.

After a task specification and all elements required to run that task are submitted to the Frontier server, the task enters the queue for scheduling on one or more provider boxes. Once scheduled, the task specification and required elements are sent to the target provider. Subsequently-generally when the provider machine becomes idle-actual task execution occurs.

During the time that the task resides on the provider node, the Frontier compute engine periodically sends status reports, which the server routes back to the client sessions until the task is complete. However, the Frontier server may schedule the task redundantly on several nodes or 'migrate' it from one node to another. This may be done, for example, when a faster node is needed to meet deadlines or when one that has more RAM must be used to overcome resource limitations reported by the original node. To accomplish this, the server obtains a checkpoint from the node currently executing the task and sends that checkpoint as a task specification to another node. All of this is invisible to the client application, which merely observes status. When a task is completed, its final results persist until the user explicitly removes the task.

Provider Perspective

Task execution on a provider node consists of several phases. At any given time, the state of a non-running task is given entirely by its specification. Initially, this is exactly equal to the original specification that the Frontier server provided. When a task is to be executed, its primary class (as given in the specification), which must implement the `com.parabon.runtime.Task` interface, will be instantiated and given a 'context' object (implements the `com.parabon.runtime.TaskContext` interface) through which to communicate. Next, parameters are set on the task object via mutator methods named according to conventions described in The Task Interface. Finally, the task's `run()` method is called.

The `run()` method is executed either until complete, until an exception is thrown, or until the engine requests that the task stop. While running, a task may request data elements named in the original parameter list from the task's context. When convenient, a task may also report status and log checkpoints, which are dealt with or ignored at the engine's discretion except for the last status reported, which is always saved and reported back to the server. Compute engines save some checkpoints by replacing the task's specification with the contents of the checkpoint.

If the `run()` method exits normally, the last reported status is tagged as the final result of the task and sent back to the server. If the engine must cease task execution, it may simply terminate execution or first call the `stop()` method on the task instance. The latter method gives the task a short amount of time during which it may optionally report a last-minute status and/or log one last checkpoint, and then exit the `run()` method by throwing a `com.parabon.runtime.TaskStoppedException`.

After stopping a task, an engine may later attempt to restart it from something close to its state before termination by repeating the entire process of task execution, using the last reported checkpoint as the task specification rather than that originally sent from the server.

The Task Interface

A task's primary class, which is instantiated when the task is to be run, must implement the `com.parabon.runtime.Task` interface. The explicit portions of this interface include the `run()` and `stop()` methods. Other important, implicit portions of this interface, however, come into play even before the `run()` and `stop()` methods. These include the constructor and parameter mutator methods.

Instantiation

The constructor requirements are straightforward. A public constructor that takes an instance of `com.parabon.runtime.TaskContext` as its single argument must exist. This context should be saved, as it is the primary means of communication with the outside world and is used to access data elements, report status, and log checkpoints.

Parameters

After a task is instantiated, its parameters are set via carefully named mutator methods, a mechanism similar to that used by, for example, JavaBeans. Client applications can specify arbitrarily named parameters, each of which is associated with a value representing one of a set of predefined types of 'parameter values' (detailed below). Each parameter value type has a set of mappings to more primitive types.

For each parameter given for a task with a name `xxx` and type `Y`, a public accessor should exist in the primary task class with a name `setXxx` that takes a single argument of type `Z`, where `Z` is one of the types to which `Y` maps. For each parameter, such a mutator will be searched for, the corresponding value mapped to the type expected by the mutator, and the mutator invoked with the resulting mapped value. If several mutators fitting the template exist, the mutator having an argument type with the best mapping is picked. Mappings from type `Y` to type `Z` include, in order from best to worst, `Y` , `Y`'s type-specific primitive type mappings (given below), and finally `ParameterValue`. Note that ownership of the argument to a mutator is not transferred to the task, and no guarantee is made as to its integrity after the mutator completes. That is, mutators must make copies of parameter arguments and not try to keep references to the actual values passed in. The exceptions, of course, are immutable primitive types (e.g., `int` or `String`).

Let's consider an example in which a parameter is `"numIterations"` with a value of type 'integer' and a corresponding mutator method in the primary task class with the signature "`public void setNumIterations(short x)`". The engine would map the value of the `"numIterations"` to a `short` primitive and invoke this method with the resulting value as an argument. If a method with the same name that took an argument of type `int` or `java.lang.Long` existed, the engine would try to call that method instead. On the other hand, a similar method that took an argument of type `byte` or `com.parabon.common.ParameterValue` would not be used, since a mapping to `short` takes precedence over these types for integer parameter values.

The parameter value types, the primary Java types used to represent them, and their primitive type mappings include the following:

- *boolean*, represented by `com.parabon.common.BooleanParameterValue`, which maps to `boolean` and `java.lang.Boolean`
- *integer*, represented by `com.parabon.common.IntegerParameterValue`, which maps to `long`, `java.lang.Long`, `int`, `java.lang.Integer`, `short`, `java.lang.Short` , `byte`, and `java.lang.Byte`
- *scalar* (floating point), represented by `com.parabon.common.ScalarParameterValue`, which maps to `double`, `java.lang.Double`, `float`, and `java.lang.Float`
- *string*, represented by `com.parabon.common.StringParameterValue`, which maps to `java.lang.String`
- *date*, represented by `com.parabon.common.DateParameterValue`, which maps to `java.util.Date`
- *data element reference*, represented by `com.parabon.common.DataReferenceParameterValue`, which maps to `com.parabon.common.DataElementID`
- *binary data*, represented by `com.parabon.common.BinaryParameterValue`, which maps to `byte[]`
- *array of parameter values*, represented by `com.parabon.common.ArrayParameterValue`
- *structure* (set of name <-> parameter value pairs), represented by `com.parabon.common.NamedParameterMap`

Running and Stopping

Once the primary task class has been instantiated and all of its parameter values set via mutator methods, the task will be started via the `run()` method. This method is guaranteed to be called at most once for any given instance of the task class. Subsequent task invocations via checkpoints occur using new instances of the task class.

While `run()` is executing, the `stop()` method may be called to request that a task stop 'gracefully'. This gives the task a chance to quickly post one last status and/or log one last checkpoint, after which the `run()` method should throw a `com.parabon.runtime.TaskStoppedException`. the Frontier compute engine may also decide to stop the task more abruptly, either without calling `stop()` or, if the task does not exit gracefully, within some arbitrary timeout after `stop()` is called.

The `run()` method can exit one of four ways:

- `run()` can terminate normally, after which the task is considered complete and the last status reported is marked as its final result and sent back to the server.
- A `com.parabon.runtime.TaskStoppedException` can be thrown, indicating a graceful exit from an incomplete task after the `stop()` method has been invoked. Throwing this exception

without `stop()` having been invoked is an error and will result in a task being marked complete with an exceptional result. On the other hand, gracefully terminating an incomplete task after `stop()` has been invoked but without throwing a `com.parabon.runtime.TaskStoppedException` will be considered successful completion despite the stop request, and no attempt will be made to restart the task from a checkpoint at a later time.

- A set of specific runtime exceptions represents such conditions as resource limitations and spurious engine errors. These include any number of exceptions that are defined by the Frontier runtime environment (not the task itself) and that extend the `com.parabon.runtime.TaskRuntimeException` class, as well as other throwables including `java.lang.OutOfMemoryError`. If such an exception is thrown, the task is considered incomplete and its mode marked as aborted. The Frontier server may attempt to restart the task on another compute engine from the original specification or the last checkpoint, or may propagate this condition back to the client and cease further attempts to run the task.
- When any other exception is thrown by the task from the `run()` method, it is considered the 'natural' result of the task. Hence, the task is marked as complete, and the exception is propagated back to the server and eventually to the client application.

The Task Context

When a task object is instantiated, its constructor is passed an instance of `com.parabon.runtime.TaskContext`. This object is used for communication with the outside world. Specifically, it can be used to obtain data elements, post results, report progress, and log checkpoints.

Accessing Data Elements

Data elements required during the running of a task can be obtained via a request to the context's `getDataElement()` method, giving the ID of the requested data element. As `DataElementID` instances can only be obtained as task parameters, required data elements must be specified somewhere in a task's parameters. Note that although it is possible to obtain IDs through other means (i.e., directly implementing `com.parabon.common.DataElementID` within a task), a task should never attempt to do so. Frontier does not guarantee that data elements identified through such means will be available to a running task.

A requested data element is returned in the form of a `com.parabon.runtime.DataElement` object. This object can then be queried to obtain a 'black-box' `InputStream` as well as other properties of the data element, which may or may not be specified (for instance, total data length). The actual data can then be read from the obtained stream. Each time the `getStream()` method is called on a data element, a new stream is returned, which will return data starting from the beginning of the data element. Both `TaskContext.getDataElement()` and `DataElement.getStream()` can be called as many times as desired over the course of a task's execution, at the possible price of additional resource usage.

Posting Results and Reporting Status

Whenever convenient, a task can post results via its context object. Each new result posted replaces any and all previously reported results. Thus, at any given time, the 'current results' of a task are those last reported by the task. Earlier reported results may be thrown away and possibly never reported back to the server or the client application. The last results posted are considered to be the final results of a task.

Results themselves are the primary component of status and are represented by a set of name <-> value pairs. This is similar to the format of task parameters, encapsulated within a single `com.parabon.common.NamedParameterMap` instance owned by the task. When results are posted, this structure is copied and possibly persisted and/or reported back to the server, and from there, possibly to the client application. To put it simply, some reported intermediate results may make it back to the client application, but in general only the transmission of the final result should be relied upon.

The frequency at which results should be posted depends strongly on both the nature of the task being run and the size of the results. An upper limit on reporting frequency can generally be set naturally, for instance, at the beginning of a central outer loop of a task when results are well defined and can easily be gathered. However, actual frequency should generally be much less than the upper bound. Posting results too often will slow down the running of a task, as some overhead related to the size of the results is incurred on every report. On the other hand, posting results too infrequently generally reduces the amount of status information available to a client application session during the course of task execution. In general, reasonably large results should be reported only once, at the end of a task, as the bandwidth cost incurred in transferring large intermediate results is generally greater than any benefit gained. Instead, intermediate results should generally be only a brief summary.

Logging Checkpoints

Checkpoints consist of modifications to a task specification in the form of a new set of parameters and optionally a new 'primary class'. Each checkpoint, when logged, replaces the relevant portions of either the previous checkpoint or the original task specification. In particular, checkpoint parameters do not augment previous parameter sets, but replace them in their entirety. Similar to results, checkpoint parameter sets are represented by a single `com.parabon.common.NamedParameterMap` instance owned by the task.

Similar to status, checkpoints should be reported whenever a task is in a sufficiently coherent state that its state can be summarized into a checkpoint and reported. The frequency tradeoff for checkpoints is somewhat more subtle, but even more important than that of status frequency. Although checkpoints tend not to be reported back to the server on a regular basis and so can be somewhat larger than results, they do require disk space on a provider machine and the size-related overhead incurred in logging. More frequent checkpoints result in more overhead-computational time lost with no benefit if a checkpoint is never needed for a restart. However, less frequent checkpoints result in more computation lost whenever a task is terminated and restarted.

In an extreme case, imagine a provider whose machine is only idle for thirty-minute spans and a task that only checkpoints every hour: this task will never make any progress. Such pathological cases may be somewhat rare, but nonetheless, a task that only reports checkpoints every hour will tend to lose a half-hour's worth of computation on average at every termination and restart. This problem can be overcome for many cases by a task that checkpoints when a stop is requested, but such functionality tends to be arbitrarily difficult to implement for many tasks. More significantly, it is not guaranteed that such an opportunity will always-or ever-be made available to a task, and so should not be relied upon.

Reporting Progress

Progress is a single piece of data that represents the amount of computation a given task has completed thus far and can be reported arbitrarily often by a task. The only restrictions are that it be a positive, nondecreasing (ideally strictly increasing) scalar over the lifetime of a task. That is, each time progress is reported, the value given should be no less than-and ideally greater than-the value given at the previous progress report, if any. Note that the lifetime of a task can span several checkpoint/restart instances of a task. That is, any progress reported after restarting from a checkpoint must be no less than that associated with that checkpoint when it was created by a previous instance of the same task. A task does not have to report progress, but it can aid in efficient scheduling and other task management issues and can provide a useful hint to the client application as well.

The current progress is defined as the value of the most recently reported progress and exists for a task at any point after it is first reported. The current progress is associated with each task status and checkpoint. Progress can be reported by itself or in conjunction with a result or checkpoint, which is equivalent to reporting progress by itself followed directly by posting a result or logging a checkpoint without an associated progress.

As progress is much cheaper to report to the engine than results or checkpoints, it makes sense to report it as often as is convenient, without the efficiency considerations involved in reporting results of logging checkpoints. Most progress reports will not be propagated back to the server, however, but merely saved internally by the engine runtime.

The Client API

The Client API is the portion of the Frontier API with which client applications deal directly when creating, monitoring, and controlling jobs and tasks. The Client API is implemented by the client library and provides a set of functionality used by a client application to communicate with the Frontier server. The functionality it encapsulates falls into several different categories:

- Performing communications with the Frontier server
- Specifying and submitting jobs and tasks
- Listening to and observing jobs and tasks
- Controlling and removing jobs and tasks

- Running tasks locally without submitting to the server

This section describes the modes in which the Client API can be operated, how interaction with the Frontier environment occurs, job and task attributes, and how jobs and tasks are launched, monitored, and controlled.

Remote vs. Local Mode

The client library can be operated in either remote or local mode. The former manages a session with the Frontier server, relaying requests and responses back and forth between server and client application via the Frontier messaging protocol. This is the mode that allows a client application to actually command the resources of Frontier. Local mode, on the other hand, provides an efficient virtual session using only resources local to the client running the application, which is important for running applications using the Frontier API transparently without bringing Frontier itself into the equation. This can be especially useful for debugging client applications.

The choice of remote or local is made when a session is created via a `SessionManager`, the primary interface to the client library. Local sessions execute tasks exclusively on the the client application's host computer, making it useful for small jobs or debugging applications. Remote sessions execute tasks across a Frontier grid. The fact that the interface to both modes is identical, plus the ease with which modes can be toggled, means that a single application can be written to the Frontier API and yet only bring the power of a Frontier grid to bear when necessary. This section is written with remote job execution in mind, however, nuances of local mode execution will be mentioned where noteworthy.

SessionManager

An interaction with the Frontier environment, be it the actual Frontier platform or the local execution of tasks, takes place within the context of a *session*. Sessions are created and managed by an instance of `com.parabon.client.SessionManager`. Thus, this class becomes the entry point into the Client portion of the Frontier API. The `SessionManager` is responsible for acting as a proxy to manage the resources of the Frontier platform and the set of jobs a user is running. Two implementations of `SessionManager` exist: `com.parabon.client.RemoteSessionManager` and `com.parabon.client.LocalSessionManager`, which operate the client library in either remote or local mode, respectively.

The interface to both implementations of `SessionManager` is identical, but the behavior may differ in some cases. For example, sessions can be destroyed via the `destroy()` method. Destroying a session in remote mode means that the client library will disconnect from the Frontier server while submitted jobs and tasks continue to run. Destroying a session in local mode means the client library will wait until all tasks have completed before control is returned to the application.

Job and Task Attributes

As jobs and tasks are created via the Client API, it is important to have a means of later identifying them. Although the client library uses internally generated identifiers to track jobs and tasks, these identifiers are not exposed through the API. Instead, a more flexible scheme is provided to allow client applications to identify jobs and tasks. At creation time, a set of attributes (i.e., name <-> value pairs in a format similar to those used for task parameters and results) can be assigned to a job or task. These attributes are then propagated through the library and Frontier server. Attributes can encode information such as simple unique identifiers, the username of the person who initiated a job, the launch date of a task, a set of email addresses to send results to, or the name of a file stored locally to keep track or more complex job information and statistics. The important point is simply that attributes are user-defined, immutable, and simply passed through the Frontier system as identifying information to be later used by the client application. Note that any assigned attributes are not sent to provider nodes actually running the tasks.

In addition to observing the attributes of any given task or job, the client library provides mechanisms to select jobs and tasks based on a simple attribute template. This facility is useful when, for instance, a client application wishes to observe or listen to a subset of tasks in a large job without incurring the memory and communications overhead of obtaining information about all the tasks in that job.

Creation of Jobs and Tasks

Jobs

Jobs are created via the `SessionManager.createJob()` method, which is passed an instance of `com.parabon.common.NamedParameterMap` specifying the attributes to be associated with the new job. The job itself is represented by a `com.parabon.client.Job` instance. After creating a job-in effect an empty repository-a client application will generally 'fill' it with job-level elements (via the `addDataElement()` and `addExecutableElement()` methods) and tasks, as described below. It is important to note that once a task or element is added to a job, ownership of the object used to specify that task or element passes to the job. Subsequent modification of that object by the client application will result in undefined behavior.

Tasks

Once a job has been created, tasks can be added to it. This is done via the `Job.addTask()` method. This method takes as arguments a `com.parabon.client.TaskSpec` instance, which describes the specification of the task to be run, and a `NamedParameterMap`, which specifies task attributes. The method will return a `com.parabon.client.TaskProxy` instance, which can be subsequently used by the client application to control the task. Note that ownership of the `TaskSpec` is transferred to the client library. Any attempts to modify a `TaskSpec` instance after using it to create a new task will result in undefined behavior, and further, the contents of the `TaskSpec` itself are not guaranteed to remain unchanged by the client library.

A task specification, as described by an instance of `com.parabon.client.TaskSpec`, consists of four primary pieces:

- *A set of elements*. These executable and data elements are defined and behave similarly to those of job-level elements. The difference is that they are task-scope, meaning they cannot be accessed outside of the task to which they belong, allowing Frontier to perform more intelligent cleanup. For instance, once a task has completed, all of its elements may be freed. Any elements that can logically be used by several tasks should be made job-level or higher rather than being added to each task individually. This will reduce memory and communication overhead and allow caching of elements on engines across tasks.

- *A list of required executable elements*. Any executable element that may be required for running of a given task must be referenced explicitly, be it task-level, job-level, or higher. Any class referenced by a task that is not part of the predefined runtime environment (that is, standard Java classes and those in the `com.parabon.runtime` package) must be included in one of the executable elements referenced explicitly in this list.

- *A task class name*. This specifies the 'runnable class' that is used as an entry-point into the task. This class must be included in one of the executable elements required for this task and must implement both the explicit and implicit portions of the `Task` interface.

- *A parameter list*. The list of parameters for a task can be specified either in its entirety via the `setParams()` method, which takes a `NamedParameterMap` instance as an argument, or piecemeal via the `putParam()` methods. Note that both methods copy their arguments when necessary and so do not transfer ownership of referenced objects.

Elements

Data and executable elements are represented by instances of the interfaces `com.parabon.runtime.DataElement` and `com.parabon.runtime.ExecutableElement`. For the purpose of adding an element to a job or task spec, a client application must either use an existing implementation of these interfaces (e.g., `com.parabon.client.JarFileExecutableElement`) or implement them itself. The interfaces are relatively straightforward. Most importantly, implementations must provide a `getStream()` method, which returns a stream representing the contents of the element itself. This method may be invoked multiple times, and each time must return an independent stream that produces data starting from the beginning of the element. The `getStreamLength()` method must also be implemented, either to return the length of the element if known (which can greatly increase the efficiency of some portions of the client library), or `-1` otherwise. In addition, executable elements must provide other pieces of information about the contents of the executable element, namely language and packing, for which only `"Java"` and `"jar"`, respectively, are currently supported.

It is important to note that the `DataElement` instances passed to tasks via a request to

`TaskContext.getDataElement()` are not necessarily the same objects or even classes used to create the data element, and so no such assumptions about the implementation of `DataElement` or its underlying streams should be made within tasks. In particular, although in local mode the same instances are often passed through from client application to task for reasons of efficiency, this is not the case in remote mode.

Submitting Data to the Frontier Server

In remote mode, as soon as a manager is connected to the Frontier server, the manager will start to submit candidate jobs and applicable portions of their contents. Such information will continue to be submitted as it is created and becomes a candidate for submission. When the manager is destroyed, it will continue attempting to submit all information that became a candidate for submission any time before its destruction. This means that `SessionManager.destroy()` will block until pending submissions have completed successfully.

Three types of data can be submitted to the server: jobs, tasks, and elements. A job becomes a candidate for submission as soon as it is started. Tasks become candidates as soon as they are started. Elements become candidates as soon as at least one task that references them has been started. This means that a task that was created but never started or an element that was never referenced will never be sent to a server and will be lost when a session ends (that is, when the manager is destroyed).

Monitoring and Controlling

Once a job is created and its tasks started, the client application needs to listen to results and status and, when a job or task is complete, remove it. There are three ways to perform these functions:

- Via the original instance of a client application.
- By a later instance of the client application when running in remote mode as described in Reestablishing.
- By an entirely different application.

These operations are performed for a job via the `Job` instance representing the job, or for a task via the corresponding `TaskProxy` instance supplied by a job or event.

Run Mode

A task's run mode represents which phase of execution it is in at a particular time as one of a small set of enumerated values, specified in `com.parabon.common.TaskRunMode`. A task's last known run mode can be obtained through the task's proxy as embodied by a corresponding `TaskProxy` instance obtained through a job or event. The mode a task was in when it generated a particular event is specified in the event itself, as described in Listening to Events. Possible run modes include the following:

- *Unknown* means not that a task is in an exceptional state but rather that the queried component has no run mode information about the task-for instance, a task proxy in a reestablished session before any messages regarding task status have been obtained from the server.

- *Unstarted* means that execution of a task on an engine has not yet begun.

- *Running* means that a task is currently running on one or more engines.

- *Paused* means that a task has begun executing and is ready to continue, but is currently in a non-active state because of some external condition (e.g., waiting for a provider machine to become idle).

- *Complete* means that a task has successfully completed execution. Note that despite being reported as complete, a task may have been malformed, threw an exception, etc. Even so, this condition is seen as the natural result of the task because of the specification of the task itself rather than an external error or resource limitation.

- *Stopped* means that a task is ready to run, but is in an inactive state because a user has requested that this task be stopped. A stopped task will remain in this state until removed or explicitly restarted by a user via `TaskProxy.start()`.

- *Aborted* means that execution of a task has been terminated because of resource limitations (such as provider memory) or other external exceptions.

Listening to Events

Both jobs and tasks allow *listeners* to be added and removed. Listeners are user-created classes that implement one of a set of listener interfaces, each of which receives via a method call one or more types of events generated by a job or task. A job listener will receive events corresponding to all the tasks in that job, while a task listener will receive only events corresponding to that particular task. Note that while all applicable job- and task-level listeners will be called for any given event, no guarantee is made about the order in which they are called.

All events generated by a task are described via a `com.parabon.client.TaskEvent` instance. This interface supplies a set of methods to query some common aspects of the task that generated an event:

- *Attributes* are always available and provide a task's identifying information as specified when it was created.

- The *`TaskProxy`* instance represents the corresponding task, which can be used to perform such functions as adding or removing listeners, removing the task itself, or query the task's state.

- *Progress*, when known, is the current progress corresponding to an event; that is, the last progress value specified by a task before generating an event. If progress is not known, it will be reported as `-1`.

- *Run mode* represents which phase of execution this task was in when this event was generated.

When a task's run mode is not unknown, the `isActive()` and `isComplete()` convenience routines use the run mode to determine whether the task was actively running (i.e., neither unstarted, complete, stopped, or aborted) or complete (i.e., complete or aborted), respectively, when this event was generated.

Five types of task event listeners are defined, each of which extends the otherwise empty `com.parabon.client.TaskEventListener` interface. A listener class should implement one or more of these interfaces to perform application-specific operations when an event is received. The events sent to any given listener are determined by which interfaces it implements-for instance, if the listener implements `com.parabon.client.TaskResultListener`, it will be sent result events. Each listener receives one set of events through a method that takes as a parameter an instance of a particular class derived from `TaskEvent`. Note that it is possible for a given event to correspond to two listener methods within a single listener object (e.g., a result listener and a status listener). In this case the event will be sent to both applicable methods in an unspecified order.

Also note that events passed to listeners, and any objects obtained from them with the exception of `TaskProxy` instances, are valid only for the length of the method call. For instance, results obtained from result events must be copied during the listener event method call if a client application wishes to reference them after the method returns. Also, as ownership of these objects is not transferred to the client application, any changes to event objects or their children will result in undefined behavior for the remainder of the session.

The five types of task events are:

- *Results*. A listener implementing `TaskResultListener` , an interface which consists of the `resultsPosted()` method, will be sent result events in the form of a `TaskResultEvent` instance. Results include both intermediate and final results. The two can be distinguished based on whether the task is complete or not (e.g., via a call to `event.isComplete()`). The `TaskResultEvent.getResults()` method provides access to the results themselves as reported by the task.

- *Exceptions*. A listener implementing `TaskExceptionListener`, an interface that consists of the `exceptionThrown()` method, will be sent result events in the form of a `TaskExceptionEvent` instance. When an exception is encountered during the execution of a task, it is reported via this mechanism. Note that in this context, an exception is considered the state of a task, and when reported to a client application, it generally means that this task has been marked as complete or aborted. As such, it is possible for an exception to be reported to listeners several times, each time representing the same exception rather than a new one. The details of the exception can be obtained via the event itself. These details include a code (an enumeration giving the category of exception, as specified in `com.parabon.common.TaskExceptionCode`) and a description (a string describing the exception, or for a user exception, the result of calling `toString()` on the exception).

- *Progress*. A listener implementing `TaskProgressListener` , an interface that consists of the `progressReported()` method, will be sent events in the form of a `TaskProgressEvent` instance. Such events consist of information about the last reported value of the task's progress scalar, as well as the current run mode of the task. These events are generally produced when progress or run mode change, but may occur either more or less frequently; not all such changes will result in the communication of a `TaskProgressEvent` to listeners, while some such events may be generated even when these values have not changed. This event type is similar to result and exception events in that result and exception events also contain progress information; however, progress events are both lighter weight (requiring less-often much less-server communication) and can be generated more frequently than result and exception events, reporting values with a finer resolution.

- *Checkpoints*. A listener implementing `TaskCheckpointListener`, an interface that consists of the `checkpointLogged()` method, will be sent events in the form of a `TaskCheckpointEvent` instance. The event can be queried to return a `TaskSpec` instance containing the parameters and runnable class for the checkpoint. Note that in remote mode, checkpoint events will generally not be propagated to client applications, at least in this version of Frontier.
- *Universal*. A listener implementing `UniversalTaskListener`, an interface that consists of the `eventGenerated()` method, will be sent all events in the form of a `TaskEvent`. This category of events is a superset of exception, result, progress, and checkpoint events. Use of this listener subscribes to *all* events from the server when applicable. This listener can be used in place of other listeners if an application so chooses in order to multiplex events using application-specific logic, rather than employing the simple multiplexing offered by the use of other listener types. For example, an event received by this listener can first be examined to determine if it is a result or an exception, and if so, processed accordingly.

Removing Tasks and Jobs

As soon as a job or task is complete and its results have been recorded locally by the client application-or sooner if it is no longer required-it should be removed by calling either the `Job.remove()` or `TaskProxy.remove()` method, respectively. Removal of a job includes removal of all the tasks it contains. This action will stop execution if necessary, clean up all related records and elements from the Frontier server in remote mode, and release all local objects associated with that job or task from the client library. Until a job or task has been removed, all records of its structure, contents, and results will be maintained on the Frontier server.

Releasing and Reestablishing

In several circumstances, in remote mode, the local mirror of the jobs and tasks on the Frontier server may be incomplete. Any such information that exists on the server but not locally is referred to as *released*. The process used to selectively obtain information from the server once again is known as *reestablishing*. Reestablishing is useful not only for regaining information that has been released over the course of a session, but also during new sessions, for monitoring and controlling jobs created during previous sessions. Note that everything in this section pertains to remote mode. Although some of the methods described are common to both modes, they are merely stubs in local mode.

Release Behavior

Client applications must be aware of the possibility of data being released. Specifically, this means two things. First, just because a piece of data is not made available directly-for instance, a job doesn't appear in a job list or a task has null results-doesn't necessarily mean that it does not exist. When faced with such an absence, a client application should not necessarily behave as though a job cannot be accessed or a task failed. Second, a client application should know how to reestablish such data from the server either when faced with a required bit of data being released or as a general preventative measure.

Entire jobs might be released, in which case they will not be included in job listings until the session is *reestablished* (as described in the next section). Finer-grained pieces of data can also be released. For instance, task progress might be given as `-1`, indicating progress is not known, after a task has been released and reestablished. The API reference documentation lists the behavior of particular methods when dealing with released data.

Reestablishing

Reestablishing is a crucial piece of the launch-and-listen paradigm when the listening half of the equation is in a different session than the launching portion. Specifically, reestablishing refers to obtaining information about the contents and status of released jobs and tasks, most often those created in previous sessions, and attaching to them in order to monitor status changes. This is somewhat nontrivial because of the wealth of information that can be associated with a single user's jobs. It is not generally efficient or even feasible to simply transfer all possibly relevant data to a client application upon establishing a new session. Hence, the process of reestablishing involves a series of Client API methods used to bring a new session into synchronization with the server with respect to a subset of jobs and tasks without incurring additional overhead, as well as a set of rules governing how a reestablished session behaves from a client application standpoint.

There are four primary levels of information that a client application may want to reestablish: jobs, tasks, and task contents. For instance, to find the results of a particular task, a client application must:

1. *Start a new session.* Create a `RemoteSessionManager` as usual.

2. *Find the correct job.* Reestablish the job list (`RemoteSessionManager.reestablish()`), and iterate through it (`RemoteSessionManager.getJobIterator()`), using each job's attributes to identify it, until the desired job is found.

3. *Find the correct task.* Either (1) reestablish the job's contents (`Job.reestablish()`), iterate through its task list (`Job.getTaskIterator()`), and look at the tasks' attributes, or (2) reestablish a partial set of tasks matching a pre-constructed attribute template (`Job.findRemoteTasksByAttribute()`) and iterate through this reduced list. The former method is more straightforward and somewhat more flexible, but requires transferring and storing the entire contents of the job-possibly tens of thousands of tasks, most of which may not be of interest. The latter method allows the Frontier server to cull the list of tasks transferred based on a simple attribute comparison mechanism, reducing the amount of data that must be

transferred over the wire. A hybrid process would be to use the `Job.reestablishPartial()`, which reestablishes the part of a job's task list matching a given attribute template, but doesn't attempt to create an iterator through these tasks. Rather, it merely makes them available through later calls to the `Job.getTaskIterator()` , at least until the tasks are released once again.

4. *Get the results of the task.* Either reestablish the task itself (`TaskProxy.reestablish()`) and query the desired fields via the `TaskProxy` (e.g., via `TaskProxy.getResults()`), or attach a listener to the task, which will then receive any further results sent by the task in addition to the current status, via the listener interface.

Many common operations require only a subset of these steps. For instance, to listen to the results of all tasks in a job, the job itself doesn't need to be reestablished; the application merely needs to reestablish the manager, find the relevant job, and attach a listener.

It should be noted that because of the hierarchical structure of the Client API, the fact that a `Job` instance exists, for instance, does not mean that `Job` is fully reestablished. `RemoteSessionManager.reestablish()` generally obtains only enough information to identify each of its jobs and make their attributes available, and hence attributes will be the only information accessible through the otherwise empty initial `Job` instance provided. Attributes are, in fact, the only piece of information guaranteed to always be available when given a `Job` or `TaskProxy` instance exists.

The attribute query mechanism bears further description. The template is in the form of a `NamedParameterMap`, similar to the attributes themselves. A task matches the template if and only if each of the entries in the template exist as attributes in the task and their respective values match as well. The definition of matching values is somewhat different for each type. For most types (e.g., strings, integers, etc.), the attribute and template value must match exactly. For structures, equality is defined as applying these matching criteria recursively-that is, the template entry value is itself treated as a template that the task's corresponding attribute value must match. Note that a task can contain additional attributes not present in the template and still match the template. This mechanism allows a set of simple, logical queries to be executed, depending on the attributes a client application employs. For instance, if each task corresponds to one entry in a two-dimensional table, and the row and table indices are stored as attributes for each task, a list of the tasks corresponding to a single row or column can be obtained by constructing a template with a single entry containing the desired row or column index.

Running Locally

When running in local mode, tasks are executed using local resources, within the same JVM as the client library itself is running. As noted earlier, local mode's primary raison d'etre is not a simulation of running a job on Frontier itself, but rather a low-overhead mechanism for direct, efficient execution of tasks written using the Frontier API within a client application. As a secondary consideration, this mode can be used for task debugging, with the caveat that bugs involved with the precise behavior of remote execution or that depend on the particular behavior of local mode may not be made apparent.

The serial or parallel nature of task execution can be controlled by a set of methods in `LocalSessionManager`. In particular, `setMaxRunningTasks()` can be used to set the number of tasks that the client library will attempt to run in parallel. Setting this to a large number will result in the client library attempting to execute all tasks in a job simultaneously, each in its own thread. This situation is generally far from optimal and could often result in resource and operating system-imposed limitations. At the other extreme, setting this value to `1` will result in tasks being executed in a completely serial fashion, each being started after the previous one has completed. The method `getMaxRunningTasks()` can be used to query the current value for this field, while `getNumRunningTasks()` will report the number of tasks currently being executed in parallel, each in its own thread. This number will always be less than or equal to the number of tasks that are candidates for execution (i.e., those that have been created and started but have not yet completed).

Size Considerations

The amount of data involved with a single job is often large enough to tax the resources of a client-side machine, and so it is often desirable to launch, monitor, and control a job without the burden of keeping all job-related data resident and up-to-date locally. This section contains a number of considerations that, when followed carefully, can allow jobs of arbitrary size to be created and monitored using limited local memory resources. Note that all of these considerations apply only to remote mode.

Message Queue Sizes

Incoming and outgoing message queues can often grow large very quickly. The former is often because task results can be retrieved more quickly than they can be processed, while the latter is because tasks can often be created more quickly than they can be serialized and sent over the wire to the server. These conditions can both be easily mitigated by using the `RemoteSessionManager.setMaxIncomingMessageQueueSize()` and `RemoteSessionManager.setMaxOutgoingMessageQueueSize()` methods, respectively. Both methods specify hints rather than hard limits. For instance, a batch of messages may be retrieved at once, resulting in the incoming message queue temporarily growing beyond its specified bounds. The price of using these limits is that efficiency may in some cases be reduced because of underutilization of available communications bandwidth. In addition, if an outgoing queue size limit is used, arbitrary methods in the Client API may block without warning, as they often have to enqueue messages in order to successfully complete. Thus, these messages may be forced to wait until a slot is available in the outgoing queue before returning.

Task Release

Jobs by default attempt to keep records of all tasks they contain and are aware of-both those created locally and those created in earlier sessions and reported by explicitly reestablishing the job or implicitly via status reports. As task records can be large in general (containing, among other items, last known results) and jobs can contain large numbers of tasks, it makes sense to optionally ease this restriction. This can be done by using the `Job.setAllowTaskRelease()` method. When task release is enabled, jobs may 'forget about' tasks arbitrarily in order to reduce resource requirements, as long as external references to them are not maintained via `TaskProxy` instances. Events will still be propagated reliably to listeners, and records will be maintained at least during the course of event consumption.

While this alternative behavior is powerful, the impact of using it is significant and twofold. The first issue is efficiency. As information can be forgotten about arbitrarily, it may not be available when a client application needs it. Thus, a reestablish process, which requires a messaging round trip to the server and a transfer of the resulting information, may occasionally be required to obtain information that was already known by the client library earlier in a session. How often this will occur depends on the structure of the client application and how many extra resources are actually available. In the best case, a client application will act only on information contained in events and so no reduction in efficiency will occur. In the worst case, the effective reduction of local cache could result in a 'thrashing' situation seen commonly in other caching situations (e.g., virtual memory usage) and reduce the performance of an application severely.

The second issue involves software development. Put simply, when utilizing remote mode, client applications will not be able to assume that information that was available a few milliseconds ago will still be available, unless the application itself keeps references to it. This means, for example, that even if a job was reestablished earlier in a session, the current list of tasks reported by a job may be incomplete. What this means in terms of programming complexity is entirely dependent on the structure of the client application itself. Be warned, however, that unless assumptions are carefully examined, unexpected and difficult-to-track application bugs could result.

Job Listeners

Applications that run with constant-order memory requirements must employ constant-order mechanisms, including listeners. If many tasks are each listened to explicitly-even if by the same listener object-then the client library must keep track of an explicit data structure for each task. Hence while listening to large numbers of tasks, a large amount of resources will be required. Furthermore, listening to a task may represent an implicit reference to that task, and so-although this behavior is not in any way guaranteed-the client library may keep track of all data relating to that task, including results, even if task release (as described above) is enabled.

The solution is to use a single job listener, which will receive events from all tasks in that job. The tradeoff is that if only a subset of tasks are relevant, then the listener will have to examine each event to determine whether it corresponds to a relevant task. In addition, all task events for that job will be propagated, possibly requiring additional unnecessary communication overhead if the subset of tasks being considered is relatively small. In summary, if most tasks within a job are being listened to, it is generally a good idea to listen to the entire job via a job listener rather than to each task via individual task listeners.

Referenced Structures

The referenced structure consideration is twofold. First, even if task release is enabled, tasks will not be released if references to them are maintained external to the client library, through `TaskProxy` instances, or any other implicit references to task information (e.g., attribute maps). The second consideration may be somewhat obvious, but it bears saying nonetheless. Applications should not, when it is possible to avoid, keep large, O(N) data structures-that is, structures in which some amount of information is kept for each task. Although doing so tends to be common practice, developers should keep in mind the scale of application made possible through Frontier and make every attempt to ensure that the client application itself and the resources available on local machines do not become the bottlenecks preventing the execution of arbitrarily large jobs.

Conclusion

The Frontier API provides a framework that allows arbitrarily large jobs to be reliably launched, monitored, and controlled from an average desktop computer. Using the API's two primary components, the Task Runtime API and the Client API, developers can create new or port existing applications to utilize the vast unused computational power of a Frontier grid of computers.

Among its features, the Frontier API:

- Provides a natural, straightforward, flexible framework to create truly arbitrary tasks that can be run securely and independently on unreliable third-party nodes.
- Uses a checkpoint mechanism to ensure efficient task execution across abrupt and total interruptions.
- Encapsulates all necessary communication processes between the client application, Frontier server, and Frontier compute engines.
- Provides a flexible scheme for retrieving status and results based on a traditional event listener model.
- Allows jobs to persist reliably and invisibly across sessions, including the ability for a single job to be launched, monitored, and controlled by different client applications.