

Globus Toolkit 4.2.1 Developer's Guide

Globus Toolkit 4.2.1 Developer's Guide

Introduction

You can download the [PDF version here](#)¹. Following are some docs you should be familiar with as well:

- [Installation Guide](#)
- [Quickstart](#)
- [Programming Tutorial](#)²
- All GT command line clients are listed [here](#).

¹ [gtDeveloperGuide.pdf](#)

² <http://gdp.globus.org/gt4-tutorial/>

Table of Contents

1. Best Practices for Developing in GT4	1
1. Implementing Services	1
2. Resource Properties Overview	4
1. Introduction	4
2. Security	4
3. Data Management	4
4. Information Services	6
5. Execution Management	6
3. Samples	12
I. Asynchronous Event Handling with Examples	13
4. GT 4.2.1: Asynchronous Event Handling	15
1. Examples	15
2. Event Models	15
3. Callback Library	17
4. Thread Abstraction	18
5. Asynchronous Model	19
6. Conclusion	21
7. References	21
5. Asynchronous Event Handling: Example 1	22
6. Asynchronous Event Handling: Example 2	23
7. Asynchronous Event Handling: Example 3	25
A. Globus Toolkit 4.2.1 Public Interface Guides	28
Glossary	30

List of Figures

4.1. State Diagram	20
--------------------------	----

Chapter 1. GT 4.2.1 Best Practices for Developing in GT4

This page provides some recommendations for developing with the Globus Toolkit 4.2.1 and its components.

1. Implementing Services

1.1. Dynamic resource creation and the factory pattern

OGSI defined a standard create method for creating new grid services. The follow on, the WS Resource Framework, no longer defines such a mechanism. This does not mean that the factory pattern is not valuable. Rather, it was removed because any non-trivial create method tends to be application specific, a observation that leads to the conclusion that there is little value in standardizing this operation. We recommend the use of the factory pattern for dynamic creation of resources. In particular we recommend writing a factory service that provides a way to create new resources and allows users to inspect state information about the aggregation of all resources created by the factory service, e.g. the number of resources managed. In addition to the factory service one would write a service that interacts with the resource instances created by the factory service.

1.2. Scalability, recovery and resource persistence

Writing scalable (in the number of resources) and recoverable, i.e. give it the ability to survive a server crash and restart, resources takes a bit of careful planning. There are several potential pitfalls:

- Since scalability relies upon using Java soft references it is important that resource do not keep hard references to any objects that would prevent the soft reference mechanism from working.
- When recovering resources after a container restart it is often important to re-establish the current state of the process the resource represents, e.g. if the resource is monitoring a external entity via notifications it should query for the current state of the external entity upon restart.
- Persisted resources need to be carefully written to avoid the following problem: If a service currently holds a hard reference to the resource and the resource is destroyed, i.e. the soft reference to the resource is removed from the resource home and the resource is removed from persistent storage, then the service holding the resource reference can still cause a call to the store persistence callback. Unless the store callback was written to prevent this, calling it will restore the destroyed resource.

For example: during a typical service operation, a service will lookup a resource using its resource home. At that point in time, the service holds a hard reference to the resource object. If another client of the service concurrently invokes a destroy operation on the same resource, thus removing the resource from the resource home and its state from persistent storage, the resource essentially becomes invalid. When this occurs it is important that the first service operation not invoke the resource's store method since doing so would inadvertently recreate (or resurrect) the resource.

To prevent this consider writing the resource's store and remove methods in such a way as to prevent recreating resource state that has been previously destroyed by another caller.

- Resource implementers need to be careful when storing objects created by axis as part of the web service operation invocation and containing fields resulting from a xsd:any in the corresponding schema. These complex types have references to sizable objects associated with the Axis deserialization/data-binding process. To remove these reference

replace the object by calling `org.globus.wsrp.encoding.ObjectSerializer.clone(originalObject)` and drop the reference to the original object.

1.3. Concurrent invocations and synchronization

The container does not provide automatic synchronization of concurrent requests. This means that service implementors need to write their services to deal with potential synchronization problems. If you service itself is stateless this means that if you will need to synchronize around access to the state captured in the resource. That being said, in a lot of scenarios you can expect a single client to drive most interactions with a given resource, in which case synchronization may not be an issue.

1.4. Lease-based lifetime management

We recommend that you make use of lease based resource lifetime management to avoid orphaned resources due to network outage and other failures. Lease based lifetime management is accomplished by specifying a initial lifetime in the resource creation operation followed by periodic updates to the lifetime using the `setTerminationTime` operation specified in the WS Resource Lifetime specification.

1.5. Resource Types and Services

Any given WSRF service can only have a single resource properties document schema associated with it. This implies a constraint on the type of resource a service can expose, i.e. resources exposed by a service must conform to the resource properties document schema associated with the service. Note that this does not necessarily mean that services can only expose resources of a single implementation type. There may of course be multiple implementation types as long as all of these types conform to the interface dictated by the resource property document schema. Also, the resource property document schema may have extensibility elements, allowing more flexibility in the resource implementation at the expense of reducing discoverability and implicitly predictability of interactions against resources exposed by such a service. That said, the Java WS Core implementations of the Resource Home interface only allow a single resource type.

In addition, services may interact with any type of resource, unconstrained by the resource property document schema, as long as these resources are not expose by the service. This occurs frequently in the standard factory pattern, where the factory creates and manipulates a resource, but the resource itself is exposed through a different service.

1.6. Messaging granularity

Our current performance profile for the Java WS Core component currently only allows for fairly coarse grained operations at a reasonable level of performance. While this statement is in relation to Java WS Core performance it is in reality relevant to any distributed applications: Remote invocations always cost more than local invocation (give infinite CPU power/memory) and should thus be treated differently than local invocations.

1.7. Choosing an authentication mechanism

GT4 provides a implementer with a choice of 4 authentication mechanisms: HTTPS (ie SSL/TLS), WS-Security with X.509 certificates, WS-Security Username/Password authentication and a GSSAPI based WS-Trust/SecureConversation/Security based mechanism. We recommend that service implementors try to support the greatest number of mechanisms possible. Generally, services that need both authentication and message protection should always allow any mechanism other than the username/password one.

The story is somewhat different on the client end of things. Whereas server side security configuration is mostly policy driven, clients actually have to pick a specific mechanism to implement. We recommend that clients use HTTPS whenever available, i.e. whenever the service URL indicates a HTTPS transport, but are able to fall back on WS-Se-

curity with X.509 certificates should the transport be a non-https one. This recommendation is based upon performance comparisons of HTTPS vs. WS-Security based mechanisms, which have shown HTTPS to be much higher performance, especially as the message payload grows.

Chapter 2. GT 4.2.1 Resource Properties Overview

1. Introduction

This page aggregates information about resource properties currently available throughout GT 4.2.1.

2. Security

2.1. Resource properties

- `supportedPolicies`: Contains identifiers for any or all access control policies that the authorization service is capable of rendering decisions regarding.
- `supportsIndeterminate`: Indicates whether the authorization service may return an "indeterminate" authorization decision. If set to false, only permit or deny is returned.
- `signatureCapable`: Indicates if the authorization service is capable of signing the decision returned. If not, only unsigned decisions are returned.

2.2. CAS Resource Properties

- `ServerDN`: The DN from the credentials used by the CAS Service
- `VODescription`: This is a string that describes the VO relevant to CAS Service.

2.3. Delegation Service Resource properties

2.3.1. Delegation Factory Service

- `CertificateChain`: This resource property is used to expose the certificate used by delegation service.

3. Data Management

3.1. RFT Resource Properties

The resource properties of RFT Factory (which acts both as a resource and a service at the same time) and RFT Resource are found below:

3.1.1. RFT Factory Resource Properties

- `ActiveResourceInstances`: A dynamic resource property of the total number of active RFT resources in the container at a given point of time.
- `TotalNumberOfTransfers`: A dynamic resource property of the total number of transfers/deletes performed since the RFT service was deployed in this container.

- `TotalNumberOfActiveTransfers`: A dynamic resource property of the number of active transfers across all rft resources in a container at a given point of time.
- `TotalNumberOfBytesTransferred`: A dynamic resource property of the total number of bytes transferred by all RFT resources created since the deployment of the service.
- `RFTFactoryStartTime`: Time when the service was deployed in the container. Used to calculate uptime.
- `DelegationServiceEPR`: The end point reference of the Delegation resource that holds the delegated credential used in executing the resource.

3.1.2. RFT Resource Properties

- `OverallStatus`: This is a complex type providing the overall status of an RFT resource by providing the number of transfers pending, active, finished, retrying, failed, and cancelled. Each of these values can be obtained by invoking `getTransfers(Finished/Active/Failed/Restarted/Pending/Cancelled)` on `OverallStatus` Resource Property. Note that this Resource Property gets updated every time one of the transfers changes state, so there can be and will be more than one update in the life time of a RFT resource if you subscribe to this RP. This Resource Property also includes the last fault (if thrown) from a transfer and can be accessed by invoking `getFault` on `OverallStatus`. This will indicate why a transfer has failed.
- `RequestStatus`: This is a complex type resource property providing the status of an RFT resource in the form of `Pending/Active/Done/Failed`. The status can be obtained from `RequestStatusType` by invoking `getRequestStatus()`. This will result in one of four status strings (`Pending/Active/Done/Failed/Cancelled`). This RP also contains a fault that denotes the last fault in a RFT resource and can be accessed by invoking `getFault()`. If a client is subscribed to this RP, there will be only be 2 updates in the life time of an RFT resource (`Pending->Active->Done`, `Pending->Active->Failed`, `Pending->Active->Cancelled`, and `Pending->Cancelled`).
- `TotalBytes`: This provides the total number of bytes transferred by the resource.
- `TotalTime`: This provides the total time taken to transfer the above-mentioned total bytes.

3.2. WS RLS Resource Properties

The resource properties for the `ReplicaLocationCatalog` `ReplicaLocatoinIndex` port types are listed below:



Note

As a *preview* component the current WS-RLS specifies the following RPs but in most cases does not implement them in the current release. The developers of the component are providing the interfaces for review.

3.2.1. ReplicaLocationCatalog Resource Properties

- `configuration`: A listing of the configuration settings for the underlying RLS service.
- `diagnostics`: A listing of diagnostics (e.g., status) from the underlying RLS service.
- `catalog`: A resource property sytle representation of the catalog contexts (i.e., attribute definitions, attributes, and mappings) of the underlying RLS service.

3.2.2. ReplicaLocationIndex Resource Properties

- `configuration`: A listing of the configuration settings for the underlying RLS service.

- `diagnostics`: A listing of diagnostics (e.g., status) from the underlying RLS service.
- `catalog`: A resource property style representation of the index contexts (i.e., attribute definitions, attributes, and mappings) of the underlying RLS service.

3.3. Batch Replicator Resource properties

Supported resource properties for DataRep include:

- `status`: The status of the resource, such as Pending, Active, Suspended, Terminated, Destroyed, etc.
- `stage`: The current stage or activity of the resource, such as Discover, Transfer, and Register.
- `result`: The final result (if any) of the resource, such as Finished, Failed, and Exception.
- `errorMessage`: A verbose description of an error (if any) encountered by the resource. The message may include error or exception information returned by one of the dependent components, such as RLS or RFT.
- `count`: An element containing counts of individual replication items pertaining to total, finished, failed, and terminated replication items.

4. Information Services

4.1. WS MDS Aggregator Framework Resource Properties

4.1.1. AggregatorServiceGroup Resource Properties

- `Entry`: This resource property publishes details of each registered resource, including both an EPR to the resource, the Aggregator Framework configuration information, and data from the sink.
- `RegistrationCount`: This resource property publishes registration load information (the total number of registrations since service startup and decaying averages)

5. Execution Management

5.1. Resource properties

5.1.1. Managed Job Factory Port Type

- `{http://www.globus.org/namespaces/2008/03/gram/job}condorArchitecture`
Condor architecture label.
- `{http://www.globus.org/namespaces/2008/03/gram/job}condorOS`
Condor OS label.
- `{http://www.globus.org/namespaces/2008/03/gram/job}delegationFactoryEndpoint`
The endpoint reference to the delegation factory used to delegated credentials to the job.
- `{http://mds.globus.org/glue/ce/1.1}GLUECE`

GLUE data

- {http://mds.globus.org/glue/ce/1.1}GLUECESummary

GLUE data summary

- {http://www.globus.org/namespaces/2008/03/gram/job}globusLocation
The location of the Globus Toolkit installation that these services are running under.
- {http://www.globus.org/namespaces/2008/03/gram/job}hostCPUType
The job host CPU architecture (i686, x86_64, etc...)
- {http://www.globus.org/namespaces/2008/03/gram/job}hostManufacturer
The host manufacturer name. May be "unknown".
- {http://www.globus.org/namespaces/2008/03/gram/job}hostOSName
The host OS name (Linux, Solaris, etc...)
- {http://www.globus.org/namespaces/2008/03/gram/job}hostOSVersion
The host OS version.
- {http://www.globus.org/namespaces/2008/03/gram/job}localResourceManager
The local resource manager type (i.e. *Condor*, Fork, *LSF*, Multi, *PBS*, etc...)
- {http://www.globus.org/namespaces/2008/03/gram/job}availableLocalResourceManager
All local resource managers that are configured in this GRAM4 instance
- {http://www.globus.org/namespaces/2008/03/gram/job}jobTTLAfterProcessing
Time in seconds a job resource will stay alive after a job finished processing in GRAM4 (including fileStageOut, fileCleanUp). When this time elapsed the job resource is destroyed and no longer be available for a client. A negative values means that the job resource will never be destroyed.
- {http://www.globus.org/namespaces/2008/03/gram/job}maxJobLifetime
Max time in seconds a user can set as initial lifetime in job submission or in subsequent setTerminationTime calls. A negative value means that there is no limit.
- {http://mds.globus.org/metadata/2005/02}ServiceMetaDataInfo
service start time, Globus Toolkit(R) version, service type name
- {http://www.globus.org/namespaces/2008/03/gram/job}scratchBaseDirectory
The directory recommended by the system administrator to be used for temporary job data.
- {http://www.globus.org/namespaces/2008/03/gram/job}stagingDelegationFactory-Endpoint
The endpoint reference to the delegation factory used to delegated credentials to the staging service (RFT).

5.1.2. Managed Job Port Type

- `{http://www.globus.org/namespaces/2008/04/rendezvous}Capacity`
Used for Rendezvous.
- `{http://docs.oasis-open.org/wsrf/r1-2}CurrentTime`
Time of creation.
- `{http://docs.oasis-open.org/wsrf/rp-2}QueryExpressionDialect`
From the QueryResourceProperties port type.
- `{http://www.globus.org/namespaces/2008/03/gram/job/faults}fault`
Faults (if generated) that happen along job processing and that cause a job to fail.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}holding`
Indicates whether a hold has been placed on this job.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}localUserId`
The job owner's local user account name.
- `{http://www.globus.org/namespaces/2008/04/rendezvous}RegistrantData`
Used for Rendezvous.
- `{http://www.globus.org/namespaces/2008/04/rendezvous}RendezvousCompleted`
Used for Rendezvous.
- `{http://www.globus.org/namespaces/2008/03/gram/job/description}service-LevelAgreement`
A wrapper around fields containing the single-job and multi-job descriptions or *RSLs*. Only one of these sub-fields shall have a non-null value.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}state`
The current state of the job.
- `{http://docs.oasis-open.org/wsrf/r1-2}TerminationTime`
Time when the resource expires.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}userSubject`
The GSI certificate DN of the job owner.

5.1.3. Managed Executable Job Port Type

- `{http://docs.oasis-open.org/wsrf/r1-2}CurrentTime`
Time of creation.

- `{http://docs.oasis-open.org/wsrfl-2}TerminationTime`
Time when the resource expires.
- `{http://www.globus.org/namespaces/2008/03/gram/job/exec}credentialPath`
The path (relative to the job process) to the file containing the user proxy used by the job to authenticate out to other services.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}exitCode`
The exit code generated by the job process.
- `{http://www.globus.org/namespaces/2008/03/gram/job/faults}fault`
The fault (if generated) indicating the reason for failure of the job to complete.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}holding`
Indicates whether a hold has been placed on this job.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}localUserId`
The job owner's local user account name.
- `{http://www.globus.org/namespaces/2008/03/gram/job/exec}localJobId`
The job id(s) of the job in the local resource manager. Note that for Fork jobs these id's are prefixed with the uuid of the job.
- `{http://www.globus.org/namespaces/2008/03/gram/job/description}service-LevelAgreement`
A wrapper around fields containing the single-job and multi-job descriptions or *RSLs*. Only one of these sub-fields shall have a non-null value.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}state`
The current state of the job.
- `{http://www.globus.org/namespaces/2008/03/gram/job/exec}stderrURL`
A GridFTP URL to the file generated by the job which contains the stderr.
- `{http://www.globus.org/namespaces/2008/03/gram/job/exec}stdoutURL`
A GridFTP URL to the file generated by the job which contains the stdout.
- `{http://www.globus.org/namespaces/2008/03/gram/job/types}userSubject`
The GSI certificate DN of the job owner.
- `{http://www.globus.org/namespaces/2008/04/rendezvous}Capacity`
Used for Rendezvous.
- `{http://www.globus.org/namespaces/2008/04/rendezvous}RegistrantData`

Used for Rendezvous.

- {http://www.globus.org/namespaces/2008/04/rendezvous}RendezvousCompleted

Used for Rendezvous.

- {http://docs.oasis-open.org/wsrf/rp-2}QueryExpressionDialect

From the QueryResourceProperties port type.

5.1.4. Managed Multi-Job Port Type

- {http://docs.oasis-open.org/wsrf/r1-2}CurrentTime

Time of creation.

- {http://docs.oasis-open.org/wsrf/r1-2}TerminationTime

Time when the resource expires.

- {http://www.globus.org/namespaces/2008/03/gram/job/faults}fault

The fault (if generated) indicating the reason for failure of the job to complete.

- {http://www.globus.org/namespaces/2008/03/gram/job/types}holding

Indicates whether a hold has been placed on this job.

- {http://www.globus.org/namespaces/2008/03/gram/job/types}localUserId

The job owner's local user account name.

- {http://www.globus.org/namespaces/2008/03/gram/job/description}service-LevelAgreement

A wrapper around fields containing the single-job and multi-job descriptions or *RSLs*. Only one of these sub-fields shall have a non-null value.

- {http://www.globus.org/namespaces/2008/03/gram/job/types}state

The current state of the job.

- {http://www.globus.org/namespaces/2008/03/gram/job/multi}subJobEndpoint

A set of endpoint references to the sub-jobs created by this multi-job.

- {http://www.globus.org/namespaces/2008/03/gram/job/types}userSubject

The GSI certificate DN of the job owner.

- {http://www.globus.org/namespaces/2008/04/rendezvous}Capacity

Used for Rendezvous.

- {http://www.globus.org/namespaces/2008/04/rendezvous}RegistrantData

Used for Rendezvous.

- `{http://www.globus.org/namespaces/2008/04/rendezvous}RendezvousCompleted`
Used for Rendezvous.
- `{http://docs.oasis-open.org/wsrp/rp-2}QueryExpressionDialect`
From the QueryResourceProperties port type.

Chapter 3. GT 4.2.1 Guide to Samples

This page contains links to samples currently available throughout GT 4.2.1.

- [Common Runtime Components](#)
 - [Java WS Core](#)
 - [C WS Core](#)
- [WS MDS](#)

Part I. Asynchronous Event Handling with Examples

Table of Contents

4. GT 4.2.1: Asynchronous Event Handling	15
1. Examples	15
2. Event Models	15
3. Callback Library	17
4. Thread Abstraction	18
5. Asynchronous Model	19
6. Conclusion	21
7. References	21
5. Asynchronous Event Handling: Example 1	22
6. Asynchronous Event Handling: Example 2	23
7. Asynchronous Event Handling: Example 3	25

Chapter 4. GT 4.2.1: Asynchronous Event Handling

The Globus Toolkit contains several APIs written in C for creating grid applications. Each of these components is built on a coherent asynchronous event model. This text will introduce and explain the philosophy behind the model and its basic concepts.

1. Examples

- [Example 1](#) - Demonstrates basic use of `globus_callback_register_oneshot()`
- [Example 2](#) - An example of `globus_callback_register_oneshot()` using condition variables
- [Example 3](#) - Game of Craps example demonstrates a more complex use of the asynchronous event model

2. Event Models

The Globus Toolkit uses an **asynchronous event model**. Details of this model are contained in the remainder of this text but it will be helpful to take a few examples of other popular models.

Applications existing in event heavy environments, such as graphical user interfaces (GUIs), IO, or inter-process signaling, must implement some event model. Events are characterized by changes in the environment at an undetermined time. There are several different popular models used to handle such events. We provide examples of them here, and then describe in detail the asynchronous event model used by the Globus Toolkit.

2.1. Blocking Event Model

In a blocking API, an event is serviced, delaying all processing in the current thread of execution until the event completes. This has the obvious disadvantage that no processing can be done while waiting on the IO. Typically this is solved by forking additional processes or creating additional threads to service each event. However, more processes and more threads make a more resource intensive application.

Example: Blocking Event Model

```
main()
{
    while(!done)
    {
        ~ other processing ~
        data = ReadData();
        ~ process event ~
    }
}
```

2.2. Non-blocking Event Model

A non-blocking model follows the same in-line procedural model as blocking except that events are polled for completion. Instead of blocking all processing until the event completes, the user asks if the event is complete. If so, the event is processed. If not, other processing may resume.

Example: Non-blocking Event Model

```
main()
{
    while(!done)
    {
        if(EventIsReady())
        {
            ~ process event ~
        }
        else
        {
            ~other processing ~
        }
    }
}
```

Unlike the blocking model, this approach allows for simultaneous processing while waiting for the event. However, it can become cumbersome as more and more events are added. Further, if there is no other processing to be done, it results in tight spin loops that use the CPU simply to poll for events.

2.3. Asynchronous Event Model

The asynchronous approach does not follow the in-line procedure. Instead events are given handler functions. A user registers for an event with the system, giving it a handler function. When the event occurs the system calls the user's handler function.

Example: Asynchronous Event Model

```
event_handler()
{
    process event
    register_next_event();
}

main()
{
    ~ other processing ~
    resiter_event()
    ~ other processing ~
    while(!done)
    {
        wait_for_events();
    }
}
```

```
}

```

Like the non-blocking model, this allows simultaneous event and data processing. In this model, programs are designed as a series of events rather than a serial execution of instructions. A programmer registers events and when they occur the necessary processing is done. Additional events may then be registered and the program goes back to waiting for events. This is the approach taken by the Globus Toolkit.

3. Callback Library

The heart of the Globus event model is the callback library. This API provides a user with functionality for asynchronous time events. In order to use the API for events, the user must implement a function (the callback) that is called when the event has occurred and processes it.

There are two fundamental functions that explain the API:

```
globus_result_t
globus_callback_register_oneshot(
    globus_callback_handle_t *      callback_handle,
    const globus_reftime_t *        delay_time,
    globus_callback_func_t         callback_func,
    void *                           callback_user_args);

globus_result_t
globus_poll();
```

The first function is fairly clear. It registers the callback `callback_func` with the system that will be called once the time specified by `delay_time` has expired.

The more interesting of the two is `globus_poll()`. Semantically this function is used to briefly turn control over to the Globus event system for processing. What this means is that `globus_poll()` must be called often enough for the Globus event system to function. This is recognized as a rather ambiguous statement. Therefore, a look at what happens with `globus_poll()` should assist in explanation. In threaded builds of Globus this `globus_poll()` simply results in a call to `thread_yield()` where control can be switched to a background thread dedicated to event processing. In non-threaded builds, a list of events is maintained by the system. A call to `globus_poll()` finds ready events in the list and dispatches the associated callback to the user within the same call stack.

In [Example 1](#) a use of these two functions is displayed. The function `user_callback` is registered for execution after 1 second has elapsed.

In a non-threaded build, there is a single thread of execution. In the main loop, the call to `globus_poll()` invokes the Globus event process code. The code checks internal data structures for any ready events. If found, the user callbacks associated with the events will be called in the same call stack.

In a threaded build a user would see two threads (possibly more, but for the sake of clarity two will be used): the main thread that is executing the loop in `main()` and an internal Globus thread that is handling polling of events. The Globus thread is created when the user calls `globus_module_activate(GLOBUS_COMMON_MODULE)`. This function must be called before any API function in the the `globus_common` package can be used. This is another common theme in Globus: all modules must be activated before use and deactivated when finished. The event thread polls all events and as they become ready the functions associated with them are dispatched.

Another important concept to note in this API is the use of the `void * user_arg` parameter. This is a simple but important part of the model. On registration of an event, a user can pass in a void pointer and this pointer will be

threaded through to their event callback. The pointer can point to any bit of memory the user likes. Typically it points to some structure that allows the user to maintain state throughout a series of event callbacks. This memory is completely managed by the user, so if the memory is used in the event callback the user needs to be careful to **not** free the memory until the callback occurs. For a more complicated example of this see [Example 2](#).

4. Thread Abstraction

The first thing to look at in understanding the Globus event model is the thread abstraction layer. Globus can be built in a variety of ways with regard to the underlying thread system. It can be built with pthreads, win32 threads, or non threaded depending on the user's preferences and the available packages on the system. All builds present the same API. This thread API is very much akin to pthreads. If the reader is not familiar with pthreads, we recommend reading the pthread manual. The more notable API interface is presented below:

```
int
globus_thread_create(
    globus_thread_t *      thread,
    globus_threadattr_t *  attr,
    globus_thread_func_t   func,
    void *                  user_arg);

int
globus_mutex_lock(
    globus_mutex_t *      mutex);

int
globus_mutex_unlock(
    globus_mutex_t *      mutex);

int
globus_cond_wait(
    globus_cond_t *      cond,
    globus_mutex_t *      mutex);

int
globus_cond_signal(
    globus_cond_t *      cond);
```

It is important to note that this is **not** a complete set of necessary functions to properly use the threaded API. However, for the purposes of this text, they will serve for an explanation.

- `globus_thread_create()` will start a new thread of execution with a new call stack running beginning at the parameter `func`.
- `globus_mutex_lock()` and `globus_mutex_unlock()` provide mutual exclusive among threads over critical sections of code.
- `globus_cond_wait()` and `globus_cond_signal()` provide a means of thread synchronization.
- `wait()` will delay the thread that calls it until some other thread calls `signal()`.

In most cases the thread layer abstraction is a very thin pass through to the underlying thread package.

The notable exception is the non-threaded build. The Globus Toolkit has created a non-threaded, semantically equivalent implementation of all the functions described above (and of most in the pthreads API) with the exception of `globus_thread_create()`. In the non-threaded case this is a no-op. However the model of asynchronous programming used in the Globus Toolkit, `globus_thread_create()` is rarely needed or used.

In the Globus model, the callback code and the thread abstraction are coupled. [Example 2](#) shows how this coupling works:

1. An event is registered in the main thread, then `globus_cond_wait()` is called.
2. When the event has been processed, the handler is called.
3. The handler signals the wait that it may continue, then exits.
4. The signal awakens the wait so the main thread may continue.
5. The main thread then exits.

In the threaded build, `globus_cond_wait()` and `globus_cond_signal()` are simple passes through to the underlying thread packages, and as described previously, a background thread delivers the event.

In the non-threaded build, `globus_cond_wait()` will call `globus_poll()` and the non-threaded polling code takes over. For this reason, it is often not necessary to call `globus_poll()` in non-threaded builds. `globus_cond_wait()` tends to be used often enough to satisfy the needs of the event system.

5. Asynchronous Model

In many ways, the asynchronous programming model is the most difficult of the three presented. The blocking model is clearly the easiest, because everything happens in-line, and when the event function (like a read or a write) returns, the event has completed and all data is available. Events in this model are treated just like any other function call and are therefore easily dealt with by programmers with modest logic skills.

The non-blocking model is a bit more complicated than blocking, but not much. The only twist is that a user must check to see if the event completed and, if so, how much of it completed. This still allows for in-line processing; it only requires an additional `if` statement. Even when event polling is multiplexed (for example, `posix select`) the processing is still inline. The user must add some branches to determine what event is ready and then process it. The most difficult challenge of the non-blocking model is making use of the idle time when no events are ready.

In both non-blocking and blocking, the user has easy, in-line control over when an event is processed. If there is any logic that must occur before the event, the user simply needs to complete that processing before calling either the blocking function or the non-blocking function which checks for ready events. The asynchronous model removes this luxury. In the asynchronous model events can occur at any time. This can complicate the logic of keeping critical sections of code safe. Further complication is caused by the fact that they come in via their own handlers. This removes the luxury of maintaining state on the local stack. Instead all state must be packed into heap allocated structures which are passed to the callbacks via `void *` pointers (see the monitor structure in [Example 2](#)).

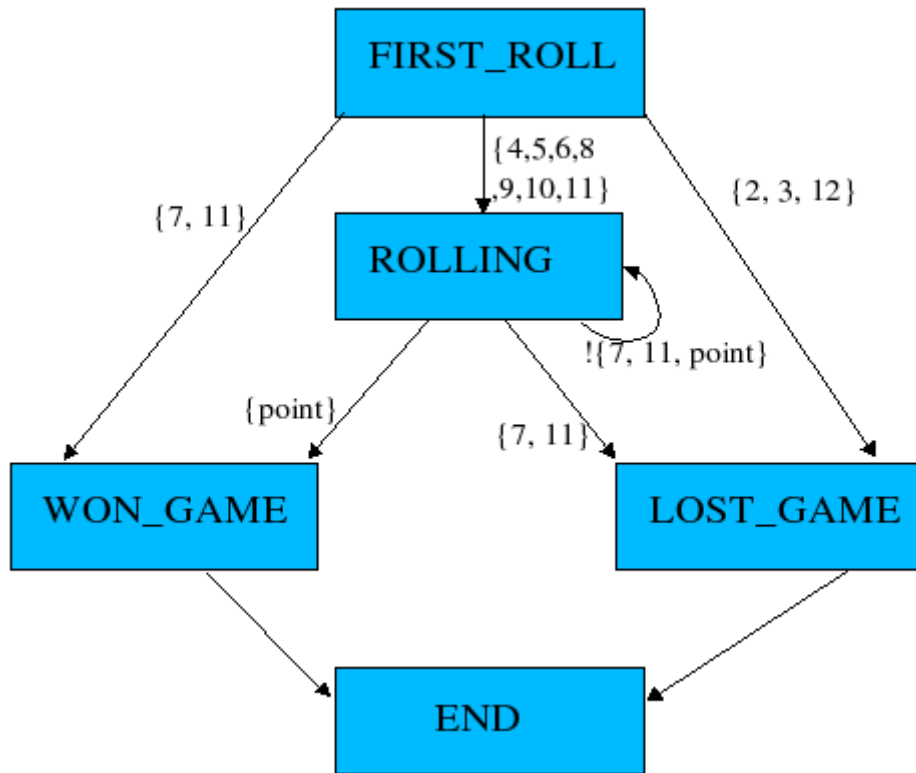
The upside to the asynchronous model is that it forces cleaner, more well thought out code. The non-blocking model does not scale well. As more events are managed, the event processing code becomes unmanageable, typically resulting in a single function that is far too long and far too interdependent for practical maintenance. Since users can use local variables, the tendency is to use many different flags to control state instead of a clean, well thought out state machine. This is especially true with software that evolves over time, growing in complexity.

In contrast the asynchronous model scales very well. Every event has a clean separation of being scoped to a user handler function. All shared states among events must be encapsulated into a data structure. A side effect of these two

characteristics is that it is easier for a user to define and follow a state machine then it is to create spaghetti logic based on many flags.

Example 3¹ shows a proper use of the asynchronous model. This example simulates the game of Craps. Craps is a dice game, the rules of which can be found with a simple web search, but the following state diagram should explain the rules well enough for this example.

Figure 4.1. State Diagram



Example 3 follows this state diagram. In the example rolls of the dice are considered events. For the sake of simplicity the example only uses a one shot event and then gets the data by calling `random()`; If this were a real world event, the values for the dice would come in as part of the event function. Notice how each time the event occurs the state is checked and, if needed, advanced to the next state. In the main function the program waits until the state machine comes to the final stage, where it signals the wait and allows for the program to end.

5.1. Blocking in Callbacks

What happens if an event handler blocks? The correct answer to this question is: **They never should**. This answer is of course a bit naive. There will be times when blocking in a callback is the only solution, and there will be even more times when it is the chosen solution, albeit the wrong one. Therefore, the Globus Toolkit does have mechanisms to allow this. That said, a user should make every effort to find alternative solutions to blocking in event callbacks. If the only solution is to block in a callback it could be an indication that the state machine is erroneous.

If an event callback is going to block, it must call the following function: `void globus_thread_blocking_will_block()`; If `globus_cond_wait()` is called, this function is implied.

¹ [globus-async-example3.html](#)

In the threaded build of Globus there is a background thread that handles the polling of events and dispatching of the handler functions. When a handler function blocks, it prevents this process. `globus_thread_blocking_will_block()` starts a new thread to handle event processing and allows the user to take over the current thread without stopping the processing of other threads. The user must also call `globus_poll()` in order to ensure that event processing continues.

This is needed:

- in the threaded case to yield the user's processing thread to the system event thread.
- in the non threaded case so that the only thread can make a non-blocking run through of any ready events.

6. Conclusion

The Globus Toolkit is middleware for the grid. Because grid infrastructure often depends heavily on both push and pull notifications (remote events), the callback style event handling model the Globus Toolkit provides is essential. It allows entire APIs within the toolkit to be designed with asynchronous functions that use the event handling model. Once an API provides that asynchronous functionality (such as XIO), software that builds on top of it can leverage this functionality. This eases the burden of the application programmer, as they need only to implement a callback function to handle possibly many notification events efficiently, instead of stopping execution until one is received, or managing multiple threads.

In the Globus Toolkit, because of the thread abstraction it provides, threads are managed by the underlying code base, so that the developer can be ignorant of using threads but still be able to get their benefits, simply by specifying a compile time switch. Overall, this flexibility is quite powerful, which is why we encourage the use of this model when designing and developing your own software components using the Globus Toolkit.

7. References

- [Posix Threads API](#)²
- [Microsoft's description of Completion Ports and Thread Pooling](#)³
- [Globus Common API](#)⁴
- [Documentation on Programming with Events](#)⁵

² <http://www.opengroup.org/onlinepubs/007908799/xsh/pthread.h.html>

³ http://msdn.microsoft.com/library/default.asp?url=/library/en-us/fileio/base/i_o_completion_ports.asp

⁴ http://www-unix.globus.org/api/c-globus-3.2/globus_common/html/index.html

⁵ <http://www.cs.bgu.ac.il/~elhadad/se/events.html>

Chapter 5. GT 4.2.1: Asynchronous Event Handling: Example 1

```
#include <globus_common.h>

void
user_callback(
    void *                user_arg)
{
    int *                count;

    count = (int *) user_arg;
    fprintf(stdout, "User callback, count = %d\n", *count);
    exit(0);
}

int
main(
    int                argc,
    char **            argv)
{
    globus_reltime_t    delay;
    int                count = 0;

    globus_module_activate(GLOBUS_COMMON_MODULE);

    GlobusTimeReltimeSet(delay, 1, 0);
    globus_callback_register_one-shot(
        NULL,
        &delay,
        user_callback,
        &count);

    while(1)
    {
        usleep(10000);
        globus_poll_nonblocking();
        fprintf(stdout, "After poll\n");
        count++;
    }

    globus_module_deactivate(GLOBUS_COMMON_MODULE);

    return 0;
}
```

Chapter 6. GT 4.2.1: Asynchronous Event Handling: Example 2

```
#include <globus_common.h>

struct test_monitor_s
{
    globus_mutex_t      mutex;
    globus_cond_t       cond;
    globus_bool_t       done;
};

void
user_callback(
    void *              user_arg)
{
    struct test_monitor_s *    monitor;

    monitor = (struct test_monitor_s *) user_arg;

    globus_mutex_lock(&monitor->mutex);
    {
        fprintf(stdout, "Signaling the wait\n");
        monitor->done = GLOBUS_TRUE;
        globus_cond_signal(&monitor.cond);
    }
    globus_mutex_unlock(&monitor->mutex);
}

int
main(
    int      argc,
    char **  argv)
{
    struct test_monitor_s    monitor;
    globus_reftime_t        delay;

    globus_module_activate(GLOBUS_COMMON_MODULE);

    globus_mutex_init(&monitor.mutex, NULL);
    globus_cond_init(&monitor.cond, NULL);
    monitor.done = GLOBUS_FALSE;

    globus_mutex_lock(&monitor.mutex);
    {
        GlobusTimeReltimeSet(delay, 1, 0);
        globus_callback_register_oneshot(
            NULL,
            &delay,
            user_callback,
            &monitor);
    }
}
```

```
    while(!monitor.done)
    {
        fprintf(stdout, "waiting...\n");
        globus_cond_wait(&monitor.cond, &monitor.mutex);
    }
}
globus_mutex_unlock(&monitor.mutex);

globus_module_deactivate(GLOBUS_COMMON_MODULE);

fprintf(stdout, "Done\n");

return 0;
}
```

Chapter 7. GT 4.2.1: Asynchronous Event Handling: Example 3

```
#include <globus_common.h>
#include <stdlib.h>

typedef enum game_state_e
{
    FIRST_ROLL,
    ROLLING,
    LOST_GAME,
    WON_GAME
} game_state_t;

typedef struct game_context_s
{
    globus_mutex_t      mutex;
    globus_cond_t       cond;
    game_state_t        state;
    int                 rolls;
    int                 point;
} game_context_t;

void
event_callback(
    void *              user_arg)
{
    int                 die1;
    int                 die2;
    game_context_t *    game_context;

    game_context = (game_context_t *) user_arg;

    die1 = rand() % 6 + 1;
    die2 = rand() % 6 + 1;

    globus_mutex_lock(&game_context->mutex);
    {
        game_context->rolls++;
        fprintf(stdout, "you rolled %d and %d, total is %d\n",
            die1, die2, die1+die2);
        switch(game_context->state)
        {
            case FIRST_ROLL:
                if(die1+die2 == 7 || die1+die2 == 11)
                {
                    game_context->state = WON_GAME;
                    globus_cond_signal(&game_context->cond);
                }
                else if(die1+die2 == 2 || die1+die2 == 3 || die1+die2 == 12)
```

```
        {
            game_context->state = LOST_GAME;
            globus_cond_signal(&game_context->cond);
        }
        else
        {
            game_context->state = ROLLING;
            game_context->point = die1+die2;
            fprintf(stdout, "The point is: %d\n", game_context->point);
            globus_callback_register_oneshot(
                NULL,
                NULL,
                event_callback,
                game_context);
        }
        break;

    case ROLLING:
        if(die1+die2 == 7)
        {
            game_context->state = LOST_GAME;
            globus_cond_signal(&game_context->cond);
        }
        else if(die1+die2 == game_context->point)
        {
            game_context->state = WON_GAME;
            globus_cond_signal(&game_context->cond);
        }
        else
        {
            globus_callback_register_oneshot(
                NULL,
                NULL,
                event_callback,
                game_context);
        }
        break;

    default:
        globus_assert(0 && "should never reach this state");
        break;
    }
}
globus_mutex_unlock(&game_context->mutex);
}

int
main(
    int          argc,
    char **      argv)
{
    game_context_t      game_context;

    globus_module_activate(GLOBUS_COMMON_MODULE);
```

```
globus_mutex_init(&game_context.mutex, NULL);
globus_cond_init(&game_context.cond, NULL);
game_context.rolls = 0;
game_context.state = FIRST_ROLL;

srandom(time(NULL));

globus_mutex_lock(&game_context.mutex);
{
    globus_callback_register_oneshot(
        NULL,
        NULL,
        event_callback,
        &game_context);

    while(game_context.state != LOST_GAME &&
          game_context.state != WON_GAME)
    {
        globus_cond_wait(&game_context.cond, &game_context.mutex);
    }
}
globus_mutex_unlock(&game_context.mutex);

fprintf(stdout, "%s, game over in %d rolls.\n",
        game_context.state == LOST_GAME ? "You LOSE" : "You WIN",
        game_context.rolls);

globus_module_deactivate(GLOBUS_COMMON_MODULE);
return 0;
}
```

Appendix A. Globus Toolkit 4.2.1 Public Interface Guides

This page contains links to each GT 4.2.1 component's Public Interfaces Guide.

- [Common Runtime Components](#)
 - [Java WS Core](#)
 - [C WS Core](#)
 - [XIO](#)
 - [C Common Libraries](#)
- [Security](#)
 - [GSI C](#)
 - [Java WS Authentication & Authorization](#)
 - [C WS Authentication & Authorization](#)
 - [CAS](#)
 - [Delegation Service](#)
 - [MyProxy](#)
 - [GSI-OpenSSH](#)
- [Data Management](#)
 - [RFT](#)
 - [GridFTP](#)
 - [RLS](#)
 - [WS RLS](#)
 - [Batch Replication Service](#)
 - [Replication Client](#)
- [Information Services](#)
 - [WS MDS Index Service](#)
 - [WS MDS Trigger Service](#)
 - [WS MDS Aggregator Framework](#)
 - [WebMDS](#)

- [UsefulRP](#)
- Execution Management
 - [GRAM4](#)
 - [GridWay](#)

Glossary

C

Condor A job scheduler mechanism supported by GRAM. See <http://www.cs.wisc.edu/condor/> for more information.

L

LSF A job scheduler mechanism supported by GRAM.
For more information, see <http://www.platform.com/Products/Platform.LSF.Family/Platform.LSF/>⁷.

P

Portable Batch System (PBS) A job scheduler mechanism supported by GRAM. For more information, see <http://www.openpbs.org>.

R

Resource Specification Language (RSL) Term used to describe a GRAM job for GT2 and GT3. (Note: This is not the same as RLS - the Replica Location Service)

⁷ <http://www.platform.com/Products/Platform.LSF.Family/Platform.LSF/>