Globus Toolkit 6.0 Developer's Guide

Globus Toolkit 6.0 Developer's Guide

Introduction

Note

Please note that the Best Practices need to be updated for GT5 and are not being published at this time.

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

- Installation Guide
- Quickstart
- All GT command line clients are listed here.

[1] gtDeveloperGuide.pdf

Table of Contents

List of Figures

Part 1. Asynchronous Event Handling with Examples

Table of Contents

Chapter 1. GT 6.0: 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 ~
    register_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_reltime_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).

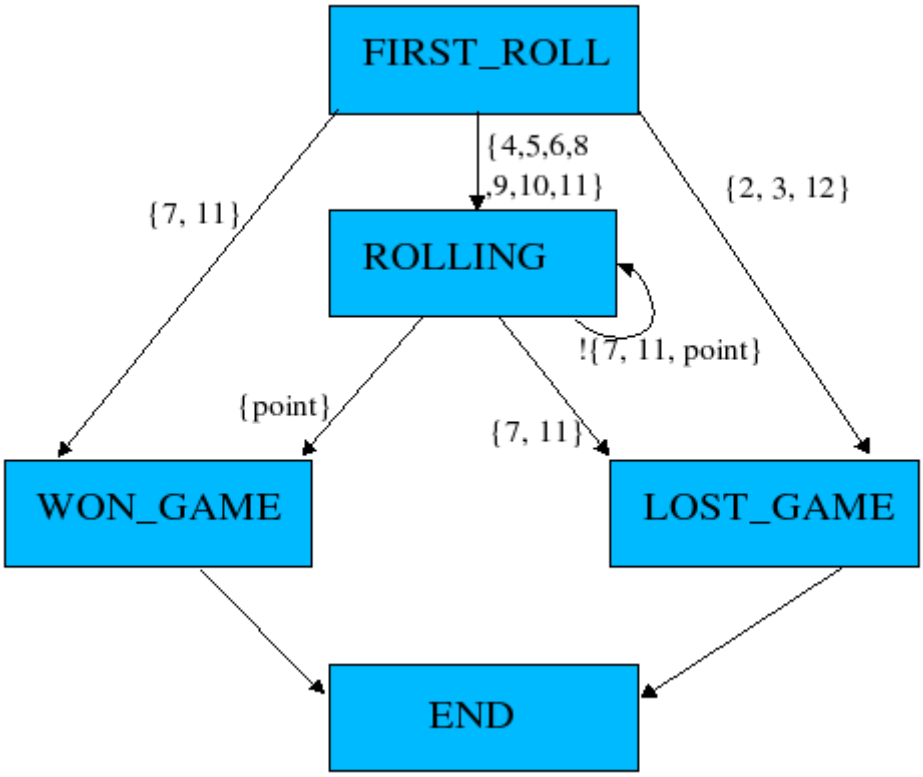
The upside to the asynchronous model is that it forces cleaner, more well thought out code. The non-blocking model does not scale well. As more events are managed, the event processing code becomes unmanageable, typically resulting in a single function that is far too long and far too interdependent 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 3[1] 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.

[1] globus-async-example3.html

Figure 1.1. 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.

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]

Chapter 2. GT 6.0: 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_oneshot(
        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 3. GT 6.0: 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_reltime_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 4. GT 6.0: 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 6.0 Public Interface Guides

This page contains links to each GT 6.0 component's Public Interfaces Guide.

- GridFTP
- GRAM5
- Security
  - GSI C
  - MyProxy
  - GSI-OpenSSH
- Common Runtime Components
  - XIO
  - C Common Libraries

Glossary