Go4 Multi-tasking Class Library with ROOT

J.Adamczewski, M.Al-Turany, D.Bertini, H.G.Essel, M.Hemberger, N.Kurz, M.Richter
Gesellschaft für Schwerionenforschung
64291 Darmstadt, Planckstr. 1, Germany

Abstract **In the situation of monitoring an experiment it is often necessary to control several independently running tasks from one Graphical User Interface (GUI). Such a GUI must be able to execute commands in the tasks even if they are busy, i.e. getting data, analyzing data or waiting for data. Moreover, the tasks, being controlled by data streams (i.e. event data samples or slow control data), must be able to send data asynchronously to the GUI for visualization.**

A multi-tasking package (C++ class library) that meets these demands has been developed at the GSI in the framework of a new analysis system, Go4, which is based on the ROOT system [CERN, R.Brun et al.]. The package provides a thread manager, a task handler, and asynchronous inter task communication between threads through sockets. Hence, objects can be sent at any time from a task to the GUI or vice versa. At the GUI side an incoming object is accepted by a thread and processed. In a task an incoming command is queued by the accepting thread and executed in the execution thread.

Utilizing the package one can implement nonblocking GUIs to control one or several tasks processing data in parallel and updating graphical elements in the GUI. The package could also be useful in building data dispatchers or in slow control applications.

All components have been tested with Go4 analysis tasks and a very preliminary GUI.

I. Introduction

At GSI a new Object Oriented On-line Off-line analysis system, Go4 [1], is under development. It is based on the ROOT system [2] developed at CERN, because ROOT provides a powerful framework to analyze data from high energy and nuclear physics experiments. One requirement for Go4 is the possibility to monitor and control a running analysis without blocking the controlling GUI, especially on-line.

For the required asynchronous operations usually the technique of multi-threading is applied. After detailed investigations, however, it was clear that threads cannot be used in general with the ROOT library because it is not inherently thread save, i.e. there are conflicts if two threads simultaneously create objects or perform ROOT system calls. This is because ROOT works internally with lists as object registries, and does not have an internal memory allocation lock.

The *Go4ThreadManager* package has been implemented to control these conflicts, e.g. by providing locking mechanisms against the ROOT application loop. It is based on the ROOT TThread library which had to be brought to a functional working status first. However, it is still not possible to run two threads both modifying the same ROOT objects simultaneously, or both using extensively the ROOT system, e.g. for data streaming, without explicit locking.

Therefore, and since at least two main working threads (data analysis and user display) were required for the Go4 framework, we were forced to split them into two tasks. For the implementation of such multi-tasking systems the *Go4TaskHandler* package has been developed.

Furthermore, the problem of common objects in different tasks had to be solved. The solution of using shared memory is limited with C++. The shared memory mechanism provided by ROOT forces an update (copy) of the whole memory for each shared access, so it is not really shared. Measurements showed, however, that local socket transfer reaches memory copy speed.

Therefore socket channels between the tasks have been implemented using ROOT TSockets which allow to transport entire objects by means of the powerful ROOT object streamer mechanism. Since Go4 depends on the ROOT environment, more general but external solutions such as CORBA have not been chosen.

It is an additional benefit, that by means of socket connections between the tasks it is as well possible to run the controlling GUI and the controlled application (analysis) on different machines.

The layers of the two packages described in the following are shown in Fig. 1. The Go4 event analysis loop and GUI are used as examples of task handlers.

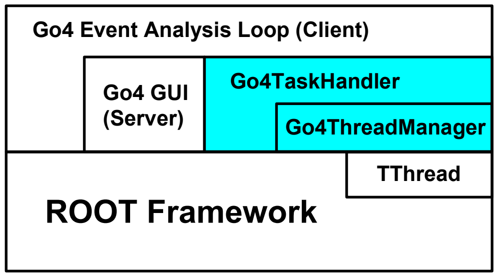


Fig.1. The software layers of ROOT and Go4

II. Thread Manager

The *Go4ThreadManager* package provides foundation and service classes to launch any number of named threads within a ROOT application. It implements the concept of runnable classes (like JAVA). Fig. 2 shows a simplified UML diagram of the thread manager classes. The TGo4ThreadManager (usually there is only one thread manager object per task) aggregates a thread-safe list of TGo4Thread objects, i.e. the TGo4ThreadHandler. The thread objects encapsulate the ROOT TThread classes and can be accessed by name from the TGo4ThreadHandler.

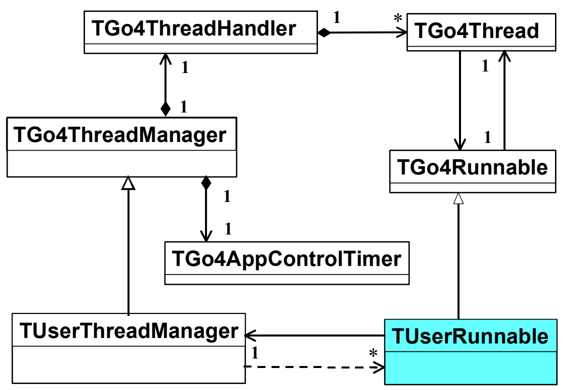


Fig. 2. Class diagram of thread manager.

Each TGo4Thread instance is linked to a TGo4Runnable instance which defines the action being executed within the thread in a virtual Run() method. By sub-classing TGo4Runnable and TGo4ThreadManager this framework can be adopted to any user application. The TGo4Thread calls the TGo4Runnable::Run() method in a loop which can be stopped and started by the thread manager without cancelling the TThread itself. The stopped TGo4Thread then waits for a TThread condition signal from the thread manager to start the runnable loop again.

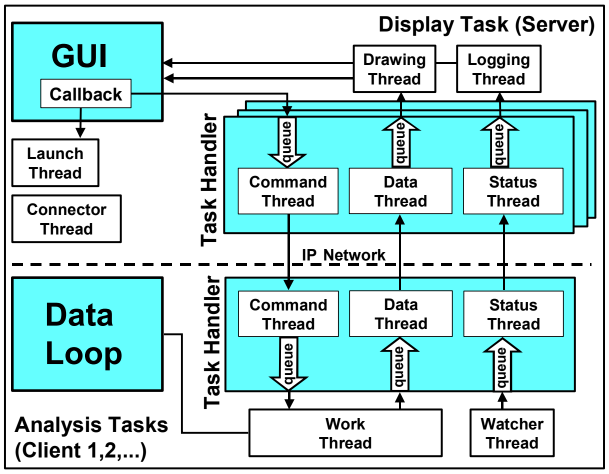


Fig. 3. Functional sketch of display server with analysis clients, connected by task handler instances.

Since the ROOT system is not thread-safe itself, a timer (TGo4AppControlTimer, Fig. 2) pending on a TThread condition is used to block or release the main application loop on demand, and to terminate the application. Other critical operations, like initializing a non thread-safe ROOT TServerSocket, may be executed by additional timers polling on status flags (e.g. the TGo4TaskConnectorTimer of the task handler, Fig. 4)

Exceptions are handled independently in each thread. Any exception thrown in the runnable function are caught from the corresponding TGo4Thread instance. The user may specify the exception reaction by overriding virtual exception handling methods of TGo4Runnable. Moreover, the Go4 exception hierarchy allows the user to create exception sub-classes containing their specific virtual handling method which may be overridden. This method is called from the framework by default on throwing this exception class. For the Go4 packages, we apply special exceptions both for rapid command action purposes and for error handling.

III. THE GO4 TASK HANDLER

To control several independent, distributed analysis clients (slaves) from one user interface server task (master) the *Go4TaskHandler* package and the related service packages (*Go4Socket*, *Go4Queue*, *Go4CommandsBase*, and *Go4StatusBase*) have been implemented. Fig. 3 shows a functional overview of this system.

The client-server communication is done via three socket channels (data, command, and status channel). Each of these channels is processed by a dedicated thread and is buffered against the user's application by means of a thread-safe template queue. The entire communication setup is encapsulated within the TGo4TaskHandler class. Fig. 4 shows the simplified UML diagram of the relevant classes. The TGo4TaskHandler object may run in client or server mode, with different directions of the transport channels (see also Fig. 3). The three communication threads of the task handler are defined by three different classes: TGo4CommandRunnable, TGo4DataRunnable, and TGo4-StatusRunnable, respectively. These are sub-classes of the TGo4TaskHandlerRunnable, which is aware of the parent task handler and which may be accessed by the task handler exceptions.

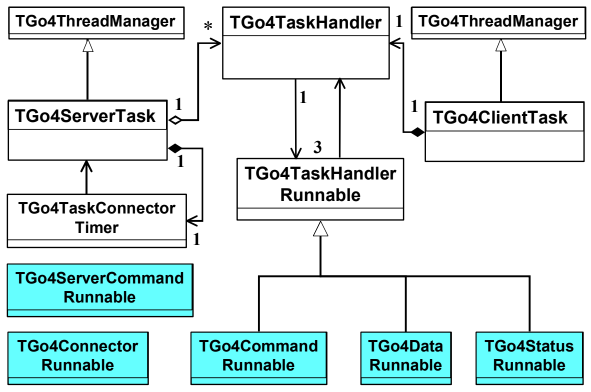


Fig. 4. Class diagram of task handler.

For each connected client, the TGo4ServerTask has one instance of the TGo4TaskHandler which it keeps in a thread-safe list, and which may be accessed by name. In addition, a connector thread of the server (TGo4ConnectorRunnable, Fig. 4) listens on a separate socket for any external client request, i.e. to set up a new connection or to shutdown an old one. A new client could be launched externally and connect to a running GUI server. But in most applications a new client will be started from within the GUI via remote shell, send a connection request back to the server and the connection will be established.

Because ROOT has problems with handling creation and termination of server sockets in a thread, an additional ROOT timer (TGo4TaskConnectorTimer, Fig. 4) to initialize and shut down the connection is used. This timer polls for a socket connection or disconnection request from the connector thread and then creates or deletes the socket instances. Once the socket is created by the timer, it is passed by class member pointer to the responsible communication thread (data, command, or status thread, respectively). These threads then can use the existing socket to send and receive ROOT objects until the client disconnects. This disconnection request again is handled by the connector thread which lets the task connector timer discard the existing connection in a ROOT compatible manner.

If a client-server connection breaks down (e.g. by a crashing or killed analysis client), an exception takes care for proper termination of the client. The server and the other clients continue.

Exchange of information between server and client is done by command objects and status objects. A command design pattern with an invoker singleton [3] and a modified memento pattern for the status objects have been implemented. Commands are created by the server and are usually sent to one of the remote clients. Local commands, like launching a new client by remote shell or quitting the server, are added to a local command queue and are executed by a special launcher thread (TGo4ServerCommandRunnable, Fig. 4). This prevents the GUI from being blocked in these cases.

Status objects are created by the client and are sent to the server which may e.g. display the current analysis status. In addition, any named ROOT object created by the analysis client (e.g. a histogram) can be transported to the GUI via the task handler data channel.

IV. APPLICATION OF THE TASK HANDLER

To adopt the Go4 task handler framework for a special application purpose, sub-classes of the TGo4ClientTask and TGo4ServerTask classes must be implemented. They provide additional working threads. The queues of the three inter-task communication channels are accessible by public methods of the task handler instance. Thus the application threads may wait for these queues to get a remote object or they may add new objects to these queues which are then sent by the task handler to the remote side.

In the following the Go4 GUI and the Go4 analysis are taken as an example of a task handler application. Fig. 5 shows the class diagram of this example.

A. Server Task

The TUserServer class inherits from TGo4ServerTask and hence also from TGo4ThreadManager. Therefore it contains all services required to launch own threads and to establish connections to a client.

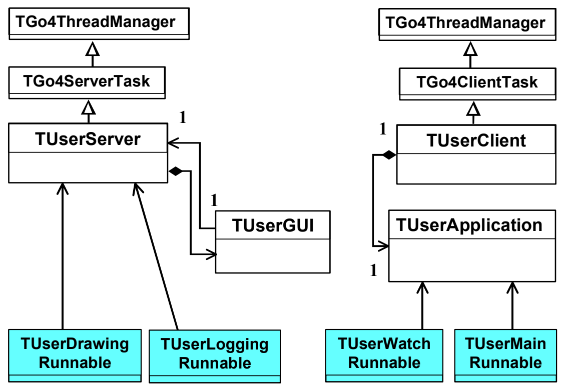


Fig. 5. Class diagram of a user client server setup implementing subclasses of the Go4 ThreadHandler framework.

It has a TUserGUI component which handles the controlling input and which may display the information fetched from the client. Pressing a GUI button, for instance, may raise a callback feeding a new command into the task handler command queue. The command is then being transported to the client and executed on the remote side (Fig. 3). Two runnable classes (TUserDrawingRunnable and TUserLoggingRunnable) specify the threaded user actions of the server: in the example the threads wait to display any client objects scheduled in the status and data queues, respectively.

B. Client Task

Similarly, the TUserClient inherits from TGo4ClientTask and from TGo4ThreadManager. It has the TUserApplication class as component, which may be a data analysis, but may as well be a control application. The client also has two additional threads working on the application (Fig. 5). The client status information is sent to the server regularly by thread TUserWatchRunnable, while thread TUserMainRunnable is doing the main work (e.g. data analysis loop) and executes the server commands scheduled in the task handler command queue.

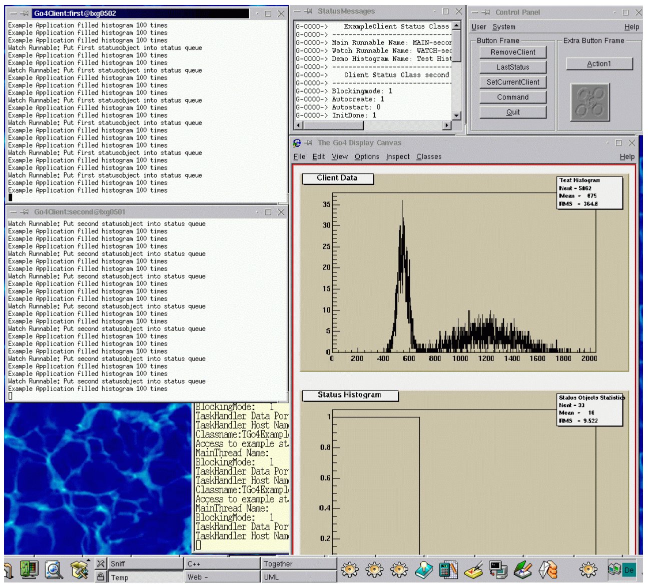
It should be pointed out that the number of additional user threads is not limited in principle, since a new thread can be added dynamically to the thread manager at any time. But there can be only one thread with unlimited functionality because of the restrictions mentioned above.

C. Go4 Analysis and GUI Example

The structure and functionality of the Go4 analysis (client) and the Go4 GUI (server) is shown in Fig. 3 and Fig. 5, a screen shot in Fig. 6. The example server has a simple GUI with a control panel, a ROOT canvas and a status window. The two server threads (Fig. 5) are responsible for independent data and status display. Pressing a GUI button, a histogram is requested from the currently selected client by command, sent to the server and drawn on the canvas (Fig. 6).

The Main runnable of the analysis client (Fig. 5) executes in the TGo4Thread loop methods of the analysis class (TUserApplication) which process one single analysis event. In addition, at the beginning of the thread loop it polls the task handler command queue and executes the waiting server commands. Hence the calculation on the data and the command execution is done sequentially by the same thread to avoid conflicts. However, some commands like killing the main thread or quitting the client are executed asynchronously by the task handler command thread itself, directly after receiving them from the command socket and without queuing.

The Watch runnable is monitoring the current analysis status independently from the main thread, i.e. it delivers information about the client to the server GUI even if the main thread in the client is stopped or killed.



```
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put first statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put first statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put first statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put first statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put first statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put first statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
```

```
Watch Runnable: Put second statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put second statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put second statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put second statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put second statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put second statusobject into status queue
Example Application filled histogram 100 times
Example Application filled histogram 100 times
Watch Runnable: Put second statusobject into status queue
Example Application filled histogram 100 times
```

```
G-0000->    ExampleClient Status Class
G-0000-> --------------------------------
G-0000-> Main Runnable Name: MAIN-secor
G-0000-> Watch Runnable Name: WATCH-sec
G-0000-> Demo Histogram Name: Test Hist
G-0000-> --------------------------------
G-0000->    Client Status Class second
G-0000-> --------------------------------
G-0000-> Blockingmode: 1
G-0000-> Autocreate: 1
G-0000-> Autostart: 0
G-0000-> InitDone: 1
```

```
BlockingMode:   1
TaskHandler Data Por
TaskHandler Host Nam
Classname:TGo4Exampl
Access to example st
MainThread Name:
BlockingMode:   1
TaskHandler Data Por
TaskHandler Host Nam
Classname:TGo4Exampl
Access to example st
MainThread Name:
BlockingMode:   1
TaskHandler Data Por
TaskHandler Host Nam
```

Fig. 6. Two analysis tasks controlled from one (very simple) test GUI.

D. *Using Qt with ROOT*

In addition to the standard ROOT widgets, we also tested the example display server together with the Qt graphics library [4]. A Qt-ROOT interface has been developed for this purpose. It showed up that the Go4 task handler is working well with a Qt GUI. Moreover, standard ROOT GUI components, such as the TBrowser [2], can be run in the same application together with the Qt GUI. This opens the option to implement a controlling display using the powerful Qt library for the GUI in connection with the ROOT based Go4 framework for the object and communication management.

V. Conclusion

The available *Go4ThreadManager* library provides a useful framework for multi-threading applications within the ROOT system. The still existing restrictions from the non-thread-safe ROOT kernel allow only one general working thread, but several dedicated threads with limited functionality.

Multi-tasking systems can be implemented using the *Go4TaskHandler* package which is based on the *Go4ThreadManager*. This available package provides services to create, manage and use independent command, data and status channels between the server and each client. By means of sub-classing, these framework services can be adopted to various user applications.

A test implementation with a preliminary GUI server and an example analysis client has in long term tests successfully demonstrated the stability of the Go4 task handler system. This client-server example has been proved to run seven client tasks on four different nodes connected to one server task. This promises a stable operation of the task handler system.

The Go4Analysis framework is to be extended in the next development phase. This phase will include the design and implementation of command classes, dynamic object organization structures, and functional patterns for a flexible and powerful GUI. Moreover, the system shall be capable of using both the native ROOT widget classes and external libraries like Qt.

Current information on further developments can be found on http://go4.gsi.de.

VI. References

[1] Status of ROOT based Analysis System Go4,
J.Adamczewski, H.G.Essel, H.Göringer, M.Hemberger, N.Kurz, M.Richter.
GSI Annual Report 1999,232, March 2000
See also http://go4.gsi.de/.

[2] *ROOT - An Object Oriented Data Analysis Framework*,R.Brun and F.Rademakers
Proceedings AIHENP'96 Workshop, Lausanne,
Sep. 1996, Nucl. Inst. & Meth. in Phys. Res. A 389 (1997) 81-86.
See also http://root.cern.ch/.

[3] *Design Patterns: Elements of Reusable Object Oriented Software*, E.Gamma, R.Helm, R.Johnson, J.Vlissides;Addison-Wesley 1999

[4] *Qt 2.30 Tutorial and Reference*, Troll Tech AS, http://www.trolltech.com/