International Journal of Networked and Distributed Computing, Vol. 5, No. 1 (January 2017) 22–36

Multi-Threaded Message Dispatcher - a Design Pattern with Innate Support for Mission Critical Applications

Marcel-Titus Marginean and Chao Lu

Computer and Information Science, Towson University
8000 York Rd
Towson, Maryland/21252, USA
E-mail: mtm@mezonix.com; clu@towson.edu

Abstract

The usage of well-tried software design patterns and application frameworks is often encountered in Mission and Safety Critical Applications development due to the high stakes involved in the case of failures. To increase reliability, some frameworks attempt to separate the implementation of business logic and low level implementation details and move the latter inside of framework-implementation in order to allow the developers to focus on the problem as much as possible, while still providing the necessary infrastructure in easy to use API's. In this paper we present a framework for message processing which takes advantage of the newer C++11 features to enforce separation of concerns, perform dead-lock avoidance, and encourage unit testing. This paper expands on our previous work presented in June 2016 at IEEE/ACIS SERA.

Keywords: Design Patterns; Critical Application; Multithreading; Message Dispatching.

1. Introduction

To implement the House Hub and House Intelligence Unit from the sDOMO system presented in our previous papers [1, 2] we designed a Multi-Threaded Message Dispatcher (MTM- Dispatcher) framework to support the processing of sDOMO Messages and Packet. The problem we addressed with this design is the typical problem of multiple messages incoming asynchronously, which is that they need to be processed by message handlers specific for each type of messages. Message processing needs to happen in parallel to take advantage of modern multicore CPUs but care has to be taken with accessing shared resources to not violating critical section discipline, while also to not creating a risk of the system entering the deadlock. The framework was designed to be reusable for other applications that require message processing, and will be offered as open-source.

Taking advantage of the lessons learned from other engineers' experience is the main reason for using well known design patterns instead of "reinventing the wheel from scratch" and running the risk of wasting time solving the same problems and making the same mistakes. A whole set of design patterns are well known in literature and we are reviewing a few related with our work.

Reactor Pattern [3] handles concurrent requests delivered to an application, by synchronously de-multiplexing them within the context of a single thread and delivering them to the appropriate service handlers. The Reactor is a very influential pattern and our MTM-Dispatcher can be viewed as a multithreading extension of it.

To handle concurrency, Monitor Object Pattern [4] synchronizes executions to ensure only one method runs within an object at any given moment in time. Active Object Pattern [5] provides each object its own thread of

control and decouples method invocation from method executions. A review of other very useful concurrency design patterns can be found in [6].

The Leader/Follower design pattern [7] addresses some of the same concerns as our Dispatcher framework, mainly: Efficient de-multiplexing of handles, threads and preventing race conditions. However not being purposely designed for the advanced template metaprogramming abilities that are available in modern C++ compilers, Leader/Follower pattern is unable to perform some safety checks at compile time, providing a strong separation of concerns and enforcing discipline for data access to aid development of safety and mission critical applications.

In "Design for Verification" [8] the authors developed a set of frameworks to aid the task of developing a system from separately verifiable parts, in order to increase the reliability of the system and therefore making it more suitable for mission critical applications.

Like our Dispatcher, the Proactor design pattern [9] takes advantage of the efficiency that can be achieved by using asynchronous I/O operations in order to speed up the processing and also proposes a design method to increase the separation of concerns in building the applications. It however does not address compile time enforcement of data access discipline and support for unit-tests for the handlers, as these features require advanced support from C++ compilers that was obviously not available at that point in time.

The core idea behind the design of this framework has been the suitability for Safety and Mission Critical Application development therefore attempting to aid with a few of the challenges encountered during application development: Deadlock Prevention, Separation of Concerns and Software Testability.

The work on this design pattern has been presented in an earlier phase in the conference paper [10] and this article just expands on the previous work providing more details and introduces some features added after the previous work has been presented at the conference.

2. Problems

The problem of having to process messages coming in concurrently from multiple devices, as illustrated in Figure 1, is a typical occurrence in software engineering and as presented in review, many design patterns have been implemented to attempt to deal with it. The most common approach has been the Reactor pattern which serializes the messages into a queue and then handles them in the context of a single thread. While Reactor is a highly successful design pattern it fails to take advantage of modern CPU's providing multiple cores, limiting it usability on modern high end systems.

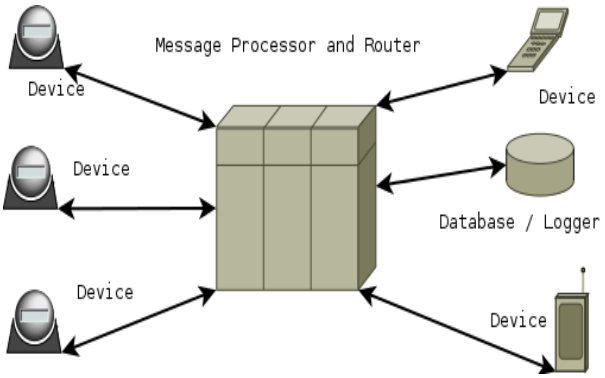

Figure 1. Multiple Message Sources and Shared Data Talking to One Message Processor

The work for the present Dispatcher design started as a multi-threaded extension of the Reactor, allowing simultaneous processing of messages in the context of multiple threads. However, in the designs that handle the processing in the context of multiple threads the *critical section problem* arises. A critical section is a fragment of code that is operating on some resource on which other threads in the system can operates too. To prevent data inconsistencies the programmers must provide synchronization methods to prevent two or more critical sections accessing the same resource to be executed simultaneously. This is achieved by synchronization primitives like Mutexes or Semaphores.

While solving the critical section problem by mutual exclusion there is the potential for another problem to be introduced, *the deadlock problem*. The deadlock occurs when two or more processes competing for the same two or more resources enter into a circular wait. The simplest example to illustrate the deadlock problem is the situation when the thread T1 already owns resource A and waits for resource B in order to do its job, while the thread T2 already owns B and waits for A which it is necessarily to be able to continue the work. The two processes will wait indefinitely making the application program non-responsive.

Having the software engineers constantly switching their attention between the business logic they have to implement and low level implementation details (for example the deadlock-prevention or validating pointers) opens opportunities for mistakes. This constant switch of attention between different domains is a known weak spot in the process of focus and is a problem that is best addressed by software design. The principle of *Separation of Concerns* recommends separating clearly the two domains, allowing the programmer to focus on and address a single class of concerns at a time.

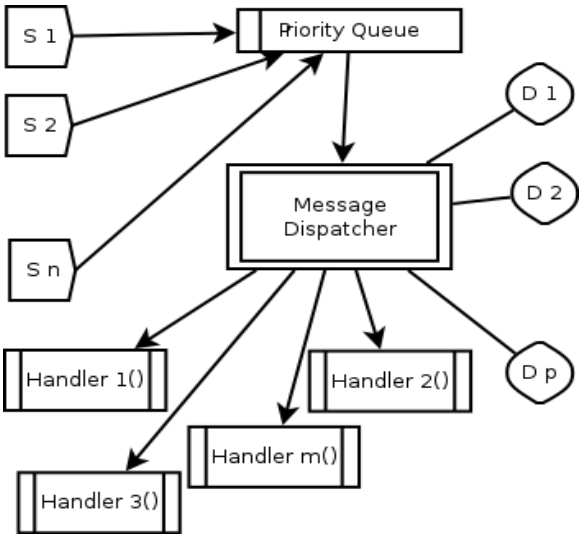

Figure 2. Dispatcher Main Components

Despite efforts in design and implementation, errors are likely to slip in during development, and software testing is the most commonly used way to detect them in order to be eliminated. Among testing methodologies *Unit Testing* emerged as a very good testing strategy allowing small units of the program to be independently put into a test harness and exercised in isolation in a controlled way. This strategy allows various combination of parameters both within the normal range, outside of it and at the boundaries to be passed to the tested fragment of code making sure that most, if not all of, the branches of the code are verified properly. Unfortunately, unit testing is neither easy nor cheap when the program was not been designed with unit testing in mind, because usually each unit makes references to other units and this increase in cascade of the complexity makes good tests harnesses notoriously hard to write.

Lastly while coding standards are usually employed to have the programmers following a certain discipline, there is usually no other way but code inspections to verify that the discipline has been respected. A framework that can enforce at least a small portion of the discipline by employing the compiler for this task it is highly desirable. If the code does not compile in case that a certain rule has been violated, the programmer will be informed right away about his or her mistake and have no other option but to fix the violation before being able to go ahead with any other work.

3. Proposed solution

While solving all of the previous problems in a general way that is suitable for any type of application program is most likely a non-achievable dream, if we focus of a certain category of problems the task of being able to implement a framework that addresses all the issues above it is a task well within the realm of possibilities. The category of problems we attempted to solve with this work came to us from the work to implement the sDOMO protocol where we have a large number of devices simultaneously sending messages that needs to be processed by a central processing entity and we also look at the typical set of problems encountered by the authors in embedded and unmanned vehicle industry.

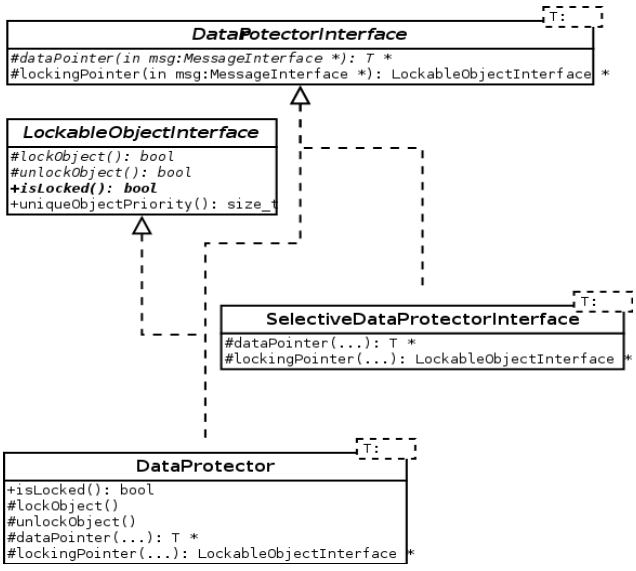

Figure 3. Class Diagram Related to Safe Data Access

These type of problems arise all over the place in embedded, networking, enterprise and business applications so a general design-pattern for a solution to them is going to have wide field of applicability.

The typical problem this framework addresses is the problem of multiple devices sending data in messages toward a central message processor (MP) which has to process the information by calling specific Message Handler Functions and eventually sending messages towards the devices in response. It addresses the problems of mutual exclusion and deadlock by a set of specific features:

- Associates a Mutex to each Shared Data Object
- Ready to use Locking/Unlocking discipline implemented by the framework
- Ready to use Deadlock Avoidance algorithm implemented inside the framework
- Compiler-enforced discipline of accessing critical shared resource

Following the system architecture presented in Figure 1 and software component architecture illustrated with Figure 2, devices are software components able to send messages; the system accepts multiple devices of the same kind as well as different kinds of devices. A "kind" of device is characterized from the types of messages that it emits and receives. Some devices can be just simple software components either one way like a logger or two ways like a database or some other type of data store, others can have real hardware backing as a camera or GPIO card. Since devices send messages asynchronous from one another and some messages can be just re-routed to other devices with little or no processing, it makes sense for the message router and processor to operate using multi-threading in order to make best use of all available CPU cores and achieve higher throughput.

Because the message processing may require access to shared data, mutual exclusion has to be implemented in order to avoid race-conditions, and whenever multiple threads and mutexes are employed there always exists the possibility of deadlock. The Dispatcher code takes charge of the locking and unlocking of the mutexes associated with the shared data however, a programmer may interfere with the deadlock avoidance algorithm by locking or unlocking the mutexes of interest from within the handlers. To prevent this from happening, we use the C++ compiler to disallow direct access from user code to shared data and their associated synchronization objects. This discipline enforcement combined with the outsourcing of synchronization problem to the framework code also aids the programmer to focus on the problem that is being solved instead of synchronization details.

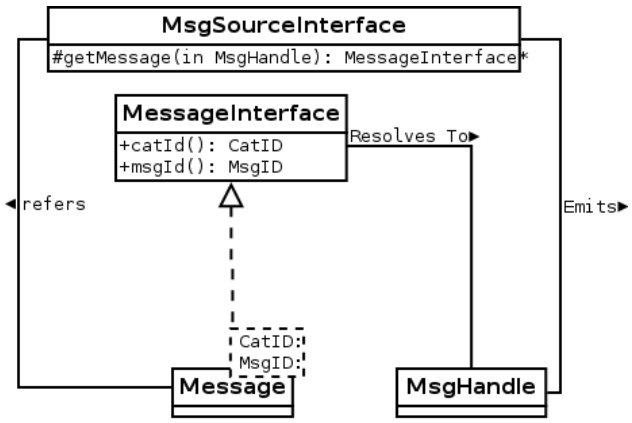

Figure 4. Class Diagram for Message Management

As depicted in the sketch from Figure 2, the main entities of the software model are a set of message sources **S1**… **Sn** which asynchronously produce Messages which the framework adds into the **Priority Queue**. The Message Dispatcher owns one or more thread which extracts the next Message (in the order of priority) from the queue. Upon successfully validating a Message, the Dispatcher looks-up all the Message Handling Entries registered for this particular message and allocates the list to a Dispatcher Thread. A Message Handling Entry consists of a Message Handler Function (**Handler 1()** ... **Handler m ()**) and a tuple of one or more references to Shared Data Objects (**D1**... **Dp**).

For each Entry, the Thread will lock the mutex associated with each Data Object using the "partial ordering deadlock avoidance algorithm" as proposed by Dijkstra as a solution to "Dinning Philosophers Problem". Once all the resources are acquired, the Dispatcher Thread calls the function handler passing a reference to Data Objects as parameters to the function.

3.1. Critical Application Support

The proposed design has a set of features to provide support for development of applications that are vital for an organization or system or for safety of people around.

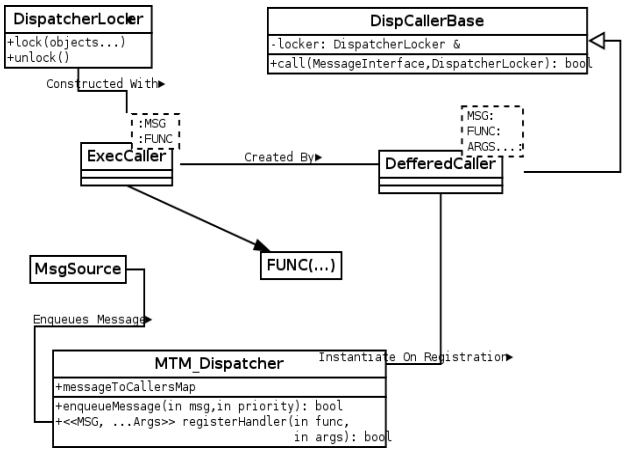

Figure 5. Class Diagram Illustrating Message Dispatching

1) Dealing with Race Conditions and Deadlock Prevention

The main goal in designing this framework was the ability to allow multi-threaded message processing while making sure the access to shared resources would never result in a deadlock situation that would make the system unresponsive and unable to perform its mission critical role. The framework implements a deadlock avoidance procedure that guarantees a deadlock-free dispatching as long as the accesses to Data Objects are non-blocking, i.e. implementing request/completion asynchronous operations.

The algorithm for deadlock avoidance works as follows:

1. All Data Objects are made inaccessible from regular user code using DataProtector template class, this makes race conditions impossible since any attempt to access a Data Object outside of the framework control results in a compiler error.
2. For each Message that needs to be handled, one or more function handlers must be declared and registered with the Dispatcher.
3. Handler registration specifies for each Handling Function the set of Data Objects that should be bound to its parameters during a call.
4. When a Message is handled, the Dispatcher will lock the Data Objects in the order of their unique locking priority, avoiding the possibility of deadlock by the partial ordering solution.
5. The references to Data Objects are retrieved by the ExecCaller object created by the Dispatcher which has a friend Relationship with DataProtectors, access their embedded data and passes it to the Handler Function as parameters.

2) Support for Separation of Concerns

Separation of Concerns is a design principle in software engineering that asserts the need to minimize the amount of time, the mind of the programmer performs context switches, like for example between high-level business logic and low-level implementation details. According to psychology studies constant context switches are a weak link in the process of focus; allowing programming errors to slip through. The presented framework design attempts to aid the programmer into the task by taking a small set of tasks on its own and enforcing others.

The fact that messages are sequenced in a priority queue guarantees that lower priority processing will not delay critical messages from being handled. Once the software engineer determines the priority of each message, either role-based or by rate-monotonic scheduling(RMS), the framework will take care of handling the proper task with the appropriate priority without further programmer's attention.

Instantiating each of the Data Objects under the control of a DataProtector prevents the programmer from accessing them directly, forcing them to rely on the framework in order to access each Data Object. This eliminates the need for the programmer to care about Critical Section problem outsourcing it to the framework. As a matter of fact, since all the handlers registered for a particular message are called sequentially under the context of a single thread, this also eliminates the need for the programmer who writes

Message Handler Functions to care about multi-threading at all. From the point of view of programmer writing handlers there is no difference between the fact that a particular handling function is called from the Multi-Threaded Dispatcher or just called from a regular function into a mono-threaded program. All the synchronization and deadlock avoidance procedures are hidden inside the framework, "out of sight out of mind," for application programmers.

The biggest support for separation of concerns brought by his framework is however the ability of Data Objects to generate messages in response to changes in the values of member data. This ability allows to implement software featuring strong separation between business logic and input/output operations as will be illustrated in the guide to implementation.

3) Support for Unit Testing

Having all shared Data Objects constructed under a Protector, forces the programmer to declare the required shared objects as parameters to the message handler function in order to be provided by the framework. As a result, all message handler's functions are self-sufficient pieces of code that can be tested individually in a test harness that just passes the required parameters to the handler subject to testing. Because the framework also takes care of all the multithreading and synchronization issues hiding this aspect from the author of the handler, all the message handler's unit-tests can be performed into a single-threaded easy to use environment.

3.2. Design Details

Two interfaces serve as the base for the Data-Protectors. LockableObjectInterface is the base for any class that the MTM-Dispatcher class is supposed to lock before calling the handler and releasing it after. DataProtectorInterface is a template abstract class parameterized with a data type that will be passed to the message handler function. The DataProtectorInterface have two protected member functions returning pointers to a LockableObjectInterface and the data type used to instantiate the template.

An auxiliary template interface SelectiveDataProtector-Interface serves as the base class for registering arrays of shared data-objects in order to pass to the handler one of them based on some information from the incoming message that is being processed.

When a handler function is being registered, the references of the classes extending DataProtector template class or SelectiveDataProtectorInterface are being passed to the registration procedure.

The framework uses a friend relationship with the protectors in order to access the methods that provide a pointer to data or associated locking mechanism.

The abstract class MsgSourceInterface is the base for all the objects that will send messages to be handled by function handlers. A message is a class inheriting a specialization of template Messages with two integer parameters, CategoryID and MessageID, and then defining their own data. Message Sources have the ability to enqueue into the Dispatcher an object of type MsgHandle which references an actual Message that needs to be sent. When the Dispatcher dequeues a message reference, it uses it to get access to the actual Message via the method getMessage() from the MsgSourceInterface. This two-step access (using a handle that resolves to message instead enqueuing a direct pointer to the message) has been implemented to address two important problems: event cancellation and messages instantiated in special memory segments.

Event Cancellation is best understood considering a time-out timer started when a request to a remote server is launched and which calls a time-out function if the answer has not been received in time. If the response is received, the reception handler will cancel the timer. However, if the queue is not empty when the reply is received, the reply message is enqueued at the end of the queue and until it will be served, it is possible that the time-out event will also be enqueued to be executed later. Without a two-step look-up, both reception and time-out handlers must perform extra accounting steps to keep track of a particular request/time-out pair since just canceling the event will have no effect on the time-out event being already enqueued as a message. With a two-step look-up, when the answer handler is processed, it cancels the timer and when the time-out event reaches the execution state, the source will just return a null message avoiding the time-out handler from being called, so event cancellation is achieved without adding any external code from the point of view of the application programmer, in direct accordance with the Separation of Concerns principle.

Dual step look-up also allows large messages to be kept in a memory managed by the Message Source itself, which can, for example, manage blocks of data in

shared or non-uniform memory blocks. When the look-up of the handler is performed, the right block can be mapped into the process address space and a pointer is returned. Enqueueing directly a pointer to the memory would require the memory to be mapped early and stay idle for the entire period the pointer is enqueued or it would require data copy into process memory. By contrast, the two-step look-up minimize the amount of time the memory segment is locked to the period when the data is effectively access improving the overall performance of multi-process systems.

With every call to the template method registerHandler of the MTM-Dispatcher a new DeferredCaller entry is added to the map indexed on the pair CatID,MsgID. The DeferredCaller entries holds the pointer to the function handler to be called and references to the data protectors associated with the data that needs to be passed to the function.

A feature that can be seen from the example in Figure 7 is the fact that the registerHandler template method is deducing the message IDs from the signature of the handler itself, the programmer not having to provide the message IDs as a separate parameter. By outsourcing the IDs deduction to the framework as opposed to the programmer we can guarantee that miss-registration is not possible, therefore preventing from the start potential system crashes due to an erroneous registration of a handler to the wrong message.

The Dispatcher starts one or more dispatching threads. Each thread runs a loop which will dequeue an MsgHandle which used to retrieve from the owning message source a pointer to the actual message. If the resolved pointer is not null, each DeferredCaller entry associated with this message is called with the message. Beside the Message pointer, a pointer to the DispatcherLocker object owned by every thread is passed along to the call(…) method of the DeferredCaller. The DeferredCaller uses the DispatcherLocker and the functor it holds to instantiate on the stack an ExecCaller functional object. The ExecCaller performs object locking in accordance to a partial ordering solution and then calls the actual function handler with the references to the data whose pointer has been retrieved from the Protectors.

The rationale for having the intermediary ExecCaller instantiated on the stack instead of allowing the DeferredCaller itself to perform locking is to assure that the same DeferredCaller can be simultaneously called from two or more threads. The rationale as to why we want that is because we have the possibility to register as handler parameter an array of data objects from which one can be selected at runtime based on the message content. If two messages resolve to the same object, the locking mechanism will be blocked and only one handler will be executed at a time, however if the messages generates separate elements of the array, the two handlers can execute simultaneously. Creating the intermediate object ExecCaller on the stack helps solve the problem in an elegant manner.

After all the handlers have been successfully called, the dispatching thread releases the Message data with the source and waits on the MsgHandler queue for the next message.

4. Recent enhancements

After the conference presentation a few featured has been added to the architecture mainly driven by the need to break with the overly simplistic model of dispatching to the same set of objects.

4.1. Registration for an Array of Objects

If a handler is registered for a message, regardless of the content of a message (other than Message ID and Category ID), the same set of Data Objects will be bound to the message handler during dispatching. It is logically however that we may have the need to dispatch to a different set of objects based on the data the message contains.

For example, the images coming from a set of cameras mounted on a house may have a camera ID field and we want to dispatch them to a particular Data Object for each camera. A naive solution will be to register the same handler for all the possible cameras with the separate object and inside the handler check if the camera id is a match and return immediately otherwise. While the naive implementation will work correctly it performances will be very poor because the framework has to lock (potential involving wait) all N-1 combinations of parameters just to have to unlock them again when the comparison fails.

To address the issue, the template class SelectiveDataProtectorInterface is inheriting the same base class template class DataProtectorInterface which is also extended by DataProtector and therefore they can be passed to the registerHandler template method of the

Dispatcher which expects as parameters a variable argument list of DataProtectorInterfaces.

The support for the selective arrays of data is provided by the template class DataProtectorInterface which defines two pure virtual methods that gets a pointer to a message as their parameter:

```
virtual T *dataPointer(const MessageInterface *)=0;
virtual LockableObjectInterface *lockingPointer(const
MessageInterface *)=0;
```

The implementation of DataProtector just ignores the parameter returning the protected data and the wrapper to the mutex respectively. However, classes derived from the template SelectiveDataProtectorInterface overrides them in order to analyze the parameter and return the appropriate values in the array.

Since the only parameter passed to these methods is a pointer to a constant object the Dispatcher is able to evaluate the values before performing any locking, therefore no unnecessary locking/unlocking delays happens, making this method the fastest selection possible.

The limitation to this approach is that all the information required to make the decision have to be present in the message and there is no way to make the selection based on the content of other objects.

4.2. Locking extra objects inside a handler

The ability to register for arrays of objects is a powerful method to increase the flexibility of the platform when the selection information is fully determined by the content of the message. However, this may not always be the case. It is likely that we will encounter situations when the need to access a different object cannot be determined until during a handler call once information from the message and the state of other objects is integrated.

One potential solution that will not need added extra support in the Dispatcher would be to temporary create a registration for a one shoot event then trigger that event. The drawbacks of this solution are that the wait time until the temporary handler is called is likely to be large into a busy system since all the queuing and handling have to start anew. Another drawback is that this solution cannot be extended to be used in safety critical applications because of the need to alter the dispatching table at run-time.

The solution is to find a way to lock the mutex associated with the new data object safely while obeying the partial ordering solution. This option is going to employ an Accessor object which uses conversion operator to present itself as a smart pointer to the object of interest. The Accessor template class have a friend relationships with the Dispatcher and the DataProtectorInterface allowing it to use the method for re-locking and unlocking.

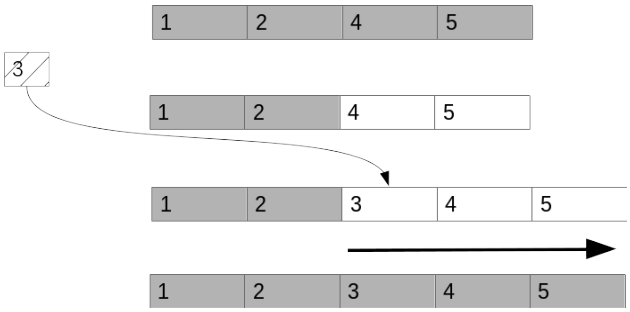

Figure 6. Adding a New Object to a Lock Set

The algorithm for locking a new object is illustrated in Figure 6 with a simple example. In it, the Locking Set already holds the objects with unique IDs 1,2,4,5 when the object with ID 3 needs to be locked. To achieve this, the Dispatcher have first to unlock all of the objects with an ID higher than the new request; in the example above it will unlock the objects with ID 4 and 5. It will then add the 3 to the locking set and re-lock all the unlocked elements in the locking set in the order of their IDs.

From the simple example above, it is easy to understand that after the partial release of the locked objects from the set, any thread waiting on the released objects but on none of the objects on which the lock is still held will get it chance to run. This has the potential to delay the re-locking process, therefore it must be understood that the object accessing with an already locked handler came at a cost in speed. However, the cost is definitely smaller than the cost of the alternate solution where all the mutexes would been released therefore allowing even more threads to pick-up and run; and on top of that we would have the overhead of registration/de-registration, queue waiting etc. Hints for implementer to minimize this delay will be provided on the next chapter. The typical usage of an Accessor is explained in the following code snippet:

```
DataProtector<int> counter2(0);


void func(const TimerMsg5 &m1,  int &i4){
   Accessor<TimerMsg5,int> integer(m1,counter2);
   if(integer)
      Log(cout)<<"func: integer="<<*integer<<endl;
}
```

4.3. Eavesdropper Handlers

It is often encountered in software engineering the need to implement loggers, monitors or recorders that intercepts the set of messages, analyze them without alteration then perform some I/O operations on the resulted data. While all these operations can be performed in regular handlers it makes sense to be able to write non-locking handlers that operates on classes of messages to do the monitoring task. To facilitate this type of solutions the Dispatcher allows to register objects inheriting the EavesdropperInterface and override the virtual method

```
void process(const MessageInterface *)=0;
```

in order to implement the specific processing. The Eavesdroppers registration happens for a whole class of messages as opposed for a specific message ID. In our implementations Eavesdroppers are called before the actual handlers gets called, but there is nothing that will restrict the order in other implementations.

5. Typical usage

The typical scenario of using the presented design pattern is in implementing applications that exhibit the semantic of parallel processing of incoming messages by a set of message handler functions. While it is true that in theory any type of application can be re-factored to exhibit this semantic, it is always a matter of judgment for the engineers and software architects to decide if the re-design is worth or not. If the change in architecture is making the code harder to understand, update and debug then the re-factoring is an obvious mistake. However, most of the applications in: Networking, Gaming, Automation, Autonomous Vehicles and Robotics exhibit an innate Message Processing semantic therefore are ideal candidates to be implemented with the presented framework. In the following paragraphs we take a closer look at the steps required to implement a Message Handling Application using the presented dispatcher framework.

REACTOR:

```
A a;
B b;
void Handle79(const Msg79 *m){
    a.foo(m->x);
    b.bar(m->y);
}

reactor.register(79, Handle79);
```

DISPATCHER:

```
DataProtector<A> a;
DataProtector<B> b;
void Handle79(const Msg79 &m,
              A &pa, B &pb){
    pa.foo(m.x);
    pb.bar(m.y);
}

disp.register(Handle79, a, b);
```

Figure 7. Typical Usage of Dispatcher vs. Reactor

The typical usage of the presented framework consists into a straight forward set of steps:

1. Design your application as a set of handlers responding of Messages emitted by Message Sources. The Messages can be not only “real messages” received from other computers over some type of network, but also events like hardware interrupts/signals or timer events or even changes in the state of some variables as a result of processing.
2. Design the Message Sources as objects providing an Asynchronous Interface. For example instead of having a Read() method that waits for the operation to complete, the object will export an BeginRead() method that initiate the process and the object will emit a Message when the operation completes and data is available. It often make sense that the BeginRead() method returns an Transaction ID to allow the application programmer to easy identify the operation upon receiving the completion message.
3. Define the set of Messages that are being processed by the application. The Framework creates messages as parametric templates with two integer parameters, named as CategoryID and MessageID allowing flexibility in mapping the messages ID coming from various device types. A Message class typically extends the

specialization of the parametric template for the two parameters, adding the member variables to hold data of interest.

4. Define all the structure of Shared Data Objects that are required to process the messages. The Shared Data usually is a C++ struct or class grouping together various pieces of data that make semantic sense when associated from the point of view of the business logic. They can also be classes that implements the Message Source Interface to be able to emit Messages in the system.
5. Write function handlers for each message, having as the first parameter a reference to the Message class, followed by references to all Shared Data Objects that need to be accessed by the function handler.
6. Instantiate each Shared Data Objects inside a Protector. Practically, declare each shared variable as a variable of the type DataProtector parametrized with the type of interest. The DataProtector constructor requires a reference to the Dispatcher to be passed as it first parameter. It is the responsibility of code reviewers to make sure that no global variables exists that are not constructed under DataProtector control unless they inherit the Active Object base class.
7. Instantiate the Message Source Servers, they also have to be written to receive a reference to the Dispatcher since it is required for construction of Message Sources Interface. All Message Source Servers must inherit Active Object base class or be constructed under the control of a Data Protector as any shared variable. If a class inherits the Active Object it is the responsibility of class designer to make sure that all the public method exhibit Monitor like behavior by providing non-blocking self-synchronization.
8. Register the handlers with the Dispatcher, passing also the list of the Protected Shared Data Variables that contains the data of interest.
9. Call the method start() of the Dispatcher.
10. Call the start() method for all the Message Sources.

The termination procedure of the program begins by calling stop() method for all the Messages Sources to insure that no new messages will be enqueued then the stop() method of the Dispatcher is to be called. Of course all Data Objects and Messages Sources shall have proper destructors to correctly clean-up the resources upon termination of the program.

The Dispatcher framework has been purposely design to enforce certain design choices, however not all of them can be enforced at the compile time and as specified above, the code reviews should be employed to aid in detecting potential mistakes. Here are a few of the potential problems that reviewers shall keep an eye onto.

- All data objects accessed by handlers must either be instantiated under the control of a Protector or be implemented as Asynchronous Active Objects. The Protector template is making sure that the programmer gets a compiler error if attempting to access data in unsafe way, but it can do nothing for variables declare outside of the framework control.
- A special consideration of the previous point is pertinent to any locking primitive. The framework guarantees that Deadlock and Critical Section problems can never be encountered, but only as long as the application programmer does not interfere with the deadlock avoidance algorithm implemented by the Dispatcher. Therefore, with the exception of Monitor behavior in Asynchronous Active Objects no other usage of locking primitives is permitted, nor necessary.
- While implementing all I/O as non-blocking Asynchronous Message Sources is not mandatory, it is highly recommended and shall be the default choice of implementing I/O. The rationale for this, is that any blocking I/O will keep locked at least one data object therefore reducing the degree of parallelism and therefore the overall performance of the

system. The framework will achieve the maximum possible throughput only if no unnecessary thread inter-locking is taking place. Asynchronous I/O solves this problem.

As already presented, it is possible for a Data Object to extend the MsgSourceInterface in order to allow Messages to be posted when certain conditions are met as a result of a handler updating data into an object. As a matter of fact, in a real life project, most of the Data Objects would probably be implemented this way allowing a clean segregation of tasks that brings the Separation of Concerns principle as “first class citizen” in our design.

More precisely, based on their intended task, message Handlers functions can be divided into three categories: Incoming Handlers, Business Logic (BL) Handlers, and Outgoing Handlers.

When an incoming message comes from an external source, the set of Incoming Handlers receive the data, validate it and unpack-it into the appropriate data objects. As a result of changing the state of data objects, they emit various business logic messages like posting an alert condition, requesting an adjustment to another value, etc. These messages are handled by the set of Business Logic Handlers which implement domain specific knowledge to assess and react to BL events. Either as a result of processing BL, by timers or other triggering sources, a set of Outgoing Request Messages are emitted which are used by the set of Outgoing Handlers to pack and send the data to external devices.

In Figure 8 a simple example of this approach is illustrated with a trivial real-time example. The two input data ports IN 1 and IN 2 receives messages from outside world and as a result sends messages that will be processed by the Input handlers IH 1 to IH 3. These message handlers after verifying the data will update the information into the set of Data Objects DO 1, DO 2 and DO 3 respectively.

The update process triggers a set of events to be fired by the Data Objects, events for which the set of Business Logic handlers BH1 to BH3 are registered for. The BH set of handlers will access the Data Objects of interest in order to perform the required calculations and update other variables of the Data Objects on their turn. For example, in the Figure 8 is illustrated the fact that BH 3 is triggered by the changes in DO 2, then access its data and updates DO 3 with the results of the processing. The DO 3 does not contain any data that is updated from input handlers, instead it contains only data processed by BH 3.

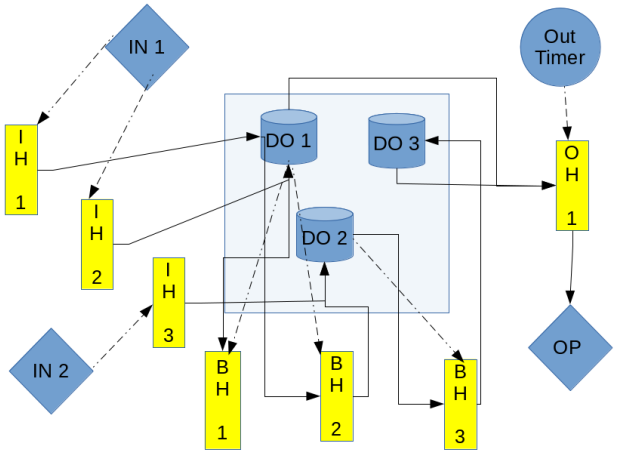

Figure 8. Taking advantage of Separation of Concerns

Finally, the output of the sample program above is generated by a real-time timer which periodically launches the output handler OH 1. OH 1 is the output data integrator collecting data from both DO 1 and DO 3 and building the outgoing message sent to the output port OP.

The clear separation between the operations of Handling External Data and Business Logic processing provides two advantages.

First, it allows different team members to focus on their specific tasks reducing the cross-domain coupling. The programmer implementing the business logic is not concerned with the format of the input or output data, nor with the hardware/protocol differences for various I/O handler objects. The only task he or she needs to be concerned about is the correctness of the business logic handlers to be implemented. The same is also valid for the programmer having to implement the input or output handlers whose only concern will be parsing and validating the data then using it to update the data objects of interest, without being concerned how, what and when that data is being to be processed.

The second advantage of this approach is the increase in code portability. Any change in the hardware, protocol or the format of In-Out data have no impact on the Business Logic code already written. Porting an existing

application to a new type of equipment requires a rewrite only of the input and output handlers without any need to tweaked or change the set of Data Objects or Business Logic Handlers.

A side effect of increased portability can be exploited to perform full testing and validation of business logic. Replacing Input and Respectively Output handlers with code that injects well known input values and validates the output allows an easy way to test the business logic without the need to perform any code change into the business logic section of the code.

An important issue to note in implementation of data sources that needs event cancellation is to make sure that in any handler needed to cancel an event must share at least a lock-able object (i.e. an regular Shared Data Object) and not only Active Objects. The rationale for this is to make sure that the handler have an object it have to wait onto. An alternative design approach will be to also make the Timers or Input Sources as Shared Objects instead of Active Objects, however this second approach may have some negative performance impact on the handlers that share many Timers/Input Sources.

An aspect that have to be kept in mind when allocating the priorities for the shared objects is the timing impact of this ordering. Since the resources priority IDs are allocated in the constructor, the order of the variable declaration is used to control the order of the IDs. Please remember that the C++ standards make no mention about the order on which global constructors on different files will be process, so it is recommended to instantiate all the Shared Objects into a single source file or as local variables in the main() of course paying attention to not exceed stack size (especially for embedded devices).

As seen from the example shown to illustrate the late-locking, the smaller the ID of the object that needs to be locked there the largest the performance penalty to be incurred by the thread performing this operation. Therefore, it make sense to declare the resources that need to be often allocated inside handlers as late as possible to insure they have highest priority IDs therefore unlocking the smallest number of resources during the re-locking process. By contrast, smaller IDs are recommended for Data Objects that are usually not shared very much between different messages to minimize the amount of time when other threads waits on them.

6. Conclusion and results

The presented framework has been used to rewrite the House Hub from sDOMO project in order to allow scalable processing of multiple devices once the original proof of concept implementation reached it limits. It is being used also in the implementation of House Intelligence Unit from the same project. It is also evaluated for being used for some support applications in unmanned aircraft industry.

The framework implements unique features for mission and safety critical applications being able to offer compile time checking of errors in message registration, enforce the usage of a deadlock avoidance protocol that guarantees the system will not lock-up due to a programming mistake, and enforce separation of concerns allowing the implementer to focuses on the problem at hand instead of low level mutual-exclusion problems. Because the framework uses handler registration, messages and shared objects that can be easily defined at any time MTM-Dispatcher framework is highly extensible and can be successfully employed in projects that are envisioned to need to scale up a lot in the future. The separation of concerns implemented by this framework allows each handler to be written as a standalone piece of code, avoiding coupling that reduces the scalability. This aspect of enforcing stand-alone handlers that are fully defined by their parameters, makes the framework highly suitable for test-driven development which is a practice highly regarded in safety critical applications.

To assess the scalability in performance of the presented Dispatcher framework, a set of tests have been run on a multiprocessor computer having 12 CPU cores. The main question to be answered by the performance testing was if the new multithreaded dispatching frameworks scale well with the number of dispatching threads. The test employed 10 Message handlers, all of them subscribing for the same message from a single message source that has been implemented both as an Active Object without the need to have the Dispatcher lock it during dispatching of the message, and respectively as Data Object requiring the Dispatcher to lock it for the duration of dispatching. There were three tests run to assess the performances.

Test #1 had the handlers printing a message then idling for the required amount of time, while Test #2 had the handlers performing CPU intensive calculations for the

same amount of time. For Test #3 we used the same handler functions as for Test #1 but the Source emitting the message to be delivered to handlers was, as of this time, a Data Object which required the Dispatcher to lock it therefore preventing other threads to run on the same time. This is a degenerated case that transformed the MTM-Dispatcher behavior in something similar with Reactor framework. For each test we run the dispatcher 32 times with a number of dispatching threads from 1 to 32 with the same workload each time.

As can be seen from the graphic in Figure 9, for the tests #1 and #2 the amount of time required to terminate the work decreased very fast until all the available CPU's cores (12) has been used by the Dispatcher. After that, the curve leveled as expected. There were no differences between the behavior of I/O and CPU intensive handlers, they took the same amount of time to complete.

By contrast, for the Test #3 where we used a Blocking Source forcing all the threads to wait for the current one holding the lock, the curve is almost flat as we would expect also from the Reactor pattern which is using a single dispatcher thread to handle all the processing. In theory, the same way as the Reactor is using a single thread to perform all the dispatching, in the degenerated case of MTM-Dispatcher we would expect the curve to be absolutely flat regardless of the number of threads employed.

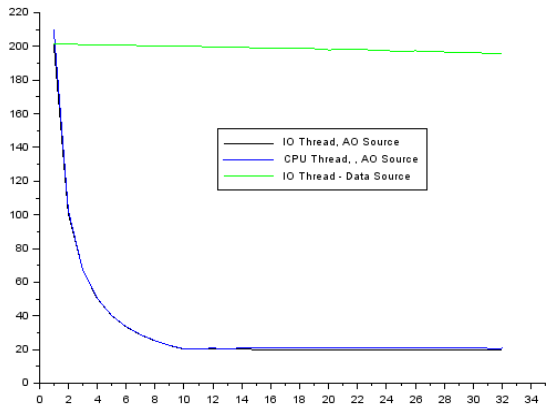

Figure 9. Illustration of the Scalability with Number of Threads

However; a closer look at the graph above shows that even for this Test #3 there is a very small improvement in performance with an increasing number of threads. The explanation for this improvement is that besides the work required to be performed by the handlers (on which the resources are locked), the Dispatcher itself has to perform some "house-keeping" overhead to manage the messages. While in the case of the Reactor pattern this overhead is executed on the same thread as the handler, in the case of MTM-Dispatcher the overhead work performed before the resources are locked and after they are unlocked takes place on a parallel thread to the one currently holding the lock and operating inside the handler. Therefore, even in the absolute worst case scenario when due to resource management our dispatcher degenerates into Reactor behavior, MTM-Dispatcher still outperforms the Reactor due to the ability to parallelize the overhead work.

In order to test the Accessor impact on the performance, we run an experiment featuring five handlers and five types of messages that have been sent to the handlers. Each Message M_i triggered the handler H_i at a frequency of 5 Hz. Handlers 1 to 4 each had their own counter that incremented at each call and they also performed a delay of 100ms before termination. The fifth handler has been implemented in two versions. The version A registered for all four counters to display one of them based on the timer message, then perform an 100ms delay. The version B registered for the counter number four only, then based on the timer message decides which one of the remaining three counters is to be displayed using an Accessor. It waits 50 ms before using the Accessor to lock the picked object then waits another 50 ms with the object locked, the expected total sleep duration is 100 ms in this case as well.

The fact that the version B of the handler locked only has an extra counter therefore allowing two of the other threads to run while it is executing its body is expected to show some performance improvements. The experiment has run twice with the version A and with B respectively, for each handler one run had the Counter-4, locked used always, at the highest and respectively the lowest priority. The time on which the test program completed on each case are presented in the table below:

Total Hnd 5	**Counter-4 Highest**	**Counter-4 Lowest**
Hnd-5 A	39.86 s 100.28 ms	39.76 s 100.28 ms
Hnd-5 B	33.40 s 116.76 ms	30.09 s 168.14 ms

The measurements show that selective locking is more efficient and therefore validates the creation the Accessor template class as a good design decision to improve the overall performance. It also highlights that using the Accessor increases the total time spent in the handler by 16 ms and respectively 68 ms. None of these results are unexpected, however we have to note the significant variation between both total processing times (33 s to 30 s) and the handler execution time (116 ms to 168 ms) based on the priority assigned to the shared objects. As predicted, better overall performance of the system are being achieved at the expense the speed of the handlers using Accessors. This makes clear that the effort to design the priority allocation of shared resources can pay big dividends in a real life project.

The Reactor design pattern [1] has been used for over 20 years to implement countless projects in mission critical applications and will still be used for a long time for application where mono-threading dispatching is preferred. Today however, due to the advancements in C++ language, on multi-core systems we are able to provide a better alternative that not only outperforms it in every aspect but also improves the safety and speed of code development by compiler enforcement of the separation of concerns.

Future work to develop this pattern may include usage of Readers-Writers locking pattern to optimize the dispatching even further by allowing multiple handlers to run simultaneously if they share only constant data objects. It is also required some research work to investigate whether we can allow multiple parallel handling for some message or not. Another envision enhancement will be to export a Dispatcher interface as an Data Object allowing handlers to safely modify the dispatching table as needed at run-time. This enhancement will increase the flexibility of the Dispatcher for Mission (non-Safety) Critical applications. For Safety Critical applications this is forbidden by rules and certification policies.

One drawback of this framework it is that like any registration based framework, the process of debugging an application written as handlers is more difficult because the framework itself interposes between the message source and the code for handling it. The recommended debugging procedure for the user code will be to put break-pointers on the handler code for a particular message and use logging to trace the message presence at the source. However, due to the development aid offered by the framework to the application writers, we hope that the amount of debugging required for any application implementing this framework is much lower than without it.

Another drawback to this Dispatcher framework is that it requires C++ techniques that are available only in the compiler that implements the C++ 2011 standard and newer, while the Reactor can be implemented in any older dialect of C++ language and even in less evolved languages like Java, C or Ada. Unfortunately, the industry for mission and even more so safety critical applications are very slow in accepting new versions of the language.

There is, however, a follow-up effort to research Java Reflection technique as a potential means to provide help in porting a “light-weight” version of the Dispatcher to Java. We are also planning to try exploring the potential for a “light-weight” implementation in Python. A “light-weight” implementation will not have parameter checking for handlers at compile time on registration statement, but will throw a run-time exception if a mismatched registration is encountered at dispatching time. Therefore “light-weight” implementations may be undesirable for Safety Critical applications.

As of this moment and in the foreseeable future, due to required strong compiler support for templates, C++11 and newer editions are the only languages in which a full-featured (“heavy-weight”) MTM-Dispatcher framework can be implemented. Once the adoption of C++11 becomes mainstream in Mission/Safety Critical software development, this drawback will no longer exist.

References

1. Marcel-Titus Marginean and Chao Lu, “sDOMO – A Simple Communication Protocol for Home Automation and Robotic Systems”, IEEE International Conference on Technologies for Practical Robot Applications; May 11 – 12, 2015.
2. Marcel-Titus Marginean and Chao Lu, “sDOMO in the context of Internet of Things”, International Conference on Computer Science, Technology and Applications; March 18 – 20, 2016.
3. Douglas C. Schmidt, “Reactor – An Object Behavioral Pattern for De-multiplexing and Dispatching Handles for Synchronous Events”, 1995.
4. R. Greg Lavender and Douglas C. Schmidt, “Monitor Object - An Object Behavioral Pattern for Concurrent Programming”, 1996.
5. R. Greg Lavender and Douglas C. Schmidt, “Active Object – An Object Behavioral Pattern for Concurrent Programming”, 1996.
6. Ifran Pirali et al.,”Patterns for Efficient, Predictable, Scalable, and Flexible Dispatching Components”. 7th Pattern Languages of Programs Conference (PLoP '00) in Allerton Park, Illinois, August 2000. Addison-Wesley, 2000.
7. Douglas C. Schmidt et al., “Leader/Follower A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching”. PLoP 2000, http://www.cs.wustl.edu/~schmidt/PDF/lf.pdf.
8. Peter C. Mehlitz, John Penix, “Design for Verification, Using Design Patterns to Build Reliable Systems”. Proceedings of the Sixth ICSE Workshop on Component-Based Software Engineering., 2003.
9. Proactor – An Architectural Pattern for Demultiplexing and Dispatching Handlers for Asynchronous Events - Pyarali, Harrison, et al. 1997.
10. Marcel-Titus Marginean and Chao Lu, “Multi-Threaded Message Dispatcher Framework for Mission Critical Applications”. IEEE/ACIS International Conference on Software Engineering Research, Management and Applications, June 2016.