Multi-Threaded Message Dispatcher Framework for Mission Critical Applications

Marcel-Titus Marginean
Computer and Information Science
Towson University
Towson, Maryland, USA
mtm@mezonix.com

Chao Lu
Computer and Information Science
Towson University
Towson, Maryland, USA
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 as much as possible on the problem to be solved while providing the necessary infrastructure into 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.**

Keywords—Design Patterns; Critical Application; Multithreading; Message Dispatching;

I. Introduction

To implement the House Hub and House Intelligence Unit from sDOMO system presented in our previous papers [1, 2] we designed a Multi-Threaded Message Dispatcher (MTM-Dispatcher) framework to support Messages and sDOMO Packet processing. The framework was designed to be reusable for other applications that require message processing, and will be offered as open-source in the upcoming release of sDOMO reference implementation.

Taking advantage of the lessons learned from other engineers' experience is the main driver 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 in 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 framework to aid the task of developing a system from separately verifiable parts, in order to increase the reliability of a system and therefore making it more suitable for mission critical applications.

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

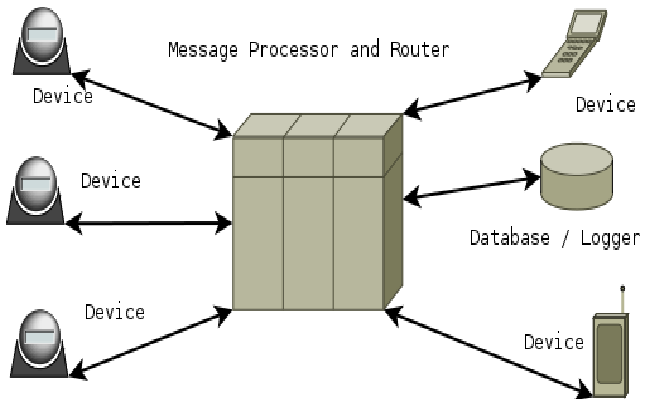

Figure 1 Multiple Message Sources and Shared Data Talking to One Message Processor

challenges encountered during application development: Deadlock Prevention, Separation Of Concerns and Software Testability.

II. PROBLEMS

The problem of having to process messages coming in concurrently from multiple devices 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 pattern it fails to take advantage of modern CPU's providing multiple cores.

In the designs that handle the processing in the context of multiple threads the *critical section problem* arises and the programmers must provide synchronization methods to protect the critical sections. While solving the critical section problem another problem can be introduced, the risk of entering in dead-lock where two or more threads are caught in a circular wait.

Having the software engineers constantly switch their attention between the business logic and the implementation details (for example the deadlock-prevention or validating pointers) opens opportunities for more mistakes. The principle of *Separation of Concerns* recommends separating clearly the two, allowing the programmer to focus on and address a single class of concerns at a time.

Despite efforts in design and implementation, errors are likely to slip in, and software testing is the most commonly used way to detect them in order to eliminate errors. 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 independently in a controlled way. 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 makes good tests harnesses notoriously hard to write.

III. PROPOSED SOLUTION

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 and eventually send messages toward the devices in response. It addresses the problem of mutual exclusion and deadlock by imposing a compiler-enforced discipline of accessing any critical shared resource. In Figure 1, the main architecture is presented.

The 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. 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 CPU cores and achieve higher throughput.

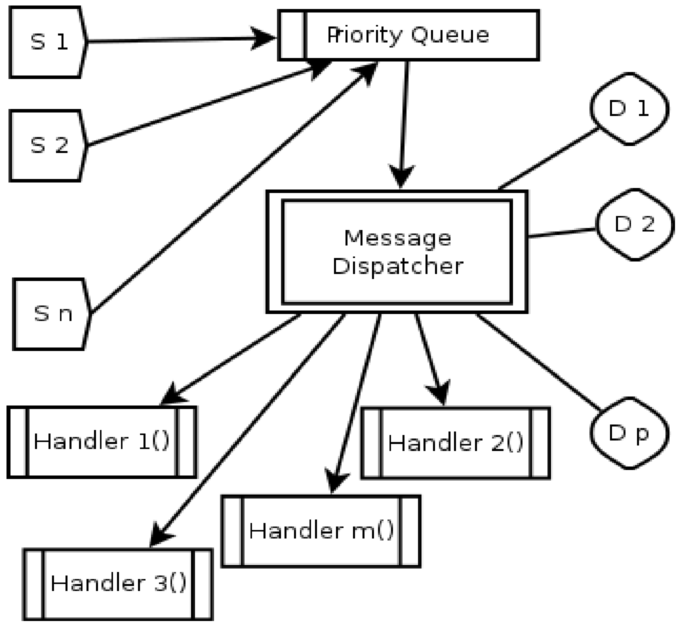

Figure 2 Dispatcher Main Components

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. To prevent this to happen, we use the C++ compiler to disallow direct access from user code to Shared Data and put the framework in charge of synchronization. This also aids the programmer to focus on the problem to be solved instead of synchronization details.

As depicted in the sketch from Figure 2, the main entities of the system 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.

A. 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.

1) Dealing with Race Conditions and Deadlock Prevention

The main goal in designing this framework was the ability to allow multithreaded 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.

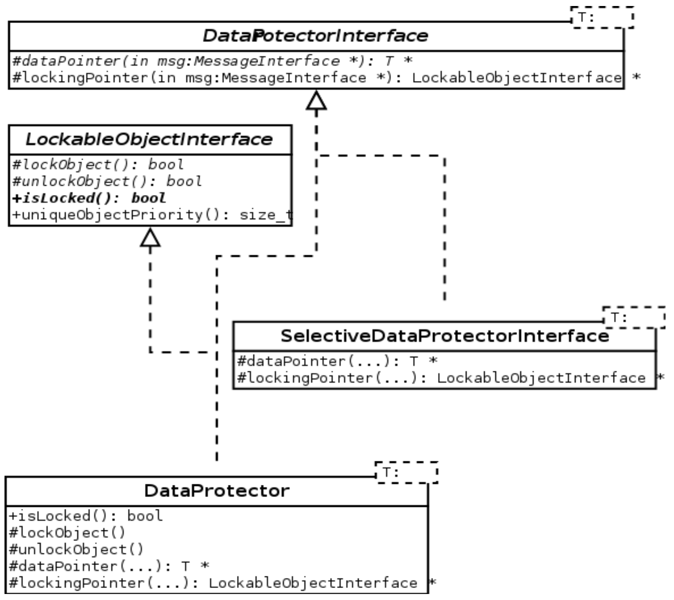
DataPotectorInterface
T:
#dataPointer(in msg:MessageInterface *): T *
#lockingPointer(in msg:MessageInterface *): LockableObjectInterface *
LockableObjectInterface
#lockObject(): bool
#unlockObject(): bool
+isLocked(): bool
+uniqueObjectPriority(): size_t
SelectiveDataProtectorInterface
T:
#dataPointer(...): T *
#lockingPointer(...): LockableObjectInterface *
DataProtector
T:
+isLocked(): bool
#lockObject()
#unlockObject()
#dataPointer(...): T *
#lockingPointer(...): LockableObjectInterface *

Figure 3 Class Diagram Related to Safe Data Access

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 determined the priority of each message, either role-based or by RMS, the framework will take care of handling the proper task with the appropriate priority without further programmer's attention.

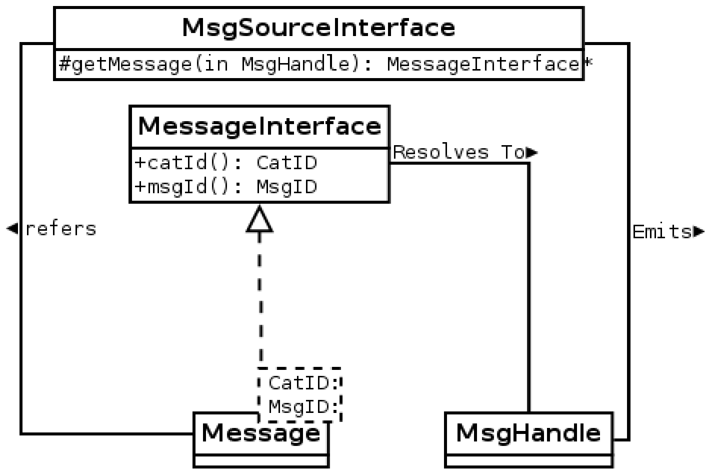

Figure 4 Class Diagram for Message Management

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 multithreading 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 Multithreaded 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.

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.

B. 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 SelectiveDataProtectorInterface 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 an instantiation of template Message 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 is launched and which calls a time-out function if the answer has not yet 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 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 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 t/o 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.

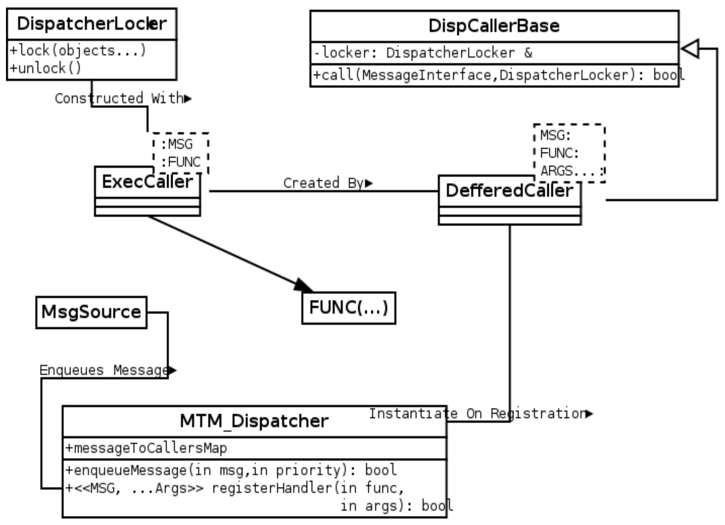

Figure 5 Class Diagram for Message Dispatching

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 handle 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.

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.

The Dispatcher starts one or more dispatching threads. Each thread runs a loop which will dequeue an MsgHandle and from the owning message source a pointer to the actual message is retrieved. 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 values 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.

IV. TYPICAL USSAGE

Using the MTM-Dispatcher framework to implement a Message Handling Application consists of a set of standardized steps:

1. Defining the Messages that are being processed by the application: The Framework defines 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.
2. Define all the structure of Shared Data Objects that are required. The Shared Data usually is a C++ struct element grouping together various pieces of data that make semantic sense when associated from the point of view of the business logic.
3. Write function handlers for each message, having as the first parameter a reference to the Message class and followed by references to all Shared Data Objects that need to be accessed by the function handler.
4. Instantiate Shared Data Objects inside a protector as Protected Data Object variables.
5. Write Message Source Servers as Active Objects inheriting MsgSourceInterface.
6. Instantiate Message Sources.
7. Register the handlers and the corresponding protected Data Objects instances with the Dispatcher.
8. Call the method start() of the Dispatcher.
9. Call the start() method for all the Message Sources.

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 matter of fact, most of the Data Objects would probably be implemented this way allowing a three step process that brings the Separation of Concerns principle as "first class citizen". More precisely, 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 will just receive the data 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 BL set of Handlers which implement domain specific knowledge to assess and react to BL events. Either as a result of processing BL or by timers, a set of Outgoing Request Messages are emitted which are used by the Outgoing Handlers to pack and send the data to external devices. The clear separation between the operations of Handling External Data and Business Logic processing allows different team members to focus on their specific tasks reducing the cross-domain coupling.

V. 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 performance of the MTM-Dispatcher, 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.

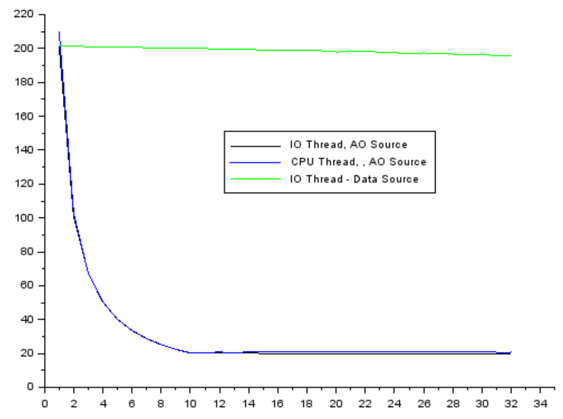

Figure 6 MTM-Dispatcher Performance Graph (lower is better)

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 6, 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.

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.

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, we are able to provide a much better alternative that not only outperforms it in every aspect but also improves the safety and speed of code development by strong 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.

The drawback to this Dispatcher framework is that it requires advanced 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. 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 runtime exception if a mismatched registration is encountered at dispatching time. Therefore "light-weight" implementations may be unsuitable 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 became 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.