Introduction to Message Passing and the MPAPI Framework

Covers MPAPI version 1.1

mpapi@sector0.dk

http://sector0.dk

Revision 3

Table of contents

Preface

How do we write secure, robust, scalable, mutilthreaded software? There are several approaches, and no definitive answer to this question. But the recent advent of multicore processors have put a new emphasis on how we as programmers write software that can utilize this new hardware architecture so that the processors are not idle and we get maximum performance.

Traditionally we write sequential programs. That is what programmers are taught, and it is (relatively) easy to understand the flow in the software. Programmers are also taught that they should avoid multithreading whenever possible, because chances are that bugs will be present – indeed one gets the feeling that bugs are inherent in multithreaded software. And the fact is that getting multithreading right in imperative languages like C++, Java or C# is extremely hard; the primitives provided in these languages for handling threads and their synchronization are not adequate. It looks simple on paper, but writing multithreaded software is rarely as trivial as the textbook examples.

The problem is shared state. All threads have access to the same memory locations so several threads can concurrently read and update the same memory. That is a recipe for disaster, as you no doubt realize. The solution to this is using locks, either as semaphores, monitors, mutexes or some other construct. But using locks can be very hard to get right – race conditions and deadlocks lurks right beneath the surface. As Simon Peyton Jones from Microsoft Research put it in the book ”Beautiful code” by Greg Wilson, ”locks are bad”. Either you take too few locks, too many lock, take the wrong locks, or take them in the wrong order. Furthermore, error recovery is very hard, because the programmer has to guarantee that any error does not leave the system in an inconsistent state. I have been a developer and architect on several enterprise level systems with a panic solution that restarts the application if an unknown error occurs and we cannot guarantee the state of the system. That is not by choice as much as by necessity – the complexity of systems rapidly grow out of control when dealing with multithreading.

About Message Passing

Message passing is a different approach to concurrency than we are used to when using modern, imperative languages like C++, C# or Java. In those languages we use *shared state concurrency*, where all threads in the same process space has access to the same areas of memory (i.e. variables etc.). This present a problem with how we synchonize the threads to avoid inconsistent memory, and it is usually done with locks, semaphores, monitors or other constructs. As explained in the preface this is not as trivial as one might think, and there are always one or more problems in the synchonization that can be very hard to detect, leading to unstable systems and a lot of time spend on debugging and rewriting parts of the software.

In *message passing concurrency* each thread has access only to its own state. This helps programmers write more robust software since the need to synchonize memory no longer exists. The only means for a thread to communicate with other threads is by sending them messages [1], and state is not transferred between threads in these messages.

[1] This is not entirely true in MPAPI; since the framework is written in C# it is still possible to break the non-shared state if one really wants to, but that would violate the primary design of the system.

This approach also has the advantage of working perfectly in a distributed environment, since the simplest solution to letting processes and threads on physically disparate computers communicate is by using message passing. This is already being done in existing frameworks – MPI (Message Passing Interface), MS-MPI (Microsoft Message Passing Interface), Beowulf clusters and others – used in high performance computing (HPC).

About Message Passing API (MPAPI)

I wrote the MPAPI framework for 3 reasons:

- I was looking for a way to simplify the development of multithreaded applications without having to worry about locks and other thread synchronization mechanisms. The problems with thread synchronization are becoming more and more apparent as we move from single core to multicore CPUs. The majority of software today is written with a single core in mind, and hence does not fully utilize the multiple cores in modern CPUs.
- I spend a couple of weeks investigating the functional programming language Erlang. Erlang has from the beginning been designed with parallel and distributed computing in mind. The solution the engineers at Ericsson came up with is very simple and elegant, and I wanted to investigate how this would work in a .NET context.
- For some time I have been researching genetic algorithms and genetic programming. This is a particularly compute intensive branch of computer science, and I wanted a framework that would easily let me write software that could scale on multiple computers.

The framework is now in a state where it can be released, although it is still in some kind of flux as I gain more and more experience with developing software with it.

Terminology

- **Worker** : A worker is the main entity of the MPAPI framework. This is where the programmer specific code is written, and a worker has all the primitives needed to communicate with other workers in the cluster. A worker runs in its own background thread inside a node, and receives messages in an asychronous manner through its mailbox. A worker has an id which is unique within the node. Together with the node id this identifies the worker in the cluster.
- **Node** : A node is the entity that runs one or more workers. The node is responsible for dispatching messages to the right receivers, and for maintaining connections to all other nodes in the cluster. There is typically one node per computer, and each node has some primitives that enables Worker-code to access information about the cluster which it is part of. Each node is assigned a unique id within the cluster.
- **Message** : Messages are the means by which workers communicate. A message consists of:
    - An Id, which is unique for each message. This is mainly used to trace messages through the cluster, and should be disregarded.
    - A message type which distinquishes it from other types.
    - A message level, which indicates if the message has been send by the system of from user code.
    - Content, which is the payload of the message. This is used to transfer state to and from workers.
- **Registration Server** : This server is used to register and unregister nodes in the cluster. It is also responsible for notifying existing nodes when a node registereds or unregisteres. The registration server is distributed as a standalone executable with the MPAPI framework, but is not necessary to start if the software runs only one node on a local machine.

Architecture

One of the design goals was to provide programmers with a simple interface to write multithreaded, single-computer applications using message passing. Since the majority of software is not distributed, this was a core design principle.

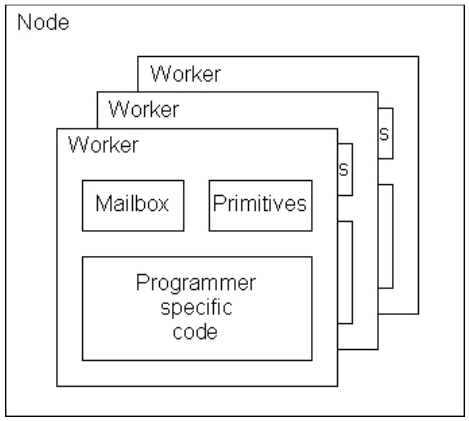

Figure 1: A local node with multiple worker threads.

But since message passing is very useful when controlling applications on multiple computers, and since some systems require more computing power than is ever possible with a single computer, I designed the distributed logic into the framework as well. In the MPAPI framework this provides the programmer with a slightly modified set of primitives, but basically he or she is unaware of the distributed aspect.
A cluster build with MPAPI consist of a main node, which controls the cluster, a number of sub nodes, which are the real work horses of the cluster, and a registration server. The registration server binds the cluster together by allowing nodes to register and unregister with the cluster, and existing nodes in the cluster is notified of such events. Communication between nodes is not handled by the registration server, but directly from node to node.

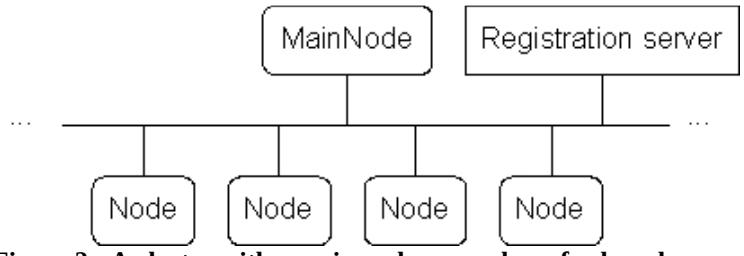

Figure 2 - A cluster with a main node, a number of sub nodes, and a registration server.

The registration server is a standalone executable which is distributed with the framework.

An application written in MPAPI implements a number of workers. Each worker communicates with each other through the node. The messages are not propagated down through the Remoting layer unless two communicating workers are on two different nodes.

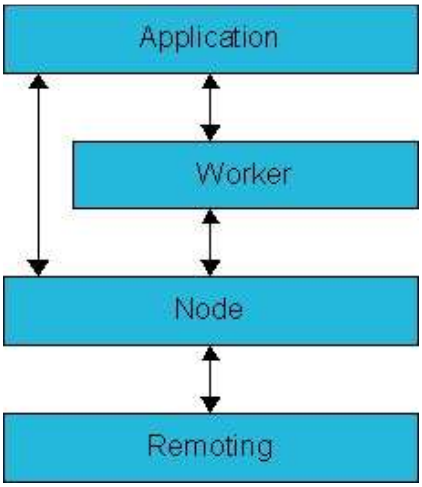

Figure 3 : The layered structure of a MPAPI application.

This asynchronous design enables programmers to write more robust in the likes of a micro kernel.

A micro kernel is a special operating system (OS) design, where the OS itself it nothing more than a message layer; drivers and everything that comprise an OS are implemented as separate entities that communicate through messaging. Linux, Unix and Microsoft Windows are examples of operating

systems that are *monolithic*, not *micro kernels*. MK design is an old concept, but lately even Microsoft has realised the potential to write robust and secure operating systems with this approach – their newly released *Singularity* research operating system, that showcases how such operating systems might (or should?) be implemented in the future, is based on a micro kernel design.

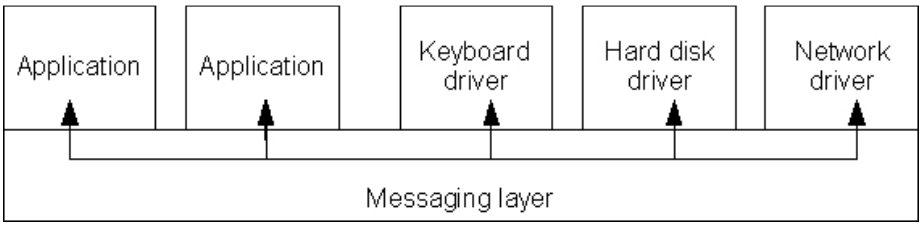

Figure 4 : A micro kernel OS, where everything - applications, drivers, etc. - communicates via message passing.

Design goals

The following goals where formulated prior to starting development.

- The framework must be compatible with both Microsoft.NET and Mono.NET. That is why I wrote the RemotingLite framework [2].
- Writing single-computer or multiple-computer (cluster) applications must be equally simple. Making an application function in a distributed environment must be trivial, and the framework must not function differently from a programmers perspective when in local- or distributed mode.
- Performance is paramount. Thus I have gone to great lengths to optimize all aspects of the framework, from serialization of messages, to controlling threads that perform all the asynchronous operations of the framework.
- The programming paradigm must be simple, the number of primitives small, and the approach must be somewhat similar to what is done in the functional programming language Erlang. Simplicity is a key design goal.
- Software systems written with MPAPI must be highly reliable, scalable and robust. This is accomplished by preventing errors from any given worker to affect all other workers – a crashing worker will only notify about it, if another worker is monitoring it.

Setting up a system

Writing an application using MPAPI is very easy, or at least the initial part is. There are two modes a node can operate in – local and distrributed mode.
In the following I will give examples taken from the two small example applications that can be downloaded together with this document. Those applications are **`PrimeCalculatorApp`** and **`DistributedPrimeCalculatorApp`**. Each one calculates the number of prime numbers between 2 and another number (specified in the class **`MainWorker`**) using a rather naïve algorithm for testing primality. This algorithm is not the most effective – certainly there are other and more

[2] RemotingLite is a small framework for writing service oriented, distributed applications. It borrows ideas from Windows Communication Foundation (WCF), but is not compatible with any other service standards. RemotingLite runs on both Microsoft and Mono .NET runtimes.

effective algorithms as well as statistical algorithms – but it is good enough for demonstration purposes.

Running the examples

The examples provided here are both a local version and a distributed version. When running the distributed example you need to start the executables up in the following order:

1. **`RegistrationServer.exe`**
2. **`DistributedPrimeCalculatorApp.exe`** – choose "slave" mode.
3. **`DistributedPrimeCalculatorApp.exe`** – choose "master" mode.

If you want to run the example on several machines you need to start all the slaves up before starting the master.

Note: You may need to alter the values for the port numbers and IP addresses in the configuration files related to the registration server and application to fit your own network setup and firewall settings.

Local node

In local mode the node will not attempt to connect to a registration server, or any other node. This operating mode can be used in applications that are not required to run in a distributed mode, i.e. a normal client- or server application.

To open a node, you first start by creating it. After that you only need to call the method **`OpenLocal<TRootWorker>()`**, which opens the node and starts a background thread running the worker specified by the **`TRootWorker`** type in this generic method. **`TRootWorker`** must inherit from **`Worker`** in one way or the other.

```
static void Main(string[] args)
{
  using (Node node = new Node())
  {
    node.OpenLocal<MainWorker>();

    /* Since the node spawns new workers in separate threads, and as
     * background threads, we have to prevent the main thread from
     * terminating here. */
    Console.ReadLine();
  }
}
```

Here we define the node in the entry point (**`Main(string[])`**) to a console application.

Note that we use **`using(…)`** to define the node. That is not strictly necessary, but **`using`** ensures a call to **`Dispose()`** when the node is no longer used. Calling **`Dispose()`** is important, *especially* when opening the node in distributed mode, since this will do a bit of cleanup.

Distributed node in a cluster

Opening a node in distributed mode will make it connect to a registration server, and afterwards all other nodes already registered in the cluster. This mode can be used in servers farms or compute clusters, where a lot of computing power and parallel processing is required. This next example shows how to open a number of slave nodes and one controlling (master) node. I use the same executable for both master and slave, but there is nothing that prevents you from writing separate applications.

We start again by opening the node in the main entry point for a console application.But first we must gather a bit of information about what operating mode (master or slave) and what port number this particular node will accept incoming connections on. Next is the address and port number for the registration server. This has been specified in the following configuration file since this is the same for both master and slave nodes.

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <add key="RegistrationServerAddress" value="192.168.1.2"/>
    <add key="RegistrationServerPort" value="8000"/>
  </appSettings>
</configuration>
```

The entry point of the application would then look something like this:

```
static void Main(string[] args)
{
  //get the variable parameters needed
  string modusOperandi = "";
  while (modusOperandi != "m" && modusOperandi != "s")
  {
    Console.Write("[m]aster or [s]lave > ");
    modusOperandi = Console.ReadLine();
  }
  Console.Write("This nodes port number > ");
  int port = int.Parse(Console.ReadLine());

  //Get the information in the config file
  string regServerAddress =
      ConfigurationManager.AppSettings["RegistrationServerAddress"];
  int regServerPort =
      int.Parse(ConfigurationManager.AppSettings["RegistrationServerPort"]);

  using (Node node = new Node())
  {
    if (modusOperandi == "m")
      //open the node in master mode and start the main worker
      node.OpenDistributed<MainWorker>(regServerAddress, regServerPort, port);
    else
      //open the node in slave mode
      node.OpenDistributed(regServerAddress, regServerPort, port);
    //prevent the main thread from terminating
    Console.ReadLine();
  }
}
```

Note here that we use one of the two **`OpenDistributed`** methods; when starting a distributed node we must specify whether or not we want a worker to start up right away. In the case of a slave node we will let the master node (**`MainWorker`**) decide when and where to spawn slave workers.

Each node that registers with the registration server is assigned an id, which is a number (of type **`ushort`**) between **`0x0000`** and **`0xFFFE`**. **`0xFFFF`** is part of a broadcast address, so there can be 65,535 nodes in a cluster. Each node can have 65,535 workers running, where the worker id **`0xFFFF`** is the last part of the broadcast address. More on that later.

That is what is needed to start a node in either local or distributed mode. As you can see there is not much difference between the two modes. Starting with a single computer application does not mean that you have to rewrite a lot of code in order to make it part of a cluster.

A note on OpenLocal and OpenDistributed

The two methods, **`OpenLocal<TRootWorker>()`** and **`OpenDistributed<TRootWorker>(string, int, int)`** both return **`TRootWorker`**. This way you can get the instance of a worker that is automatically started up.

Defining a worker

A worker can be thought of as a thread. In fact the framework will spawn a new background thread for each worker that is spawned.
You start by subclassing the type **`Worker`**, which is defined in the framework as an abstract class. In this example we have the type **`PrimeWorker`**, which receives batches of numbers from the **`MainWorker`** instance in the main node, and tests them for primality.

```
public class PrimeWorker : Worker
{
  public override void Main()
  {
  }
}
```

The **`Main()`** method has to be overridden, and is the main entry point of a worker. There are other optional methods to override, which I will get back to later.
What we want this worker to do is enter a message processing loop. When it gets a **`Start`** message it will immediately send a **`RequestBatch`** message to the main worker to get a batch. The main worker will reply with a **`ReplyBatch`** message which will hold an instance of the **`Batch`** class in its content. All numbers between **`Batch.Minimum`** and **`Batch.Maximum`** is tested for prime numbers in a class called **`PrimeFinder`**. This class will count the numbers of prime numbers in the current batch, and send the result back to the main worker with a **`Result`** message. After that the prome worker requests a new batch to test.

As programmer you define the message types yourself. In this example I have specified them like this:

```
public class MessageTypes
{
  public const int Terminate = 0; //tells the prime worker to stop
  public const int Start = 1; //initialize the prime workers
  public const int ReplyBatch = 2; //the main worker sends a batch of numbers
  public const int RequestBatch = 3; //the prime worker requests a new batch
  public const int Result = 4; //the prime worker sends the count back
}
```

and the **`Batch`** type looks like this:

```
  [Serializable]
  public class Batch
  {
    private long _minimum;
    private long _maximum;

    public Batch(long minimum, long maximum)
    {
      _minimum = minimum;
      _maximum = maximum;
    }

    public long Maximum
    {
      get { return _maximum; }
    }

    public long Minimum
    {
      get { return _minimum; }
    }
  }
```

Notice that all types that are sent as content in a message must be serializable, regardless of whether or not the message is send to a local worker or across a TCP/IP network to a remote worker; all messages are serialized to prevent shared state, and if the content cannot be serialized the framework will flag that as an error and continue.

The **`PrimeFinder`** class looks like this:

```
public class PrimeFinder
{
  public void CountPrimes(long min, long max, WorkerAddress returnAddress)
  {
    long count = 0;
    for (long primeCandidate = min; primeCandidate <= max; primeCandidate++)
      if (IsPrime(primeCandidate))
        count++;
    Worker.Current.Send(returnAddress, MessageTypes.Result, count);
  }

  public bool IsPrime(long primeCandidate)
  {
    //two is by definition a prime number, 1 is not however
    if (primeCandidate == 2)
      return true;
    //throw an exception if the number is less than 2
    if (primeCandidate < 2)
      throw new ArgumentException("Prime candidates cannot be less than 2");
    //throw away even numbers
    if ((primeCandidate % 2) == 0)
      return false;

    long max = (long)(Math.Sqrt(primeCandidate) + 1);
    for (long divisor = 3; divisor < max; divisor += 2)
      if ((primeCandidate % divisor) == 0)
        return false;
    return true; //it is a prime
  }
}
```

Since this class does not inherit from **`Worker`** we have to access the current worker in order to send a message. This is done with the static property **`Worker.Current`**, which provides an interface to the current worker.

Important: The current worker is found via the current thread context. So calling **`Worker.Current`** from a thread that you have created yourself will only yield **`null`**.

We can now write the code for the **`PrimeWorker.Main()`** method:

```
public override void Main()
{
  PrimeFinder primeFinder = new PrimeFinder();
  Message msg;
  do
  {
    msg = Receive(); //block and wait for incoming messages
    switch (msg.MessageType)
    {
      case MessageTypes.Start:
        //request the first batch to process
        Send(msg.SenderAddress, MessageTypes.RequestBatch, null);
        break;

      case MessageTypes.ReplyBatch:
        //We have receives a batch from the main worker. Process it
        Batch batch = (Batch)msg.Content;
        primeFinder.CountPrimes(batch.Minimum,
           batch.Maximum,msg.SenderAddress);
        //request the next batch to process
        Send(msg.SenderAddress, MessageTypes.RequestBatch, null);
        break;

      default:
        //we do not care about all other messages
        break;
    }
  }
  while (msg.MessageType != MessageTypes.Terminate);
}
```

The main worker (**`MainWorker`**), which is responsible for a number of things:

1. Spawn as many **`PrimeWorker`** instances on each node (except its own) as there are available processors.
2. Initialize all prime workers by broadcasting a **`Start`** message.
3. Send new batches of numbers to test until we have reached the limit.
4. Send a **`Terminate`** message to all workers if there are no more batches to test.
5. Keep score of the number of primes found so far.
6. Terminate if there are no more prime workers running.

```
public override void Main()
{
  //set up some preconditions
  long max = 9999999;
  long batchSize = 50000; //the number of numbers to test in each batch
  long currentMinimum = 2;
  long primeCount = 0;

  //spawn workers on each remote node
  List<WorkerAddress> workers = new List<WorkerAddress>();
  List<ushort> remoteNodeIds = Node.GetRemoteNodeIds();
```

```
  foreach (ushort nodeId in remoteNodeIds)
  {
    // Spawn a prime worker for each processor/processing core
    int processorCount = Node.GetProcessorCount(nodeId);
    for (int i = 0; i < processorCount; i++)
    {
      WorkerAddress worker = Spawn<PrimeWorker>(nodeId); //here we specify
                                                         the exact node id
      workers.Add(worker);
      Monitor(worker); //receive messages when the worker terminates
    }
  }

  //Initialize all workers
  Broadcast(MessageTypes.Start, null);

  //start listening for requests and results
  Message msg;
  do
  {
    msg = Receive();
    switch (msg.MessageType)
    {
      case MessageTypes.RequestBatch:
        //a worker has requested the next batch
        //check if there are any left to process
        if (currentMinimum <= max)
        {
          long currentMax = currentMinimum + batchSize;
          currentMax = currentMax > max ? max : currentMax;
          Send(msg.SenderAddress, MessageTypes.ReplyBatch,
                      new Batch(currentMinimum, currentMax));
          currentMinimum = currentMax + 1;
        }
        else
          Send(msg.SenderAddress, MessageTypes.Terminate, null);
        break;
      case MessageTypes.Result:
        primeCount += (long)msg.Content;
        break;
      case SystemMessages.WorkerTerminated:
        //a worker terminated - remove it from the list
        workers.Remove(msg.SenderAddress);
        break;
      case SystemMessages.WorkerTerminatedAbnormally:
        //a worker terminated abnormally - remove it from the list
        workers.Remove(msg.SenderAddress);
        break;
      default:
        break;
    }
  }
  while (workers.Count > 0); //keep listening until all workers are terminated
}
```

All primitives used in this example will be explained in the next chapter.

Basic primitives

This chapter gives an introduction to the most basic primitives used in an application using MPAPI.

Spawning a new worker

WorkerAddress, broadcast address

Getting a list of registered nodes in the cluster

Each worker has a reference to the node in which it is running through the **`Node`** property. Amongst other things you can get a list of ids of all the remote nodes in the cluster by calling the method **`List<ushort> GetRemoteNodeIds()`**on this property.

Getting the number of processors available to a node

The **`Node`** property also gives you two methods for getting the processor count available to each node:

- **`int GetProcessorCount()`**, which gives you the number of processors available to the local node.
- **`int GetProcessorCount(ushort)`**, which gives you the number of processors available to the node with the specified id. If the id equals the local nodes id, then the result is the same as calling **`GetProcessorCount()`**.

Sending a message

Unicast

Sending a message to a specific worker requires you to know the exact address of this worker. A **`WorkerAddress`** comprise of a node id and a worker id, which uniquely identifies the worker. The **`WorkerAddress`** class has one constructor which takes both ids as arguments.
The **`Send`** primitive has the following signature:

```
void Send(WorkerAddress receiverAddress, int messageType, object content)
```

where **`messageType`** is used to distinguish different types of messages, and **`content`** is the payload of the message. **`content`** must be serializable, but otherwise there are no restrictions.

Limitations on address and message type

As previously mentioned, there can be 65,565 nodes in a cluster, each containing a maximum of 65,565 workers. The address **`0xFFFF,0xFFFF`** for node id and worker id respectively is reserved for broadcasting.
The message type is an integer that the programmers specify. Allowed values are in the range [0 ; 2,147,483,647], both included, which should be adequate for most purposes. Values smaller than 0 are reserved for system message types, defined in the class **`SystemMessages`**.

Broadcast

It is also possible to broadcast messages to all workers in the cluster using the **`Broadcast`** primitive which has the following signature:

```
void Broadcast(int messageType, object content)
```

Receiving a message

The Receive primitives

There are 4 overloaded **`Receive`** primitives in MPAPI. Each one will block the current thread if there are no messages that fulfills their respective search criteria, and is only unblocked when a new message arrives. The blocking behaviour is done using wait handles so the thread will not use any CPU cycles while it is not necessary. All **`Receive`** primitives uses the mailbox as a FIFO-stack so the oldest messages are processed first.

The overloaded primitive has the following signatures:

`Message Receive()`
Returns when there is any message in the mailbox.

`Message Receive(int messageType)`
Returns when there is at least one message with the specified message type.

`Message Receive(WorkerAddress senderAddress)`
Returns when there is at least one message from the specified sender.

`Message Receive(WorkerAddress senderAddress, int messageType)`
Returns when there is at least one message from the specified sender, and with the specified message type.

The HasMessages primitives

While the **`Receive`** primitives will block the worker until a message fulfills the criteria, it is sometimes necessary to check the mailbox without blocking the thread. This is done with the **`HasMessages`** primitives, which has the following signatures:

`bool HasMessages()`
Returns **`true`** if there is any message in the mailbox.

`bool HasMessages(int messageType)`
Returns **`true`** if there is at least one message with the specified message type.

`bool HasMessages(WorkerAddress senderAddress)`
Returns **`true`** if there is at least one message from the specified sender.

`bool HasMessages(WorkerAddress senderAddress, int messageType)`
Returns **`true`** if there is at least one message from the specified sender, and with the specified message type.

Filtering on incoming messages

If you are not a bit careful when writing the message loops that fetches messages from the inbox you might end up with a congested mailbox. Failing to empty the mailbox properly will quickly fill it up, and the framework will warn you when there are more than 10,000 unprocessed messages waiting for any given worker. This is not in itself dangerous, just bad style.
To further ensure that this will not happen you can use the **`SetMessageFilter`** primitive to block any unwanted messages. The filter operates on message types, and the primitive has the following signature:

`void SetMessageFilter(params int[] filters)`

This way you can call it with a variable number of inputs. Each time you call **`SetMessageFilter`** the original filters are deleted.

Simulating synchronous behaviour

All this message passing is done asynchronously – immediately after sending a message the worker continues without worrying about the dispatching and possible replies to this message. So how do we perform synchronous "calls" in the framework? Normally this might indicate a wrong design in the application since message passing is asynchronous by nature, and to maintain maximum performance normal message passing behaviour is encouraged. But this is the real world, and experience dictates that all the best plans are often revised.

The following example shows how to simulate a synchronous call to another worker.

```
private int SquareValue(int a, WorkerAddress address)
{
  /* The following message types should be defined in a separate class.
   * This is only to show how to use message passing to call a method
   * synchronously */
  int msgType_RequestSquareValue = 20; //request a worker to square a value
  int msgType_ReplySquareValue = 21; //the result from the operation

  //send the request to square the integer a
  Send(address, msgType_RequestSquareValue, a);

  /* Block the worker until a reply with the correct message type
   * is received from the worker */
  Message msg = Receive(address, msgType_ReplySquareValue);

  //return the squared value
  return (int)msg.Content;
}
```

You have to be careful, though, that the two message types are not used for anything else. Although the sequence in which messages are received and processed is guaranteed by the framework, it cannot prevent the code from sending messages in the wrong order.

Monitoring another worker

One of the design goals of the MPAPI framework is the ability to write robust software. This in itself is a grand goal, and if a programmer is dedicated enough it is possible to make an entire distributed application crash. But one thing I have learned over the years is that proper exception handling is only a theory – exception handling is *always* spotty, and more than once has this resulted in the system crashing with some esoteric error message hidden in several gigabytes of logfiles on the server.
Errors, either bugs in the code or faulty data, exists in all systems. One of the key design features in MPAPI is that any worker crashing does not interfere with others, or crashes the entire system. This is done by encapsulating the worker in a **`try-catch`** statement that will catch any unhandled exceptions that might be thrown in the code. That, together with the fact that state is not shared between threads, makes it hard to crash a system written using MPAPI.
But preventing an error from propagating through the system and bringing it to a halt, is not in itself useful if we are unaware of it. For that reason you have the **`Monitor`** primitive available. This primitive binds one worker to another so that the monitoring worker will receive a special message when the monitored worker terminates, either normally or abnormally. That enables a programmer to take the appropriate action – either restart the worker or do something else - and the system can continue to work without causing too much trouble for users, or aborting a big computation.

The **`Monitor`** primitive has the following signature:
`void Monitor(WorkerAddress monitoree)`
and takes as argument the address of the worker which we want to monitor.

The message types that are received when a worker terminates are defined in the **`SystemMessages`** [3]class. If the monitored worker terminates abnormally due to an unhandled exception, the monitoring worker will receive a **`SystemMessages.WorkerTerminatedAbnormally`**, and the exception that caused the error is in the **`Content`** property of the message. A normal termination will result in a **`SystemMessages.WorkerTerminated`** message.

See the example code for the class **`MainWorker`**, where I have shown an example of how to monitor other workers, and a possible example of how to react when a worker terminates.

[3] All system messages has the message level set to MessageLevel.System, whereas normal messages has the level set to MessageLevel.User.

Using the log

No large system can function properly without logging information about errors and other informating in order to weed out bugs or trace functionality, especially multithreaded and distributed systems. The MPAPI framework has a build in thread safe log you can use if you do not want to use **`Console.WriteLine()`** or one of the major logging frameworks like NLog and Log4Net.

The **`Log`** class in MPAPI has a number of static thread safe methods with the following signatures:

`static void Info(string formattedMessage, params object[] arguments)`
Logs a message if the **`Log.LogLevel`** properrty is set to **`LogLevel.Info`**.

`static void Warning(string formattedMessage, params object[] arguments)`
Logs a message if the **`Log.LogLevel`** properrty is set to **`LogLevel.Warning`**.

`static void Error(string formattedMessage, params object[] arguments)`
Logs a message if the **`Log.LogLevel`** properrty is set to **`LogLevel.Error`**.

`static void Debug(string formattedMessage, params object[] arguments)`
Logs a message if the **`Log.LogLevel`** properrty is set to **`LogLevel.Debug`**. This is typically enabled when trying to trace bugs in the system.

Each log-method takes a **`formattedString`** argument. This, together with the variable number of arguments, is used exactly like a **`string.Format(…)`** call. For example, the call **`Log.Info("The string '{0}' is {1} characters long", str, str.Length)`** will give the following output if **`str`** is "A string":

```
Info | The string 'A string' is 8 characters long
```

Consult the MSDN library for further information.

The log can be customized. You can use the **`Log.LogType`** property to set *how* the log treats log messages. The **`LogType`** enumeration can be one of the following values:

- **`LogType.None`** : Nothing is logged
- **`LogType.Console`** : The log writes messages to standard out (usually the console window).
- **`LogType.File`** : The log writes messages to a file. The file is located in the same directory as the executable running the node, and has the filename **`<executable>.exe.log`**.
- **`LogType.Both`** : The log will write message to standard out as well as the log file.

All values can be OR'ed, but **`LogType.Console | LogType.File`** is the same as **`LogType.Both`**. The default value for **`Log.LogType`** is **`LogType.Console`**.

The **`Log.LogLevel`** property is used to set *what* is logged. The **`LogLevel`** enumeration can have one of the following values:

- **`LogLevel.None`** : Nothing will be logged.
- **`LogLevel.Info`** : A message is logged if **`Log.Info`** is called.
- **`LogLevel.Warning`** : A message is logged if **`Log.Warning`** is called.
- **`LogLeve.Error`** : A message is logged if **`Log.Error`** is called.
- **`LogLevel.InfoWarningError`** : A message is logged if **`Log.Info`**, **`Log.Warning`** or **`Log.Error`** is called.
- **`LogLevel.Debug`** : A message is logged if **`Log.Debug`** is called.
- **`LogLevel.DebugCore`** : This level is used by the framework to trace messages through the entire system. ***Warning:*** **`LogLevel.DebugCore`** generates a *lot* of information, and can severely degrade performance!

`LogLevel` values can be OR'ed to provide different functionality. The default value for **`Log.LogLevel`** is **`LogLevel.InfoWarningError`**.

An example of how to set the log type and log level to something else than the default values is this:

```
static void Main(string[] args)
{
  Log.LogLevel = LogLevel.Info | LogLevel.Debug;
  Log.LogType = LogType.Both;
  using (Node node = new Node())
  {
    ...
  }
}
```

Troubleshooting

Problems with Mono.NET

One of the design goals of MPAPI was, that it should be compatible with both Microsoft.NET and Mono.NET, and this has been accomplished. But I have experienced some problems when trying to run the binaries compiled with Microsofts compiler on the Mono.NET runtime, and vice versa. The framework *can* run on both platforms, so I suggest you compile the source with Monos gmcs compiler if you plan to run it in Monos runtime. The binary package available in the release contains binaries compiled with both compilers, and there is a **`mono_build.bat`** file available for both MPAPI, RemotingLite and the example code.

Heterogenous clusters vs. homogenous clusters

A cluster is made from two or more computers. If those computers are not similar in hardware there will be differences in the number of workers they can handle, and the execution time of an algorithm. Think this into your design.
For example, one computer can have an older processor with hyperthreading technologi. This will make the node report back that it has two processors, while in fact it has only one.

Too many or too large messages congest the network / framework

When designing a system with any kind of message passing – not just MPAPI - as synchronization mechanism the mantra is “small and few messages, big computations”. Remember that two nodes in two different processes share no memory, and a message has to be serialized, sent and deserialized, even between two workers in the same node (to prevent shared state, and threads locking each other). The problem will be greater if the communication has to take place over a network.

Since messages are the life blood of such a system I have gone to great lengths to improve performance of these operations. But nevertheless, regardless of messaging framework, it is not as fast as direct method invocation on an object in the same process space. Even the topology and general usage of your network can have an impact.

If performance is not as expected you might try to fine tune your hardware (network, primarily), or the system by redefining how often a message is sent. It might improve performance to lower the rate of sending, but with bigger payloads in the individual messages.

All primitives

Overridables in a Worker

When subclassing **`Worker`** you have a few options regarding methods to override in order to receive specific notifications. The worker entry point is mandatory to implement.

`public abstract void Main()`
This is the main entry point to a worker, and it is mandatory to implement.

`public virtual void OnWorkerTerminated()`
This method is called when the current worker terminates correctly.
It is optional to override this method.

`public virtual void OnWorkerTerminatedAbnormally(Exception e)`
The method is called when the current worker terminates due to an unhandled exception. The exception that caused the error is an argument to the method.
It is optional to override this method.

`public virtual void OnRemoteNodeRegistered(ushort nodeId)`
This method is called when a new node is registered in the cluster.
It is optional to override this method.

`public virtual void OnRemoteNodeUnregistered(ushort nodeId)`
This method is called when a node unregisters from the cluster, perhaps for a software update.
It is optional to override this method.

Opening a Node instance

A node can be opened in 3 different ways using the following methods:

Method	Arguments	Return value	Explanation

`OpenDistributed(`
`  string registrationServerNameOrAddress,`
`  int registrationServerPort,`
`  int port)`
	• **`registrationServerNameOrAddress`** : The IP address or DNS hostname of the computer the reg. server is running on.
	• **`registrationServerPort`** : The port number of the registration server.
	• **`port`** : The port number the node will accept incoming connections on.
		`void`	Opens a node in distributed mode withou spawning a worker.

`OpenDistributed<TRootWorker>(`
`  string registrationServerNameOrAddress,`
`  int registrationServerPort,`
`  int port)`
	• **`TRootWorker`**: The type of worker to spawn at startup.
	• **`registrationServerNameOrAddress`** : The IP address or DNS hostname of the computer the reg. server is running on.
	• **`registrationServerPort`** : The port number of the registration server.
	• **`port`** : The port number the node will accept incoming connections on.
		`TRootWorker`	Opens a node in distributed mode, and starts a worker of type **`TRootWorker`**.

`OpenLocal<TRootWorker>()`	• **`TRootWorker`**: The type of worker to spawn at startup.	`TRootWorker`	Opens a node in local mode, and starts a worker of type **`TRootWorker`**. The node will not attempt to connect to the registration server or any other nodes.

Primitives available from a Worker subclass

These are the primitives defined in the **`IWorker`** interface.

Primitive	Arguments	Return value	Explanation
`Broadcast(` **`int messageType,`** **`object content)`**	• **`messagetype`** : The type of message. • **`content`** : The contents of the message.	**`void`**	Broadcasts a message to all workers.
`HasMessages()`		**`bool`**	Checks if there are any messages waiting.
`HasMessages(int messageType)`	• **`messageType`** : The type of message to filter on.	**`bool`**	Checks if there are any messages with the specified message type waiting.
`HasMessages(WorkerAddress senderAddress)`	• **`senderAddress`** : The sender address to filter on.	**`bool`**	Checks if there are any messages with the specified sender address waiting.
`HasMessages(` **`WorkerAddress senderAddress,`** **`int messageType)`**	• **`senderAddress`** : The sender address to filter on. • **`messageType`** : The type of message to filter on.	**`bool`**	Checks if there are any messages with the specified sender address and message type waiting.
`Id`	**`None – property`**	**`Ushort`**	Gets the Id of the worker.
`Monitor(WorkerAddress monitoree)`	• **`monitoree`** : The address of the worker to monitor.	**`void`**	Binds this worker to the worker with the specified address, so this worker will receive notifications if the monitored worker terminates.
`Node`	None, it is a property	**`IWorkerNode`**	Gets an interface to the local node.
`Receive()`		**`Message`**	Gets the first message in the queue. The thread will be blocked if there are none.
`Receive(int messageType)`	• **`messageType`** : The type of message to filter on.	**`Message`**	Gets the first message in the queue with the specified message type. The thread will be blocked if there are none.
`Receive(WorkerAddress senderAddress)`	• **`senderAddress`** : The sender address to filter on.	**`Message`**	Gets the first message in the queue with the specified sender address. The thread will be blocked if there are none.
`Receive(` **`WorkerAddress senderAddress,`** **`int messageType)`**	• **`senderAddress`** : The sender address to filter on. • **`messageType`** : The type of message to filter on.	**`Message`**	Gets the first message in the queue with the specified sender address and message type. The thread will be blocked if there are none.

Primitive	Arguments	Return value	Explanation

`Send(`
`  WorkerAddress receiverAddress,`
`  int messageType,`
`  object content)`
	• **`receiverAddress`** : Address of the receiving worker.
	• **`messageType`** : The type of message.
	• **`content`** : The contents of the message.
		`void`	Sends a message with a specific type and content to another worker.

`SetMessageFilter(params int[] filters)`	• **`filters`** : A variable list of message types.	`void`	Sets the message filters on the current worker. This will prevent messages with a message type not in the filter list to arrive in the mailbox.
`Sleep(int ms)`	• **`ms`** : The number of milliseconds to sleep.	`void`	Forces the worker thread to sleep for a period of time.

`Spawn(`
`  string workerTypeName,`
`  ushort nodeId)`
	• **`workerTypeName`** : The fully qualified name of the type of worker to spawn.
	• **`nodeId`** : Id of the node on which to spawn the worker.
		`WorkerAddress`	Spawns a worker with the specified fully qualified name on the specified node.
			A fully qualified name is of the format "Namespace.SubNamespace.ClassName, AssemblyName". See MSDN documentation on **`Type.GetType(string)`** for further information.
			Use this if the target worker type is not known in the project holding the current worker.

`Spawn<TWorkerType>()`	• **`TRootWorker`** : The type of worker to spawn.	`WorkerAddress`	Spawns a worker of the specified type on the local node.
`Spawn<TWorkerType>(ushort nodeId)`	• **`TRootWorker`** : The type of worker to spawn. • **`nodeId`** : Id of the node on which to spawn the worker.	`WorkerAddress`	Spawns a worker of the specified type on the specified node.

Primitives available from the local node

These primitives on the local node is available from a worker through the **`Worker.Node`** property, which will return an instance of the interface **`IWorkerNode`**.

Primitive	Arguments	Return value	Explanation
`GetId()`		`Ushort`	Gets the Id of the node.
`GetIPEndPoint(ushort nodeId)`	• **`nodeId`** : The id of the node.	`IPEndPoint`	Gets the end point of the specified node. From the end point you can get IP address and port number.
`GetProcessorCount()`		`int`	Gets the number of processors/cores available to the local node.
`GetProcessorCount(ushort nodeId)`	• **`nodeId`** : The id of the node.	`int`	Gets the number of processors/cores available to the specified node.
`GetRemoteNodeIds()`		`List<ushort>`	Gets a list of ids of remote nodes that has been registered in the cluster.
`GetWorkerCount()`		`int`	Gets the number of workers running on the local node.
`GetWorkerCount(ushort nodeId)`	• **`nodeId`** : The id of the node.	`int`	Gets the number of workers running on the specified node.

Accessing the current Worker instance from anywhere

Usually you will not restrict you code to be inside a worker instance. This is bad object oriented design, and the worker classes will get unnecessarily large.
So to access the current worker from anywhere in the code there is a static property on the **`Worker`** class that will return the instance of the current worker.

```
public static IWorkerNode Current {get; }
```

Note : The framework will try to match the current thread context against a worker instance. So if you call **`Worker.Current`** from within a thread you yourself have spawned, a **`null`** value is returned.

Registration server

The MPAPI framework comes with a registration server readily available in the **`RegistrationServer.exe`** executable. But this does not mean that you have to use this secondary application. If you want to bundle the registration server with for example the master node application this is possible.
The **`RegistrationServer.exe`** executable does nothing more than create an instance of the **`RegistrationServerBootstrap`** class in the namespace **`MPAPI.RegistrationServer`**. It has a single method – **`Open(int)`** – which takes a port number as an argument. The following code is from the **`RegistrationServer.exe`** executable and shows how to use it:

```
static void Main(string[] args)
{
  int port;
  string sPort = ConfigurationManager.AppSettings["Port"];
  if (!int.TryParse(sPort, out port))
  {
    Console.WriteLine(string.Format("Cannot parse '{0}' as an integer", sPort));
    return;
  }
  RegistrationServerBootstrap registrationServer =
      new RegistrationServerBootstrap();
  registrationServer.Open(port);
}
```

If you plan to instantiate the registration server within another application, remember to do it *before* instantiating any nodes so that the nodes can register properly.

The **`RegistrationServerBootstrap`** class implements the **`IDisposable`** interface. Remember to call the **`Dispose()`** method, either explicitly or implicitly with the **`using`** keyword, in order to properly clean up. This is not done in the above example since the application terminates when the call to **`Open(port)`** returns.