

# Using AppDomains to Build Reliable Systems

by Pinku Surana

---

Building reliable systems is very hard because it is an *all or nothing* game. If there exists even one unhandled exception, the entire system is considered unreliable. This is an impossible standard to meet for a large system. In fact, the first step to building reliable systems is to accept that it is impossible. Instead, we will attempt to build a more reliable system from a collection of smaller unreliable components. The idea is to manage failure rather than pursue an impossible perfection.

The motivation for this article is a programming language called Erlang (<http://www.erlang.org>). It was originally designed to build soft real-time systems at Ericsson, a telecommunication company. Erlang is a dynamically-typed functional concurrent programming language. The primary feature of Erlang is one-way message passing between local and/or remote lightweight processes. A lightweight process in Erlang is a very efficient way to construct small, isolated, concurrent programs. Communication is handled by sending discrete messages asynchronously over a channel to another Erlang process. The runtime will move the messages very efficiently if the receiver process is on the same machine; otherwise, it will serialize and send the message over the network to a remote process. By forcing concurrent programs into this restrictive pattern, it reduces concurrency errors and promotes parallelism.

Erlang makes use of these lightweight processes to build reliable programs. These “processes” have similar isolation properties as normal OS processes: protected memory, concurrency, and failure in one process will not bring down the whole system. That last property can be used to build “supervisor” processes, which are responsible for launching and monitoring one or more worker processes. If a worker process fails, the supervisor will restart the worker. Therefore, the entire system will continue to operate even if there are intermittent failures in some components. If the component fails repeatedly, the supervisor can launch a simpler process instead to hopefully keep the system up and running.

To implement this idea in .NET, we can use .NET AppDomains to simulate Erlang processes. AppDomains in .NET are like cheap managed processes that run on the same instance of the CLR. In .NET v1 and v1.1, an unhandled exception in an AppDomain would fail quietly without bringing down the entire application. Unfortunately, this behavior was removed in .NET v2; therefore, an unhandled exception in any AppDomain (really, any thread) will bring down the entire application. We no longer have process isolation and the entire application is vulnerable to a flaw in any part of a program. In addition to unhandled exceptions in the normal control flow of a program, one must be careful about asynchronous exceptions like `StackOverflowException` and `OutOfMemoryException` that can occur at any time.

For each worker process that one wishes to load into a separate AppDomain, the main supervisor program creates an instance of a Channel which supports reliable messaging between AppDomains. Like Erlang, messages are enqueued concurrently but processed sequentially. The Channel loads the assembly into a new AppDomain, creates an instance of the receiver type, and offers an `Enqueue` method so other processes can send messages to this process.

```

// in assembly: supervisor
// do foreach worker process
Channel cq = new Channel("worker1", "Receiver1") ;
channels.Add("worker1", cq.Enqueue) ;

```

The Receiver1 class inherits from MarshalByRefObject because we use remoting to cross AppDomain boundaries. Remoting between AppDomains on the same CLR is very fast. This class will be in a different assembly. The class Receiver1 has a simple method to get an input message. It also has a method that initializes the worker and gets a mapping of process names to delegates, which send messages to the corresponding process. So if Receiver1 wants to send messages to Receiver7 or 8, it grabs those delegates from the dictionary.

```

// in assembly: worker1
public class Receiver1 : MarshalByRefObject, IReceiver {
    public void OnStart (Dictionary<string, Send> channels) {
        // initialize system and grab the channels it needs
    }
    public void ReceiveMessage (object msg) {
        // process the message
    }
}

```

The implementation of channels in .NET is an ad-hoc message queue within the same process. Essentially, it is a queue that is safe for concurrent writes, because many processes may write to the same queue concurrently. To make this a little easier to use, the Enqueue method on the queue class will be wrapped in a Send delegate.

```

public delegate void Send (object msg) ;
public class Channel {
    string assemblyName, typeName ;
    AppDomain ad ;
    IReceiver r ;
    Thread channelMonitor = new Thread(Dequeue) ;
    public void Enqueue (object msg) {
        // carefully push the msg on the queue
    }
    void Dequeue () {
        // block until the queue is nonempty
        try {
            r.ReceiveMessage(internalDequeue()) ;
        } catch (Exception e) {
            AppDomain.Unload(ad);
            LoadAppDomain();
        }
    }
    void LoadAppDomain () {
        ad = AppDomain.CreateDomain(assemblyName);
        r = (IReceiver)newAD.CreateInstanceFromAndUnwrap

```

```
        (assemblyName, typeName) ;
    }
    public Channel (string a, string t) {
        assemblyName = a ;
        typeName = t ;
    }
}
```

The channel creates another thread to monitor the queue for new messages. The code is elided here, but it basically blocks until new messages arrive on the channel. Then it dequeues a message and calls `ReceiveMessage`. Finally, here's the code that makes the system more reliable. If `ReceiveMessage` fails with an unhandled exception, the `Dequeue` method catches the exception, unloads the "broken" `AppDomain` and reloads a fresh copy of it. Unfortunately, this code matches the semantics of Erlang: if the process fails while processing a message, the process is restarted but that particular message is lost.

The purpose of this design is to build a system that roughly emulates the reliability feature in Erlang. The main program (aka supervisor) creates a channel for every worker process, allocating a new `AppDomain` and connecting to a `Receiver` object via remoting. Worker processes do not contact each other directly; instead, they are loosely connected by a "reliable" communication channel, similar to a message queue. If an `AppDomain` should fail catastrophically for any reason, the channel catches the error, unloads the errant code and reloads the `AppDomain`. Though this design will not magically make your application 100% reliable, it will help your application survive the inevitable errant, uncaught exceptions that plague all systems.