Stiv Remote Commander Overview:

# Background

The Remote Commander was an exercise to overcome a problem I had managing large numbers of servers. My basic problem was that I needed to run commands on large numbers of servers, and I needed to get the results back. There were multiple solutions I found, but none worked the way I wanted.

* They could not work across Domain Boundaries
* They did not provide a status in an easy to view format if at all
* They were unreliable, or required components that might not be installed.
* They did not provide results in near real-time.
* They ran sequentially, not in parallel.

# Genesis

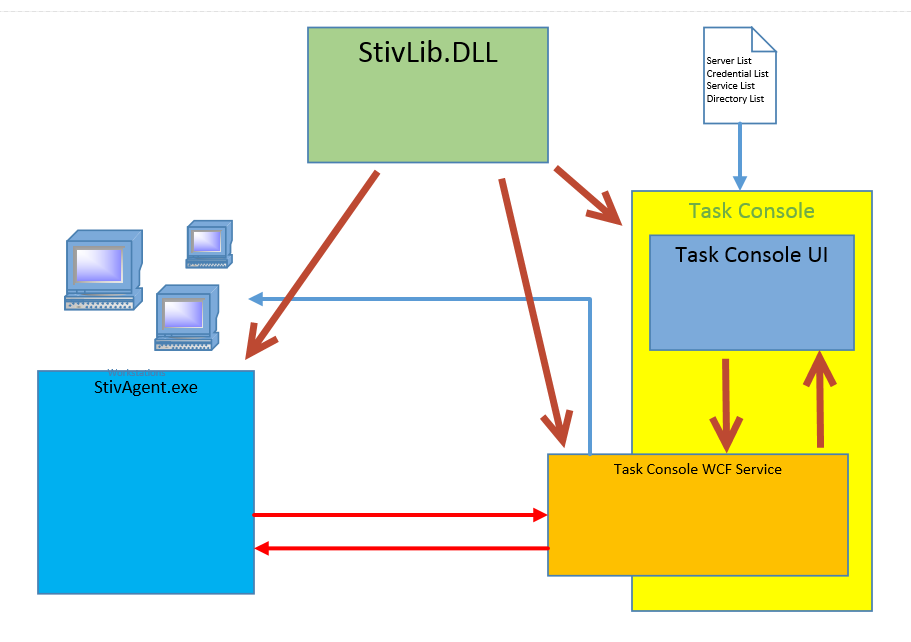
I had been working on multiple side projects over the years, and had been collecting useful functions into a personal library called “StivLib”. I am aware of a program called “PSExec” that can execute programs across the network, but it wasn’t native C# so I couldn’t work with it. After stumbling around with all sorts of different possibilities to accomplish what I wanted, I finally managed to get a function in my library called “RemExec” to work. This was the key to what I wanted, but I realized it would not be practical to manage the servers by RemExecing all of them. I also realized that I needed to use Threading if I wanted to get results from a large number of servers in a short time span. For this reason, I decided to use an Agent pushed to the servers to execute my commands locally. To keep it simple, the five parts of the solution work like this:

1: StivLib is a library of shared functions used across the other components. It contains the commands to DO things, like RemExec, Check Services, CheckIPAddress etc. It is my central storage for useful functions, and it does contain functions not used in this project.

2: Task Console WCF Service: This is the heart of the solution. The WCF Contract that drives the service is hosted inside the WPF application that holds the Console UI. Both the Console and the Agent talk through the contract to the WCF service to make things happen.

3: The TaskConsole WPF application: The console is responsible for giving the WCF service the information it needs to do its work. For instance, the Service needs to know what the current operating set of servers to use is. It needs to know what tasks to assign, what directories need to be scanned, what services should be monitored. It needs to know what credentials to use to connect to each server. This information is loaded in via the console, which also will query the service for status updates.

5: The StivAgent: This is the file that gets copied to the remote system, along with command line parameters telling it how to talk back to the WCF service. The Agent queries the Service every 5 seconds to see if it has a job to perform. If it does, the Service sends it a job object, which tells the client what it needs to do. The job object is populated with the results of the job, and when complete the job is sent back to the server. If the Agent is not able to talk to the server for a period of time, it will automatically kill and delete itself. The Agent also logs the tasks it performs into the Event log, along with the name of the person requesting them. The Agent is multithreaded, with one thread that monitors connection status and terminates everything if connection is lost, one thread that check for new jobs, and fires off a new worker thread for each new job received.



6: The document icon represents files that can be used to manage groups of configuration settings. The idea was that I needed to be able to work with different groups of Servers and Services, as well as different domains. I also wanted to keep a list of directories to monitor and rules for those directories that could be passed to the Agent on demand to do system checks.

# Working with the code

As a side project, I try to make sure that the program is flexible enough to be adapted to other uses. The key infrastructure is targeted at generic processes but to really be useful I wanted to make sure that it could be extended easily to do other things. As part of the way to test this, (and to make my life easier at work) I started adding functions specific to Ericsson’s Mediaroom product. While it is possible to do more, the way to add new simple tasks to the system is best analyzed by looking at the data contract portion of the WCF Contract. All communication is handled through the passing of Job objects (Part of the Jobber class from the contract). The object has a bunch of information in it related to the job in question. At the very basic level, all that is required is a TASKNAME. When a new job is created, the service will look through the list of servers currently in scope and queue a copy of that job to each of them. If you have added a new TASKNAME you next need to edit the Agent code

The agent listener will pick up that job at next check-in, update the status and time received, and send a copy back to the server to let it know the task was received. It then fires off a new worker thread and hands the job off to it.

In the worker thread is a case statement containing a list of commands the agent will handle. I prefer to add a function to each case statement rather than write the code there, so as to keep the code better organized. The function you add to handle a new task should perform a series of steps:

1. Update the status of the Jobber to Executing, and update the time of execution
2. Post the job with updated status back to the Task Console so we know it is being dealt with
3. Do whatever it is you need to do. If this is a multistep process, there is a field in the Jobber for listing out each step as it is completed.
4. Update the Jobber again with the results, mark as completed, set the completion status color, and status, completed time.
5. Post the completed job back to the service.