Bonus Chapter 1

Distributed Computing Using .NET Remoting

Light is the task when many share the toil.

Homer, *The Iliad*

Distributed computing is a model for solving computationally expensive problems by handing out small units of work to a network of computers, where each unit is solved and then combined back into the final computation. Distributed computing uses the resources of a large number of disparate computers connected to a network, treated as a virtual cluster. The network can be the Internet, in which case the appropriate term is *grid computing*, which is different from the traditional distributed computing approach because grid computing extends across administrative domains. Grid computing is sometimes confused with clustered computing, which is incorrect because grid computing is not restricted to a single administrative domain.

There are many successful grid computing projects available over the Internet, with perhaps the largest being the SETI@home project, which uses personal computers to analyze data from the Arecibo radio telescope in the hope of identifying signatures that indicate extraterrestrial intelligence. Currently, there are around half a million personal computers delivering 1,000 CPU years a day, which makes SETI@home the fastest special purpose computer in the world. There are many other projects available, such as the search for the largest prime number, solving protein folding, earthquake simulation, financial modeling, and weather modeling.

There is even talk that the Xbox 360 and PlayStation 3 may be able to support distributed computing, which would allow gamers to support projects like SETI@home when they are not using their consoles.

At a lower scale than grid computing, distributed computing can also be harnessed to solve computationally intensive problems for a game development studio over a local network. Current hardware offers much more processing power than it did a decade ago, but there is still a need to preprocess and store complex calculations prior to running the game. Calculations like computing potential visibility sets or calculating radiosity lightmaps for complex scenes are ideal candidates for preprocessing. Even the offline calculations take a considerable amount of time, and usually tie up the machine until the calculations are done.

Distributed computing can be used to manage the intermittent demands of these large computational exercises, drastically decreasing the computation time by spreading the work across multiple computers. Granted, moving complex calculations into a distributed environment is non-trivial, and must be done correctly. Distributed radiosity, and distributed solutions in general, can be quite tricky to implement correctly for a number of reasons. One reason is that all clients generally need the complete data set for the entire scene, which can be a costly packet to send to clients in certain environments. Of course, the algorithm itself must be rewritten to accommodate units of isolation for each client, which can be extremely difficult to develop and debug. You must also consider network performance, security, and scalability. If your network cannot adequately support a distributed model, you may, in fact, decrease your performance. The same concern goes along with the computers you are using, although generally you can solve the issue of underperforming computers by adding additional machines, which is fairly inexpensive.

The easiest type of calculation to distribute is a simple array of objects that can each execute independently of one another. Another applicable situation, albeit a little trickier, is a hierarchical algorithm, where you can pass a subset of the whole computation tree to a client by sending in a parent node containing child nodes. If implemented correctly, distributed computing can save significant hours of execution time, which will result in an increase in productivity.

This chapter covers further discussion of the usage and role of distributed computing, and then a relatively simple solution is presented that attempts to provide a framework upon which a distributed solution could be built and operated. The solution employs the use of .NET Remoting to communicate between server and clients. The solution is not overly complex, and it is missing functionality that would be present in a robust and mature middleware product.

Investigating Scalability

In systems analysis and design, scalability indicates the ability of a system to increase total throughput under an increased load and when hardware or software resources are added. Scalability is extremely important when working with distributed environments, because it implies performance. A distributed environment whose performance improves after adding resources proportionally to the capacity added is said to be *scalable*.

The concept of scalability can be measured in three categories:

- The first category is *load scalability*, where it should be easy to expand or contract the resource pool to accommodate heavier or lighter loads. In terms of distributed computing, this generally refers to adding and removing client machines from the environment. A distributed environment should be able to function normally, but with varying performance, independently of the number of available clients. Even situations where no clients are running should be considered, although it is quite obvious that processing would never complete.
- The second category is *geographic scalability*, where a scalable system should be able to maintain usefulness and usability no matter how far apart the resources are. Generally, game development studios would not employ clients outside of their own network, so this category is somewhat meaningless in this situation.
- The third category is *administrative scalability*, where it should be easy to use and manage a single distributed environment, regardless of how many organizations share it.

There is typically a loss in performance when a distributed environment scales in one or more of these categories. Therefore, it is important that the system itself be as optimized and scalable as reasonably possible.

Note

It is far better to focus on hardware scalability than capacity. It is generally much cheaper to improve performance by adding a new computer to the environment instead of performance tuning the system or the calculations to handle increased load.

There are two directions in which a distributed system can scale. A distributed system can scale vertically or up, which involves adding to or tweaking the code or

adding more memory or faster hardware to an existing computer. A distributed system can also scale horizontally or out, which involves adding new computers to the distributed environment.

Fault Tolerance

Perhaps the biggest concern when implementing a distributed system is fault tolerance and reliability of data. You can never be guaranteed that all the jobs will complete successfully, which can be caused by an exception in the calculation, a packet lost during communication, or a service interruption with the network or a computer on the network. Therefore, it is important to build a mechanism to handle fault tolerance. This mechanism can work in a variety of ways, but there are a couple of common situations that should be addressed by all mechanisms.

The first situation is when an exception occurs during processing as a result of the calculation. How this situation is handled depends on the calculation being performed and whether or not the unit of work can be restarted. The rule for this situation depends on implementation criteria for your system.

The next situation is when a computer on the network closes communication for an unknown reason, typically when a crash occurs. A common solution is to assign an estimated completion time for a job, and when a job has not completed by that time, you send out a heartbeat packet that the computer in question must reply to. If the packet is replied to, then the computer is still processing the job, and you can reset the estimated completion time. If the packet is not replied to, you can cancel the job and place it back into the pending jobs pool. Generally, distributed systems use pull technology rather than push, so you should not have to keep track of any particular computer. If the faulty computer restarts, it should be configured to relaunch the client and start pulling jobs again.

Another situation is when a packet is sent to a computer and is lost in transmission. You can extend the solution for when a computer goes offline by having the heartbeat packet reply with a status mentioning that it is still waiting for a job (meaning that it never received the job to process), and resend the job to the computer in question. You can go even further by having a confirmation packet that a client sends to the server when it receives a complete job packet.

Another possible situation is when the entire network goes down. The answer to this situation is generally straightforward, though. The easiest way is to cancel the entire calculation, log the failure, and restart the calculation when the network is back online.

The most important situation to consider is when the server itself crashes or goes offline. Imagine if the calculation results from each client are stored in memory on the server, and the server crashes at 97 percent. You would lose all that processing time, having to restart the entire calculation all over. Therefore, it is important to factor in a transaction system that persists completed jobs to a data store and enough information that a restarted server could recorrelate the results into the final result and restart any incomplete jobs if need be.

Managing Security

Security is not necessarily the largest concern when running lighting calculations in a game development studio, but sometimes it can be when dealing with sensitive data. If you are on an untrusted network and you are worried about job tampering, it is important that you manage network security appropriately. Network security is beyond the scope of this chapter, but you could consider implementing an account system into the clients so that completed jobs are marked with the computer identifier and the account of the session. This way, you can detect people who are injecting data into the results of the calculations. If acceptable, you can just stamp completed jobs with the Windows identity extracted from Active Directory.

Middleware Considerations

The solution provided in this chapter can be extended to build a custom solution, but some studios just want to get something up and running in the shortest amount of time possible. There are some middleware products available, many of them free, which provide a flexible interface for application composition, and a robust execution engine that has been thoroughly tested to account for fault tolerance, scalability, and performance.

If a middleware product appeals to you, then I suggest that you take a look at Alchemi, which is a flexible and robust middleware product that provides a distributed computing framework at an enterprise level. Alchemi is built on the .NET platform and makes use of .NET Remoting for job communication. Alchemi supports an object-oriented application programming model and a file-based job model. Cross-platform support is also provided by a web service layer, which can support the communication between other distributed systems. Alchemi also offers the capability for dedicated and non-dedicated (voluntary) execution by grid nodes. An excellent feature of Alchemi is that it is open-source, which means that it is much easier to adopt this technology into your studio instead of dealing with licensing issues.

Note

For more information about Alchemi, please visit **http://www.alchemi.net.**

Sample Framework

We will start off by covering the definition of what a job is, shown by the following interface. A job is a single unit of work and is contained within a job batch that can have one to many jobs. A job batch allows us to send chunky calls to clients, instead of doing it one job at a time (chatty calls), which causes a significant slowdown from the increase in network traffic.

```
/// <summary>Generic interface to represent a unit of work</summary>
public interface IDistributedJob
{
    /// <summary>Gets or sets the result of the job</summary>
    object Result
    {
        get;
        set;
    }

    /// <summary>Gets or sets the input data of the job</summary>
    object InputData
    {
        get;
        set;
    }

    /// <summary>Gets or sets the identifier for the associated batch</summary>
    Guid BatchId
    {
        get;
        set;
    }

    /// <summary>Gets or sets the identifier for the job</summary>
    Guid JobId
    {
        get;
        set;
    }
```

```
    /// <summary>Gets or sets the correlation id used for reordering</summary>
    long CorrelationId
    {
        get;
        set;
    }

    /// <summary>Gets or sets the time the job started processing</summary>
    DateTime StartTime
    {
        get;
        set;
    }

    /// <summary>Gets or sets the time the job finished processing</summary>
    DateTime FinishTime
    {
        get;
        set;
    }
}
```

The next interface to cover is the job batch, which is used to group a collection of jobs that will be sent to a client as a chunky call. Batches allow for correlation, in case jobs need to be reordered into a specific order based on an index or some other attribute, and assembly, which can be used to build a batch result if applicable.

```
/// <summary>Generic interface to represent a group of jobs</summary>
public interface IDistributedJobBatch
{
    /// <summary>Gets or sets the result for this batch of jobs</summary>
    object Result
    {
        get;
        set;
    }

    /// <summary>Gets or sets the input data for this batch of jobs</summary>
    object InputData
    {
        get;
        set;
    }
```

```
    /// <summary>Gets or sets the identifier of this batch</summary>
    Guid BatchId
    {
        get;
        set;
    }

    /// <summary>Gets or sets the array of jobs for this batch</summary>
    IDistributedJob[] Jobs
    {
        get;
        set;
    }

    /// <summary>This method can be used to reorder the jobs array
    /// before assembly. This can be based on the correlation id of
    /// the jobs if applicable.</summary>
    void Correlate();

    /// <summary>This method can be used to build the result from
    /// this batch if applicable. Some systems will only submit one
    /// job per batch or not care about the results of a batch.</summary>
    void Assemble();
}
```

The next interface is for distributed servers. Servers can sign out job batches, sign in job batches, and dismiss job batches.

```
/// <summary>Generic interface that servers must implement</summary>
public interface IDistributedJobServer
{
    /// <summary>Signs out an available batch from the
    /// server to process</summary>
    /// <returns>The job batch to process</returns>
    IDistributedJobBatch SignOut();

    /// <summary>Signs in a completed batch back into the server</summary>
    /// <param name="batch">The completed job batch to
    /// send back to the server</param>
    void SignIn(IDistributedJobBatch batch);

    /// <summary>Dismisses a job batch so the server can make
    /// it available again</summary>
```

```
    /// <param name="batch">The job batch to dismiss</param>
    void Dismiss(IDistributedJobBatch batch);
}
```

There is a lot of common functionality that will be present between all implementations of the distributed server using the supplied framework. The following code implements the base class that a distributed server inherits from. This base class handles the initialization of Remoting, keeping track of the available and unavailable job batches, and sets the lifetime of the Remoting proxy to infinite (until the proxy is explicitly released or the application is quit).

```
/// <summary>Base class that all distributed servers inherit from</summary>
public abstract class DistributedJobServer : MarshalByRefObject, IDistributedJobServer
{
    /// <summary>Remoting channel that the server will
    /// communicate with clients over</summary>
    private IChannel channel;

    /// <summary>A list of available job batches ready for processing</summary>
    protected readonly List<IDistributedJobBatch> availableBatches
                                      = new List<IDistributedJobBatch>();

    /// <summary>A list of unavailable job batches
    /// currently being processed</summary>
    protected readonly List<IDistributedJobBatch> unavailableBatches
                                          = new List<IDistributedJobBatch>();

    /// <summary>Gets the list of available job batches
    /// ready for processing</summary>
    public List<IDistributedJobBatch> AvailableBatches
    {
        get { return availableBatches; }
    }

    /// <summary>Adds a job batch to the available batches list</summary>
    /// <param name="jobBatch">The job batch to list</param>
    public void EnqueueJobBatch(IDistributedJobBatch jobBatch)
    {
        availableBatches.Add(jobBatch);
    }

    /// <summary>Loads settings from the configuration file
    /// and creates the remoted proxy server</summary>
```

```
/// <param name="configPath">The file system path to the
/// configuration file</param>
/// <returns>True if successful; false otherwise</returns>
public bool Publish(string configPath)
{
    try
    {
        DistributedJobServerConfig config
                                        = new DistributedJobServerConfig();

        if (!config.Load(configPath))
        {
            return false;
        }

        IDictionary bindings = new Hashtable();
        bindings["port"] = config["port"].ToString();
        bindings["name"] = string.Empty;

        string extension = string.Empty;

        switch (config["protocol"].ToString())
        {
            case "http":
                {
                    channel = new HttpChannel(bindings, null, null);
                    extension = ".soap";
                    break;
                }

            case "tcp":
                {
                    channel = new TcpChannel(bindings, null, null);
                    extension = string.Empty;
                    break;
                }
        }

        ChannelServices.RegisterChannel(channel);
        RemotingServices.Marshal(this, config["proxy"].ToString());
```

```
            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }

    /// <summary>Releases the remoting channel and remoted proxy</summary>
    public void Release()
    {
        RemotingServices.Disconnect(this);
        ChannelServices.UnregisterChannel(channel);
    }

    /// <summary>Signs out an available batch from the
    /// server to process</summary>
    /// <returns>The job batch to process</returns>
    public abstract IDistributedJobBatch SignOut();

    /// <summary>Signs in a completed batch back into the server</summary>
    /// <param name="batch">The completed job batch
    /// to send back to the server</param>
    public abstract void SignIn(IDistributedJobBatch batch);

    /// <summary>Dismisses a job batch so the server can
    /// make it available again</summary>
    /// <param name="batch">The job batch to dismiss</param>
    public abstract void Dismiss(IDistributedJobBatch batch);

    /// <summary>Informs remoting that we will manage the
    /// lifetime of the proxy (infinite lifetime)</summary>
    /// <returns>Null to signify infinite lifetime until we close</returns>
    public override object InitializeLifetimeService()
    {
        return null;
    }
}
```

Like the base class for distributed servers, the following code implements the base class that a distributed client inherits from. This base class handles the initialization of Remoting and retrieval of the remoted server proxy.

```
/// <summary>Base class that all distributed clients inherit from</summary>
public abstract class DistributedJobClient
{
    /// <summary>The remoted proxy reference to the distributed server</summary>
    private IDistributedJobServer proxy;

    /// <summary>The remoting channel to communicate with the server</summary>
    private IChannel channel;

    /// <summary>Gets the remoted proxy reference to
    /// the distributed server</summary>
    public IDistributedJobServer Proxy
    {
        get { return proxy; }

    /// <summary>Loads connection information from the
    /// config file and creates the remoted proxy reference</summary>
    /// <param name="configPath">The file system path to
    /// the configuration file</param>
    /// <returns>True if successful; false otherwise</returns>
    public bool Subscribe(string configPath)
    {
        try
        {
            DistributedJobClientConfig config = new DistributedJobClientConfig();

            if (!config.Load(configPath))
            {
                return false;
            }

            IDictionary bindings = new Hashtable();
            bindings["name"] = string.Empty;

            string extension = string.Empty;

            switch (config["protocol"].ToString())
            {
                case "http":
                    {
                        channel = new HttpChannel(bindings, null, null);
```

```
                        extension = ".soap";
                        break;
                    }

                case "tcp":
                    {
                        channel = new TcpChannel(bindings, null, null);
                        extension = string.Empty;
                        break;
                    }
            }

            ChannelServices.RegisterChannel(channel);

            string proxyAddress = String.Format("{0}://{1}:{2}/{3}{4}",
                                                config["protocol"],
                                                config["address"],
                                                config["port"],
                                                config["proxy"],
                                                extension);

            proxy = Activator.GetObject(typeof(IDistributedJobServer),
                                        proxyAddress) as IDistributedJobServer;

            return true;
        }
        catch (Exception)
        {
            return false;
        }
    }

    /// <summary>Unregisters the proxy and the remoting channel</summary>
    public void Unsubscribe()
    {
        ChannelServices.UnregisterChannel(channel);
    }
}
```

The example provided with the framework requires an implemented version of all the interfaces. The following code describes the example implementation of the job interface; the implementation is extremely simple.

```
/// <summary>Example implementation of the IDistributedJob interface</summary>
[Serializable]
public class ExampleJob : IDistributedJob
{
    /// <summary>The result of the job</summary>
    private object result;

    /// <summary>The input data of the job</summary>
    private object inputData;

    /// <summary>The time the job started processing</summary>
    private DateTime startTime;

    /// <summary>The time the job finished processing</summary>
    private DateTime finishTime;

    /// <summary>The identifier for the associated batch</summary>
    private Guid batchId;

    /// <summary>The identifier for the job</summary>
    private Guid jobId;

    /// <summary>The correlation id used for reordering</summary>
    private long correlationId;

    /// <summary>Gets or sets the result of the job</summary>
    public object Result
    {
        get { return result; }
        set { result = value; }
    }

    /// <summary>Gets or sets the input data of the job</summary>
    public object InputData
    {
        get { return inputData; }
        set { inputData = value; }
    }

    /// <summary>Gets or sets the time the job started processing</summary>
    public DateTime StartTime
    {
```

```
        get { return startTime; }
        set { startTime = value; }
    }

    /// <summary>Gets or sets the time the job finished processing</summary>
    public DateTime FinishTime
    {
        get { return finishTime; }
        set { finishTime = value; }
    }

    /// <summary>Gets or sets the identifier for the associated batch</summary>
    public Guid BatchId
    {
        get { return batchId; }
        set { batchId = value; }
    }

    /// <summary>Gets or sets the identifier for the job</summary>
    public Guid JobId
    {
        get { return jobId; }
        set { jobId = value; }
    }

    /// <summary>Gets or sets the correlation id used for reordering</summary>
    public long CorrelationId
    {
        get { return correlationId; }
        set { correlationId = value; }
    }
}
```

The following code describes the example implementation of the job batch interface. Correlation and assembly are not used in this example.

```
/// <summary>Example implementation of the IDistributedJobBatch interface</summary>
[Serializable]
public class ExampleJobBatch : IDistributedJobBatch
{
    /// <summary>The identifier of this batch</summary>
    private Guid batchId;
```

```
/// <summary>The array of jobs for this batch</summary>
private IDistributedJob[] jobs = new IDistributedJob[0] { };

/// <summary>The result for this batch of jobs</summary>
private object result;

/// <summary>The input data for this batch of jobs</summary>
private object inputData;

/// <summary>Gets or sets the identifier of this batch</summary>
public Guid BatchId
{
    get { return batchId; }
    set { batchId = value; }
}

/// <summary>Gets or sets the array of jobs for this batch</summary>
public IDistributedJob[] Jobs
{
    get { return jobs; }
    set { jobs = value; }
}

/// <summary>Gets or sets the result for this batch of jobs</summary>
public object Result
{
    get { return result; }
    set { result = value; }
}

/// <summary>Gets or sets the input data for this batch of jobs</summary>
public object InputData
{
    get { return inputData; }
    set { inputData = value; }
}

/// <summary>This method can be used to build the result
/// from this batch if applicable.
/// Some systems will only submit one job per batch or not
/// care about the results of a batch.</summary>
public void Assemble()
```

```
    {
        // Not used in this example
    }

    /// <summary>This method can be used to reorder the
    /// jobs array before assembly. This can be based on
    /// the correlation id of the jobs if applicable.</summary>
    public void Correlate()
    {
        // Not used in this example
    }
}
```

The example server provided with the sample framework has two events that send update notifications to the user interface to report on the completed and pending jobs. The following class describes the event arguments that are passed into the events.

```
/// <summary>Event arguments for user interface updates</summary>
public class UpdateJobsEventArgs : EventArgs
{
    /// <summary>The associated message to display</summary>
    private string message = string.Empty;

    /// <summary>Gets or sets the associated message to display</summary>
    public string Message
    {
        get { return message; }
        set { message = value; }
    }

    /// <summary>Constructor</summary>
    /// <param name="message">The associated message to display</param>
    public UpdateJobsEventArgs(string message)
    {
        this.message = message;
    }
}
```

The following code describes the example implementation of the distributed server that inherits from the base server class.

```
using DistributedJobSystem.Proxies;
/// <summary>Example implementation of the distributed server</summary>
```

```
public class TestServer : DistributedJobServer
{
    /// <summary>Event fired when the pending jobs label
    /// requires an update</summary>
    public event EventHandler<UpdateJobsEventArgs> UpdatePendingJobs;

    /// <summary>Event fired when the complete jobs label
    /// requires an update</summary>
    public event EventHandler<UpdateJobsEventArgs> UpdateCompletedJobs;

    /// <summary>Signs out an available batch from the server
    /// to process</summary>
    /// <returns>The job batch to process</returns>
    public override IDistributedJobBatch SignOut()
    {
        lock (availableBatches)
        {
            if (availableBatches.Count > 0)
            {
                IDistributedJobBatch batch = availableBatches[0];

                availableBatches.RemoveAt(0);
                unavailableBatches.Add(batch);

                lock (MainForm.MainFormInstance.PendingJobs)
                {
                    string countText
                          = MainForm.MainFormInstance.PendingJobs.Text;
                    int count = Convert.ToInt32(countText) - 1;

                    if (UpdatePendingJobs != null)
                        UpdatePendingJobs(this,
                                    new UpdateJobsEventArgs(count.ToString()));
                }

                return batch;
            }
            else
                return null;
        }
    }
```

```
/// <summary>Signs in a completed batch back into the server</summary>
/// <param name="batch">The completed job batch to
/// send back to the server</param>
public override void SignIn(IDistributedJobBatch batch)
{
    lock (unavailableBatches)
    {
        for (int batchIndex = 0;
                batchIndex < unavailableBatches.Count;
                batchIndex++)
        {
            if (batch.BatchId == unavailableBatches[batchIndex].BatchId)
            {
                unavailableBatches.RemoveAt(batchIndex);
                break;
            }
        }

        lock (MainForm.MainFormInstance.CompletedJobs)
        {
            // This could be improved
            string countText = MainForm.MainFormInstance.CompletedJobs.Text;
            int count = Convert.ToInt32(countText) + 1;

            if (UpdateCompletedJobs != null)
                UpdateCompletedJobs(this,
                                new UpdateJobsEventArgs(count.ToString()));
        }

        Application.DoEvents();
    }
}

/// <summary>Dismisses a job batch so the server
/// can make it available again</summary>
/// <param name="batch">The job batch to dismiss</param>
public override void Dismiss(IDistributedJobBatch batch)
{
    IDistributedJobBatch dismissedBatch = null;

    lock (unavailableBatches)
    {
```

```
            for (int batchIndex = 0;
                    batchIndex < unavailableBatches.Count;
                    batchIndex++)
            {
                if (batch.BatchId == unavailableBatches[batchIndex].BatchId)
                {
                    dismissedBatch = unavailableBatches[batchIndex];
                    unavailableBatches.RemoveAt(batchIndex);
                    break;
                }
            }
        }

        if (dismissedBatch != null)
        {
            availableBatches.Add(dismissedBatch);

            // This could be improved
            string countText = MainForm.MainFormInstance.PendingJobs.Text
            int count = Convert.ToInt32(countText) + 1;

            if (UpdatePendingJobs != null)
                UpdatePendingJobs(this,
                                new UpdateJobsEventArgs(count.ToString()));
        }
    }
}
```

The following code snippet describes the publishing logic for the example server user interface. Basically, 100,000 job batches, each with 10 jobs, are created and made available for clients to process. This example does not really do anything with results or input data, but the framework definitely supports both properties because they are somewhat important in a real solution!

```
private void PublishButton_Click(object sender, EventArgs e)
{
    PublishButton.Enabled = false;

    TestServer server = new TestServer();
    server.UpdatePendingJobs +=
        new EventHandler<UpdateJobsEventArgs>(server_UpdatePendingJobs);

    server.UpdateCompletedJobs +=
```

```
        new EventHandler<UpdateJobsEventArgs>(server_UpdateCompletedJobs);

    if (!server.Publish("..\\..\\ServerConfig.xml"))
        throw new Exception("Could not register");

    for (int batchIndex = 0; batchIndex < 100000; batchIndex++)
    {
        ExampleJobBatch jobBatch = new ExampleJobBatch();

        jobBatch.BatchId = Guid.NewGuid();
        jobBatch.Result = null;
        jobBatch.InputData = null;

        List<IDistributedJob> jobList = new List<IDistributedJob>();

        for (int jobIndex = 0; jobIndex < 10; jobIndex++)
        {
            ExampleJob job = new ExampleJob();

            job.BatchId = jobBatch.BatchId;
            job.JobId = Guid.NewGuid();
            job.CorrelationId = jobIndex;
            job.StartTime = DateTime.Now;
            job.Result = null;
            job.InputData = null;

            jobList.Add(job);
        }

        jobBatch.Jobs = jobList.ToArray();

        server.EnqueueJobBatch(jobBatch);
    }

    PendingJobs.Text = server.AvailableBatches.Count.ToString();

    MessageBox.Show("Done Startup");
}
```

The following code shows the example implementation of the distributed client that inherits from the base client class. Nothing really special here, but the implementation is shown to fill in the blanks. This class relies on the underlying functionality already implemented in the base class.

```
/// <summary>Example implementation of the distributed client</summary>
public class TestClient : DistributedJobClient
{
    // No specialized overrides needed for this example
}
```

The following code snippet shows the processing logic for the client user interface. This code signs out and processes jobs while they are available from the server. You can cancel and resume processing at any time as well. The example does not really process much on the jobs, but instead pauses for a little bit to simulate a real solution.

```
private void ProcessButton_Click(object sender, EventArgs e)
{
    try
    {
        ProcessButton.Enabled = false;
        CancelProcessButton.Enabled = true;
        stopped = false;
        int count = 0;

        TestClient client = new TestClient();
        client.Subscribe("..\\..\\ClientConfig.xml");

        ExampleJobBatch jobBatch = null;
        while ((jobBatch = (ExampleJobBatch)client.Proxy.SignOut()) != null)
        {
            // Do stuff with jobBatch here

            System.Threading.Thread.Sleep(1);
            count++;

            TreeNode batchNode = new TreeNode("Batch# " +
                                              jobBatch.BatchId.ToString());

            foreach (ExampleJob job in jobBatch.Jobs)
            {
                // Do stuff with each job here

                TreeNode jobNode = new TreeNode("Job# " +
                                                job.CorrelationId.ToString());
```

```
                batchNode.Nodes.Add(jobNode);
            }

            treeView1.Nodes[0].Nodes.Add(batchNode);
            treeView1.Nodes[0].Text = "Completed Batches - " +
                                      count.ToString();

            Application.DoEvents();

            try
            {
                Debug.Assert(jobBatch != null);
                client.Proxy.SignIn(jobBatch);
            }
            catch (Exception exception)
            {
                MessageBox.Show(exception.ToString());
            }

            if (stopped)
            {
                break;
            }
        }

        client.Unsubscribe();

        ProcessButton.Enabled = true;
        CancelProcessButton.Enabled = false;
    }
    catch (Exception exception)
    {
        MessageBox.Show(exception.ToString());
    }
}
```

The user interface for the distributed server example is shown in Figure 29.1.

The user interface for the distributed client example is shown in Figure 29.2.

Figure 29.1 Screenshot of the distributed server example.

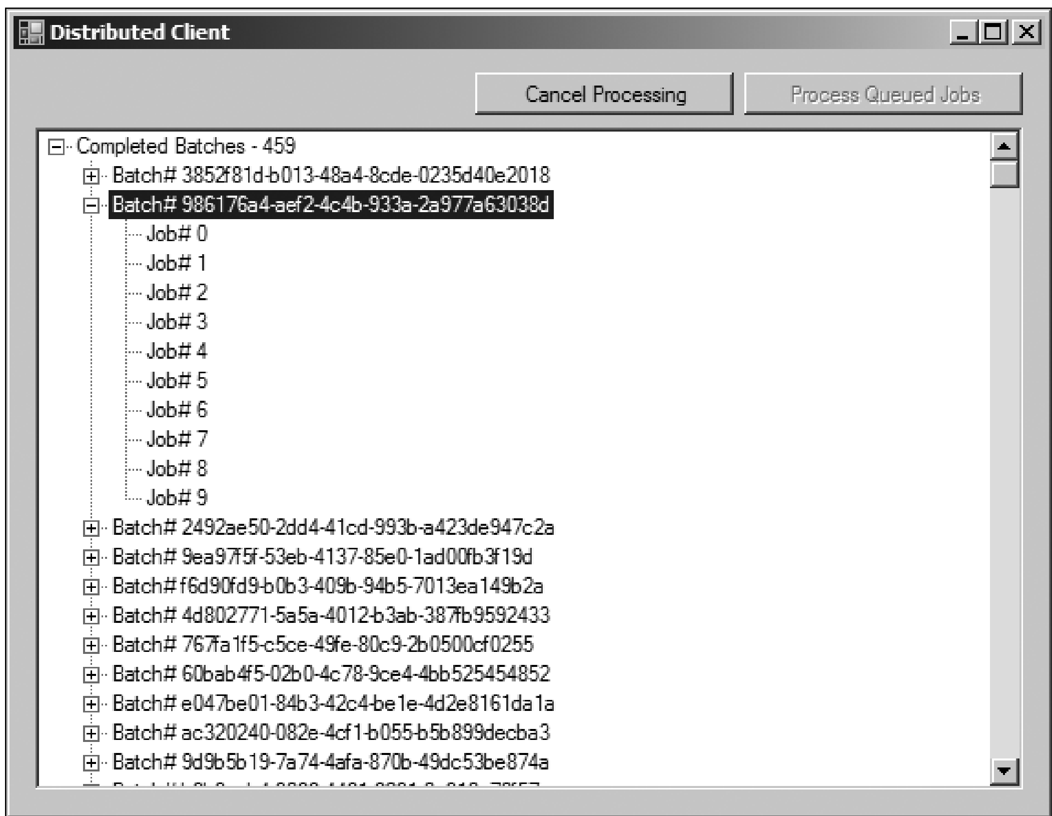

Figure 29.2 Screenshot of the distributed client example.

Conclusion

This chapter started off by discussing what distributed computing is, and its role in the game development industry. After which, some design considerations about scalability, fault tolerance, and security were covered that are important if you are planning on building a custom solution. Alchemi was then introduced as a viable .NET-based middleware solution, if that is the road you wish to take. Finally, a simple framework using .NET Remoting was presented that could be extended to support a custom solution. The presented solution, although fairly flexible, is still missing some important components such as fault tolerance and much better abstraction. It is merely presented as a building block with which you can build your own solution. Rather than a Windows Forms project for both the client and server, it would be recommended to use a Windows service for each process.

Note

> You will get much more performance out of a distributed system that sends large jobs to clients for processing, as opposed to a whole bunch of small jobs that finish quickly. You need to account for a small drop in performance with the network activity. Having large distributed jobs easily counteracts the small network performance costs.

The idea of distributed computing has been around for a number of years, although it is starting to exponentially pick up speed and adoption. A poorly designed solution can lead to enormous development and support headaches, though a properly designed solution can lead to significant cost and time savings, a huge gain for any game development studio.